Package ‘rmoriebricklayer’

October 31, 2026

Title Reproducible Data Capsules with Provenance and Fallback

Version 1.0.4

Description For analyzing published and unpublished tables, from open-data portals and statistical agencies to a researcher's own data, so that a result can be traced to the exact data behind it and checked again afterwards. A 'capsule' records where each source came from, its Secure Hash Algorithm 256 ('SHA-256') hash, the schema it should have and an archived copy in the Internet Archive 'Wayback Machine' ('web.archive.org'); sources are found through the Comprehensive Knowledge Archive Network ('CKAN', 'ckan.org'). Run again, a capsule says whether each source still matches and, if it does not, how its distribution moved (Kolmogorov-Smirnov, chi-square, population stability index and 'Benford' first-digit screens), and when a source cannot be reached it runs on schema-driven synthetic data that is marked as such. Capsules can be signed with keyed hashes ('HMAC-SHA-256', RFC 2104) or with the post-quantum signature standards 'ML-DSA' (FIPS 204), 'SLH-DSA' (FIPS 205) and 'XMSS' (RFC 8391), implemented from the standards with no system cryptography library and checked against 'OpenSSL' 3.5 and the known-answer vectors of 'NIST' (the US National Institute of Standards and Technology). For the administrative tables these capsules usually hold, it computes period-over-period change with exact intervals, trends (Mann-Kendall with 'Theil-Sen' slopes), rates, and the bounds that rounding and suppression in a release imply; small-area rates with the empirical Bayes shrinkage of Clayton and 'Kaldor' (1987) DOI 10.2307/2532003; self-exciting event series as 'Hawkes' processes; custodial populations by stock-and-flow measures; and the burden of air pollution from dispersion to attributable deaths.

License AGPL-3

Copyright The bundled excerpts of published statistics in inst/extdata belong to their publishers; see inst/COPYRIGHTS.

Encoding UTF-8

Language en-US

Depends R (>= 4.1.0)

Imports graphics, grDevices, methods, parallel, stats, tools, utils

Suggests withr, bslib (>= 0.9.0), digest, fixest, jsonlite, knitr, later, lme4, MASS, mgcv, nlme, nnet, openssl, pkgdown, rmarkdown, sf, promises, shiny, stringi, survival, survey, testthat (>= 3.0.0)

VignetteBuilder knitr

URL https://github.com/rootcoder007/rmorie-bricklayer, https://rootcoder007.github.io/rmorie-bricklayer/

BugReports https://github.com/rootcoder007/rmorie-bricklayer/issues

Config/testthat/edition 3

SystemRequirements C++17, libcurl (deb: libcurl4-openssl-dev, rpm: libcurl-devel, brew: curl), zlib (deb: zlib1g-dev, rpm: zlib-devel). Optional, for mkbndlchk() running a bundle: unshare (util-linux) and systemctl on Linux; podman or docker for container = TRUE.

Config/roxygen2/version 8.1.1

ByteCompile yes

NeedsCompilation yes

Author Vansh Singh Ruhela [aut, cre] (ORCID: <https://orcid.org/0009-0004-1750-3592>)

Maintainer Vansh Singh Ruhela <vsruhela@proton.me>

Repository CRAN

Date/Publication 2026-10-31 11:10:02 UTC

Contents

admits 10
adp 11
adpcnts 12
agentbndl 13
agestd 15
aipwcf 17
alos 19
anltab 20
arccap 23
arcfind 26
arcget 27
arcpush 28
asciifb 29
attrfrac 30
audcats 32
b64 34
bacondcmp 35
baltab 37

bandsens 38
bandvals 39
benfordtst 41
blcli 42
bldl 46
blfetch 48
blpal 50
blprog 52
bufasgn 53
capdeps 54
capenv 56
caseflow 57
cblake2b 59
cbootmean 60
ccovm 62
ccrc32 63
certchnvrf 64
certprs 67
cgamcdf 68
chgenv 70
chn 71
chnkfile 74
chwkfit 75
chwkjit 79
chwknll 80
chwkres 82
chwksim 84
cipw 85
citecps 86
clncolnms 88
clrdcmp 89
cmlrisk 91
cmom 93
cnormlpdf 94
cnormpdf 96
cnscwb 97
cnsprof 98
cnsstd 100
cnttrnd 101
co2fp 103
co2int 106
cohtrnd 107
commidx 109
concentration 111
connassoc 113
cortbl 115
cpsatst 116
cpsbndl 119

cpschk 122
cpsdrft 125
cpsfals 128
cpsplant 130
cpspwr 132
cpsrfrsh 134
cpsrpt 135
cpsseal 138
cpssign 139
cpsspec 142
cpsvrf 144
cpsxvrf 145
cramerv 147
crank 148
crfpm25 150
crmdens 152
crob 154
csha256 156
csha512 157
cspread 158
cstat 160
ctrm 161
cunif 163
cwbscr 164
cwtd 166
deccodes 167
declbld 168
dgstobj 170
dispmix 171
dldata 173
dlogin 174
dlogout 177
dmnf 178
drbggen 181
drbgnew 182
drbgrsd 184
drftapp 185
drftcal 186
drftchisq 188
drfthomog 190
drfthtml 192
drftks 194
drftpsi 195
drp 197
drvkey 199
dstfit 200
dstn 202
dstntst 204

dtctloc 206
dualcont 207
duprows 209
ebrates 210
envdiff 212
evlrr 213
evstudy 215
expbands 217
expcnts 218
expconcidx 220
explerr 221
exprspplr 223
exptrnd 225
falsfam 226
fdl 228
fetchvrf 229
filedgst 231
fipskey 232
fipskg 233
fipsmkg 235
fipsmod 236
fipsmu 238
fipssz 240
fnllim 241
fpeqv 243
freqtbl 244
frmaud 245
fromjs 247
fylbl 249
genexpo 250
ghann 251
grdbin 252
grdlvls 253
grdrcd 254
healchg 256
hilltlidx 258
holbal 260
hotpers 263
hqccmpkey 265
hqcdecap 267
hqcencap 268
hqckg 269
hqcsz 272
hurwzeta 273
idcrd 275
indgap 277
indiex 280
indtabs 282

infsch . . . 282
inlhist . . . 284
instcli . . . 286
jsgz . . . 287
jsserdoc . . . 289
justice-data . . . 290
kemdecap . . . 292
kemencap . . . 293
kemkg . . . 294
kemsz . . . 296
keyeddgst . . . 298
knoxtst . . . 299
ldgrread . . . 301
lifetbl . . . 303
limfrom . . . 305
llmask . . . 306
llmcfg . . . 309
llmchat . . . 311
llmmdls . . . 313
llmprvd . . . 314
llmrte . . . 315
llmstat . . . 317
loadprov . . . 318
loounit . . . 319
ltrlrng . . . 321
mahaoutl . . . 327
mcartst . . . 329
mhcont . . . 331
missgrid . . . 333
misspat . . . 334
missruns . . . 336
misssmry . . . 337
mkbndl . . . 338
mkbndlchk . . . 341
mkbndlproj . . . 343
mkmnf . . . 344
mkset . . . 346
mksyncol . . . 347
mksyncsv . . . 349
mnfcanon . . . 351
mnfdiff . . . 352
mnfrcmp . . . 354
mnfrecseed . . . 356
morani . . . 357
mortdisp . . . 359
mrk . . . 360
mtchdid . . . 362
onl . . . 365

optsqr 367
oqskg 368
orchk 369
ordcmp 371
perdays 373
perminf 374
pgsig 376
piperate 380
plmfld 382
plndid 383
plndml 386
plnpois 388
plnpwr 390
plot.rmbl_census_cwb 393
plot.rmbl_dstfit 407
plotevgrid 408
plothwk 409
plotlrnz 410
plotmore 412
plotseries 414
plrcf 415
polbrdarea 417
polbrdn 418
polex 420
pqcbknd 421
pqckg 422
print.bricklayer_attestation 424
profcols 428
profshr 430
prrgdecl 432
prsbands 433
pubbnd 435
qltyrcd 437
ralloc 438
rallocvrf 440
raplok 441
ratechg 442
rategap 445
rcdmnf 446
recevid 447
recrd 448
relbl 450
rgncov 451
rgnmapalt 452
rgnmapcmp 454
rgnmapintg 456
rgnmappts 457
rlblfrns 459

rmblapp 461
rmbldemo 463
rndbytes 464
rptanl 466
rptmd 467
rslvarc 468
rslvckan 469
rslvckns 471
rslvsoc 472
rsv 473
rtcalc 475
rul 477
rulelib 478
rvkfetch 481
scanadj 483
scests 484
schemapath 487
schval 487
sctidy 489
sevidx 490
sha256file 491
shrs 493
signpubkey 495
siudate 496
siufetch 497
siufprs 498
siuprs 499
siusch 501
siuso 502
siutxt 503
sprd 504
srchfig 506
staysmry 507
stdinc 508
stdprev 510
stepchg 512
stgatt 514
stkflow 516
stpdnpv 518
svcs 520
svycmp 521
svydes 523
svyest 526
swtdid 528
synctl 530
syndid 532
toascii 534
tojs 535

topcors 537
trccmpl 539
trchlthgap 540
trcind 542
trcprog 543
trndtst 544
tse 546
tsstmp 548
tsvrf 549
usecpstmpl 552
valrules 553
valsch 555
vrfcps 557
vrfmrg 559
vrfpol 561
vrfrcd 564
vrfsha256 565
wbrtry 566
wbsave 568
wbsnpnat 571
wbsnpurl 572
wcb 573
welprof 575
wrmnfjs 578
wrrcdmnf 579
wrsmrytxt 580
wrtxtfb 582
xfrvrf 583
xmssadv 585
xmssidxseen 587
yoy 588
yoybnd 593
yoydelim 594
yoylbl 596
yoypals 597
yoypv 598
yoyren 599
yoysmry 602
yoywr 603

Index **605**

admits *Admissions implied by a population and a length of stay*

Description

Works out how many admissions a given average daily population and average length of stay imply over a period. Published statistics often give the population held and the average stay but not the number admitted (or the other way round); this recovers the missing quantity from the identity that links all three.

Usage

```
admits(adp, alos, t = 365)
```

Arguments

adp	Numeric scalar, the average daily population (people held on an average day); must be positive.
alos	Numeric scalar, the average length of stay in days; must be positive.
t	Numeric scalar, the length of the period in days; must be positive. Default 365.

Details

Rearranging the identity $\bar{X}_{hc} = N_a \bar{X}_t / t$ (Lakner 1976, p.18-20) for N_a. Useful when two of the three quantities are published and the third is not. The identity holds exactly only in a steady state (admissions and stays roughly constant across the period); with a population that is growing or shrinking quickly the result is an approximation.

Errors when any argument is not numeric, is NA or Inf, is not positive, or has more than one value.

Value

A numeric scalar: the implied number of admissions over the period, $\mathrm{adp} \times t / \mathrm{alos}$. It is not rounded. It has class c("rmbl_admits", "rmbl_measure", "numeric") and an attribute "lakner" (adp, alos and t), so it prints as a box with its inputs, formula and caveats; arithmetic, round(), sum(), mean(), [, c() and as.numeric() give a plain number.

See Also

adp(), alos(), stkflow()

Examples

```
# Lakner (p.20) runs this the other way: t = 365, an average daily
# population of 25 and 1,750 admissions imply a stay of 5.2 days.
alosimplied <- 25 * 365 / 1750
round(alosimplied, 1)
```

```
# and back again
admits(25, alosimplied)

# a 90-day quarter with 40 held on average and a 12-day average stay
admits(40, 12, t = 90)

# an average stay of zero is refused
try(admits(25, 0))
```

---

adp *Average daily population*

Description

The mean number of person-days served per day over a period: a STOCK, answering how many people are held at one time. Use it to report how much confinement (or bed occupancy, shelter use, open caseload) a system used over a year or any other window, from the days each person spent in it. It is the measure capacity planning needs, because it counts beds in use rather than people who passed through.

Usage

```
adp(days, t = 365)
```

Arguments

`days`	Numeric vector of person-days served during the period, non-negative, no `NA` or `Inf`. The vector is summed, so one element per person and a single pre-computed total give the same result.
`t`	Numeric scalar, the length of the period in days; must be positive. Default `365`. A vector of several values is refused. `perdays` turns two dates into this.

Details

This is Lakner's $\bar{X}_{hc} = \sum X_i / t$ (1976, p.15), where X_i is the days served by person i inside the period. The denominator is TIME, which is what makes it a stock. Compare `alos()`, whose denominator is people: the two are linked by $\mathrm{ADP} = N \times \mathrm{ALOS}/t$ (Little's law), so `admits()` and `stkflow()` can move between them.

Only days served inside the period should be counted: a stay that began before the period or ends after it contributes its days within the window.

Errors when `days` is not numeric (text and factors are refused), is empty, has `NA`, `Inf` or a negative value, or when `t` is not a single positive number.

Value

A numeric scalar: person-days per day, that is the average number of people held on any one day of the period. It has class `c("rmbl_adp", "rmbl_measure", "numeric")` and an attribute `"lakner"` (the total days and `t`), so it prints as a box with its inputs, formula and caveats; arithmetic, `round()`, `sum()`, `mean()`, `[`, `c()` and `as.numeric()` give a plain number.

References

Lakner, E. (1976) *A Manual of Statistical Sampling Methods for Corrections Planners*. University of Illinois at Urbana-Champaign.

See Also

`alos()`, `stkflow()`, `adpcnts()`

Examples

```
# Lakner's own worked example (p.15): 3,000 inmates served 13,500
# detention days in a year.
adp(13500)

# per-person days give the same total
adp(c(10, 20, 30), t = 30)

# a leap year, with the period length taken from the dates
adp(13500, t = perdays("2024-01-01", "2024-12-31"))

# negative days are refused
try(adp(c(10, -2)))
```

---

adpcnts *Person-days estimated from periodic headcounts*

Description

When only a count of people present on certain days is available – not a record per person – the person-days over the period are the mean of those counts scaled to the period's length. Use it with `adp()` or `alos()` when the source is a daily or weekly count sheet rather than admission and release dates.

Usage

```
adpcnts(cnts, t = 365)
```

Arguments

`cnts`	Numeric vector of headcounts, one per day on which a count was taken; non-negative, no `NA` or `Inf`. The days need not be consecutive.
`t`	Numeric scalar, the length of the period in days; must be positive. Default `365`.

Details

Lakner's $X_t = (\frac{1}{C}\sum N_i)t$ (1976, p.21). The counts need not cover every day: counting on weekdays only is the usual case, and the mean of the days counted stands in for the days not counted. That substitution is an assumption, and it fails if the days counted differ systematically from the days missed – weekday-only counting where weekend admissions are released before Monday, for instance.

Value

A numeric scalar: the estimated person-days over the period, `mean(counts) * t`. Despite the function's name this is a total of person-days, not an average daily population; divide by `t` (or pass it to `adp()`) for the population held on an average day. It has class `c("rmbl_adpcnts", "rmbl_measure", "numeric")` and an attribute `"lakner"` (the number of counts, their mean and `t`), so it prints as a box with its inputs, formula and caveats; arithmetic, `round()`, `sum()`, `mean()`, `[`, `c()` and `as.numeric()` give a plain number, and `adp()` takes it as days.

See Also

`adp()`

Examples

```
# Lakner (p.21): counts on 255 days summing to 34,935 imply just over
# fifty thousand detention days across the year.
adpcnts(rep(34935 / 255, 255))

# weekly counts over a 31-day month, then the average daily population
pd <- adpcnts(c(120, 130, 125, 128), t = 31)
pd
adp(pd, t = 31)

# a missing count must be dropped or filled first
try(adpcnts(c(120, NA, 125)))
```

`agentbndl` *Agent-assisted reproducibility-bundle help*

Description

Asks a language model to plan a brick-proof, reproducible data bundle, through `llmask`: your own key, your own endpoint or a local Ollama server, with the packages' help at hand (the model reads the help pages of the functions it proposes before it answers). Use it to get a first draft of the steps and R code for packaging an analysis: finding the portal's download URL, fetching with a Wayback fallback, recording hashes and a manifest, writing a synthetic stand-in, then packaging, signing and verifying the bundle.

Usage

```
agentbndl(req, model = NULL, route = NULL)
```

Arguments

`req`	Character scalar describing the bundle task.
`model`	Optional model id, a string; `NULL` uses the route's model (`rmbl models` lists them).
`route`	`NULL` (the saved route, else the first that is set up), `"auto"`, `"own"`, `"ollama"` or a provider id such as `"anthropic"` or `"openai"`.

Details

The request is sent with a fixed system prompt that describes that task and asks for a short plan and R code. The route is chosen as in `llmask()`; when no route is set up, the function returns a message saying what to set up (a provider key, a local Ollama or your own endpoint) instead of an error, and makes no network request. When a route is set up, the request and the prompt are sent to that provider or server. The answer is model output: check the code it proposes against the help pages before running it.

Errors: `req` not a single non-empty string; `route` not one of the names listed; `route = "hosted"` (the hosted tier is retired); and errors raised by `llmask()`.

Value

Character scalar: the answer, or a message saying what to set up when no route answers.

Examples

```
# Needs a language model: a provider key (e.g. ANTHROPIC_API_KEY), your own
# endpoint or a local Ollama.
agentbndl("scaffold a bundle for analysis.R from the Toronto CKAN data")
agentbndl("sign and verify my capsule", route = "ollama")


# With every route switched off (no key in the environment and no saved
# settings: the settings folder is pointed at an empty one) the call returns
# a setup hint, not an error, and reaches no network -- safe to run anywhere:
keys <- c(
  "OLLAMA_HOST",
  "MORIE_LLM_BASE_URL",
```

```
  "XDG_CONFIG_HOME",
  llmprvd()$key_env,
  llmprvd()$alt_env[nzchar(llmprvd()$alt_env)]
)
old <- Sys.getenv(keys, unset = NA)
Sys.setenv(
  OLLAMA_HOST = "off",
  MORIE_LLM_BASE_URL = "off",
  XDG_CONFIG_HOME = tempfile()
)
Sys.unsetenv(setdiff(
  keys,
  c("OLLAMA_HOST", "MORIE_LLM_BASE_URL", "XDG_CONFIG_HOME")
))
cat(agentbndl("hello"))
for (v in keys) {
  if (is.na(old[[v]])) {
    Sys.unsetenv(v)
  } else {
    do.call(Sys.setenv, as.list(old[v]))
  }
}

# argument errors come first
try(agentbndl(""))
try(agentbndl("plan a bundle", route = "hosted"))
```

`agestd` *Directly age-standardised rates, with Fay-Feuer gamma intervals*

Description

The rate a population would have with the age structure of a standard population, for one or more groups, with Fay and Feuer's gamma confidence interval.

Usage

```
agestd(events, pop, std, grp = NULL, per = 1e+05, conflvl = 0.95)
```

Arguments

`events`, `pop`	Numbers of events and of people, one per age group (and group).
`std`	The standard population, one count per age group (its shares are the weights).
`grp`	Optional labels: one rate per group, each over the same age groups in the same order.
`per`	Rate per this many people (default 100,000).
`conflvl`	Confidence level of the Fay-Feuer intervals (default 0.95).

Details

With d_i events and n_i people in age group i and standard weights w_i (the standard population's shares), the rate is $y = \sum w_i d_i/n_i$, its variance $v = \sum (w_i/n_i)^2 d_i$, and with $w_M = \max_i w_i/n_i$ the limits are $\frac{v}{2y}\chi^2_{2y^2/v}(\alpha/2)$ and $\frac{v+w_M^2}{2(y+w_M)}\chi^2_{2(y+w_M)^2/(v+w_M^2)}(1-\alpha/2)$ (Fay and Feuer 1997). The interval is conservative (coverage at least nominal in Fay and Feuer's simulations) and reduces to the exact Garwood interval for a single age group: suited to the few events of small communities. With no events the lower limit is 0. Rates are per `per` people. Age groups are matched to `std` by position: order the rows of each group by age, in the order of `std`.

The choice of standard matters: Statistics Canada's comparisons of First Nations, Metis and Inuit mortality standardise to the Indigenous population's own age structure (Kumar and Tjepkema 2019), which is younger than Canada's.

Value

A data frame with `group`, `events`, `rate`, `lower`, `upper`, and the attributes `var` (the variances, on the per-person scale) for `rategap`.

References

Fay MP, Feuer EJ (1997). Confidence intervals for directly standardized rates: a method based on the gamma distribution. Statistics in Medicine 16(7):791-801.
DOI 10.1002/(sici)1097-0258(19970415)16:7<791::aid-sim500>3.0.co;2-#

Kumar MB, Tjepkema M (2019). Suicide among First Nations people, Metis and Inuit (2011-2016): Findings from the 2011 Canadian Census Health and Environment Cohort (CanCHEC). Statistics Canada 99-011-X2019001.

See Also

`rategap`.

Examples

```
# made-up counts for one small community, by five age groups, to show the
# computation
std <- c(30000, 25000, 20000, 15000, 10000) # a younger standard population
agestd(
  events = c(4, 9, 11, 6, 3),
  pop = c(9000, 8000, 6000, 4000, 2000),
  std = std
)
```

`aipwcf` *Cross-fitted augmented inverse probability weighting for a binary treatment*

Description

The average effect of a binary treatment (on everyone, or on the treated) from one row per person, with the outcome and the propensity fitted by learners on other folds, combined in the doubly robust (Neyman-orthogonal) score of Robins, Rotnitzky and Zhao; with survey or sampling weights if given.

Usage

```
aipwcf(
  data,
  y,
  trt,
  covs,
  estd = c("ATE", "ATT"),
  lrn = "lasso",
  basis = c("linear", "poly2"),
  w = NULL,
  des = NULL,
  folds = 5L,
  trim = 0.01,
  seed = 1L,
  conflvl = 0.95
)
```

Arguments

`data`	A data frame, one row per person.
`y`, `trt`	Column names: the outcome and the 0/1 treatment.
`covs`	Column names of the confounders (numeric; code categories as indicators).
`estd`	`"ATE"` (default) or `"ATT"`.
`lrn`	The outcome learner (as in `plndml`).
`basis`	`"linear"` (default) or `"poly2"`.
`w`	Optional column name of survey or sampling weights.
`des`	Optional survey design from `svydes`, built on `data`, in place of `w`: the standard error then comes from its replicate weights or its strata and primary sampling units.
`folds`	Cross-fitting folds (default 5).
`trim`	Propensity clipping (default 0.01).
`seed`	Seed for the folds (default 1).
`conflvl`	Confidence level of the interval (default 0.95).

Details

With $\hat{\mu}_d(X)$ the outcome fitted among those with treatment d and $\hat{e}(X)$ the propensity, both out of fold, the score for the average effect is

$$\psi = \hat{\mu}_1 - \hat{\mu}_0 + \frac{D(Y - \hat{\mu}_1)}{\hat{e}} - \frac{(1 - D)(Y - \hat{\mu}_0)}{1 - \hat{e}} - \theta,$$

and for the effect on the treated

$$\psi = \frac{D(Y - \hat{\mu}_0) - \hat{e}(1 - D)(Y - \hat{\mu}_0)/(1 - \hat{e})}{P(D = 1)} - \frac{D}{P(D = 1)}\theta.$$

$\hat{\theta}$ solves the weighted mean of the score at zero; its variance is the weighted mean of ψ^2 over n, the influence-function variance of Chernozhukov et al. (2018). Propensities are clipped to `[trim, 1 - trim]`. The estimator is consistent when either the outcome or the propensity model is right, and with cross-fitting the learners can be flexible (lasso or ridge on a second-order basis, or any learner function).

The average effect equals `DoubleML`'s interactive regression model given the same folds and learners; for the effect on the treated, `DoubleML` estimates $P(D = 1)$ within each fold and this function in the whole sample, so the two differ slightly.

The propensity learner is logistic regression on the covariates (with `basis = "poly2"`, on their squares and products too). With `w` the point estimate is design-weighted, but the standard error treats the weights as fixed and the sample as independent.

For a complex survey, pass the design from `svydes` instead: its weight gives the point estimate and its replicate weights (or strata and primary sampling units) the standard error, with the variance formula of the replicate type, the confidence interval on the design's degrees of freedom. The learners are fitted once, without the weights, which enter only the average of the scores, so each replicate estimate is the replicate-weighted average of the same cross-fitted scores: the same numbers as re-running the function on every replicate weight, from one fit instead of one per replicate. Rows with a missing value are left out as a domain (their scores count as zero in every replicate), not dropped from the design.

Value

A data frame of class `rmbl_aipw`: `estimand`, `estimate`, `se`, `lower`, `upper`, `p_value`; attributes `n`, `treated`, `psrange`, `trimmed` (how many propensities were clipped), `variance` (`"influence"`, or the design's type) and `propensity` (a data frame of each row's out-of-fold propensity before clipping, its treatment and its weight, which `plot()` draws).

References

Robins JM, Rotnitzky A, Zhao LP (1994). Estimation of regression coefficients when some regressors are not always observed. Journal of the American Statistical Association 89(427):846-866. DOI 10.1080/01621459.1994.10476818

Chernozhukov V et al. (2018), see `plndml`.

See Also

`plrcf`, `plndml`, `baltab`.

Examples

```
set.seed(2)
n <- 800
d <- data.frame(age = runif(n, 18, 70), income = rlnorm(n, 10, 0.5))
d$belong <- rbinom(n, 1, plogis(-2 + 0.03 * d$age + 0.2 * log(d$income)))
d$wellbeing <- 50 + 5 * d$belong + 0.2 * d$age + 2 * log(d$income) +
  rnorm(n, 0, 8)
aipwcf(d, "wellbeing", "belong", c("age", "income"), lrn = "ols")
# with 50 bootstrap replicate weights: one fit, the replicate standard error
d$w <- runif(n, 0.5, 2)
reps <- sapply(1:50, function(b) d$w * rpois(n, 1))
colnames(reps) <- paste0("bsw", 1:50)
d <- cbind(d, reps)
des <- svydes(d, w = "w", reps = colnames(reps))
aipwcf(d, "wellbeing", "belong", c("age", "income"), lrn = "ols",
              des = des)
```

---

alos *Average length of stay*

Description

The mean number of person-days served per person: the FLOW side of the same person-days. Use it to say how long a typical admission lasts, and together with `adp()` to tell whether a change in the population held came from more people or from longer stays. For the full distribution of stays (median, spread, interval) use `staysmry()`.

Usage

```
alos(days, n)
```

Arguments

`days`	Numeric vector of person-days served during the period, non-negative, no `NA` or `Inf`; the vector is summed.
`n`	Numeric scalar, the number of people; must be positive. A vector of several values is refused. For an unbiased average this should be the people both admitted AND released within the period, because anyone still held has an unfinished stay.

Details

Lakner's $\bar{X}_t = \sum X_i / N'$ (1976, p.16), with a caveat worth repeating (p.16-17): the period must be longer than the longest stay people actually serve, or the average is biased DOWNWARD, since the longest stays are the ones that fail to finish inside the window. For short-stay facilities a year is comfortable; for long sentences it is not, and the period has to be set from the records.

Errors when `days` is not numeric, is empty, or has `NA`, `Inf` or negative values, and when `n` is not a single positive number.

Value

A numeric scalar: days per person, $\sum X_i/n$. It has class `c("rmbl_alos", "rmbl_measure", "numeric")` and an attribute `"lakner"` (the total days and `n`), so it prints as a box with its inputs, formula and caveats; arithmetic, `round()`, `sum()`, `mean()`, `[`, `c()` and `as.numeric()` give a plain number.

References

Lakner, E. (1976) *A Manual of Statistical Sampling Methods for Corrections Planners*. University of Illinois at Urbana-Champaign.

See Also

`adp()`, `stkflow()`

Examples

```
# Lakner (p.17): 2,700 inmates admitted and released served 12,150
# detention days between them.
alos(12150, 2700)

# from one value per released person
stays <- c(2, 5, 1, 30, 4, 7)
alos(stays, length(stays))

# zero people cannot have an average stay
try(alos(12150, 0))
```

`anltab` *Analyse a published administrative table in one call*

Description

The questions asked of a published table are always the same: what is in it, what changed between periods and how sure can one be, what the rates are if there is an exposure, whether a short series trends, and whether this release differs from the one held before. Each has a function in this package; this runs them together, with the pieces a reviewer needs and that are easy to forget on a deadline: exact intervals on every change, multiple-comparison adjustment over the groups scanned, the envelope that rounding and suppression in the release imply, and a drift screen against the prior capsule.

Usage

```
anltab(
  data,
  value,
  prd,
```

```
  by = NULL,
  pop = NULL,
  prior = NULL,
  units = c("count", "continuous", "percent"),
  rnd = NULL,
  rndkind = c("nearest", "random"),
  suplim = NULL,
  dir = c("neutral", "higher", "lower"),
  per = 1000,
  conflvl = 0.95,
  alpha = 0.05,
  adjust = "BH",
  prdord = NULL
)
```

Arguments

`data`	A data frame in long form: one row per period and group.
`value`	Name of the column (a string) with the published count (or measure).
`prd`	Name of the column with the period (fiscal year, quarter, ...); numeric periods are used as the time axis of the trend tests. Text periods (`"2019/20"`, `"2021 Q3"`, `"Mar 2021"`) are ordered as `yoy()` reads them (see its `prd`), or as `prdord` gives.
`by`	Optional character vector of grouping column names (institution, region, ...). `NULL` (the default) treats the table as one series.
`pop`	Optional name of an exposure column for rates (`NULL`, the default, skips the rate step).
`prior`	Optional data frame: the previous capsule of the same table, for the drift screen. `NULL` (the default) skips it.
`units`	`"count"` (the default), `"continuous"` or `"percent"`, as in `yoy()`.
`rnd, rndkind, suplim`	How the release rounds or suppresses cells; see `pubbnd()`. `rnd` is the rounding base (for example 5), `rndkind` `"nearest"` (the default) or `"random"`, `suplim` the threshold below which cells are suppressed. `NULL` when the counts are exact. Used only for counts: with other `units` they are ignored, with a warning.
`dir`	Which way is an improvement, as in `yoy()`: `"neutral"` (the default), `"higher"` or `"lower"`.
`per`	Rate denominator, as in `rtcalc()`; default 1000 (rates per 1,000 of pop).
`conflvl`	Confidence level for every interval, default 0.95.
`alpha`	Level for the multiple-comparison decision and the drift screens, default 0.05.
`adjust`	Multiple-comparison method for `scanadj()`. The default is `"BH"` (Benjamini-Hochberg).
`prdord`	Optional character vector: the period labels in order, for text periods the reader does not know (see `yoy()`); passed to `yoy()`, `ratechg()` and the trend tests.

Details

The steps, in order:

1. profile: `profcols()` and `misssmry()` on `data`;
2. change: `yoy()` between consecutive periods within each `by` group. For `units = "count"` the exact p-values are then adjusted across all comparisons with `scanadj()`, and, when `rnd` or `suplim` is given, `yoybnd()` adds the publication envelope (see `pubbnd()`). For `"continuous"` and `"percent"` units neither step runs: `rnd` and `suplim` are then ignored, with a warning;
3. rates (only with `pop`): `rtcalc()` per group and period, and `ratechg()` between periods, adjusted with `scanadj()`;
4. trend: per group, the values summed within each period, then `trndtst()` (Mann-Kendall tau, p-value and Theil-Sen slope) and, for counts, `cnttrnd()` (Poisson rate ratio per period). A group with fewer than three periods gets `NA`; a test that fails gets `NA` rather than an error;
5. drift (only with `prior`): `cpsdrft()` of `data` against `prior`.

Nothing is written and nothing is fetched; a progress bar for the trend step is drawn on stderr unless `options(morie.quiet = TRUE)`. Errors: `data` or `prior` not a data frame; a named column not in `data`; an invalid `units`, `dir` or `rndkind`; a `conflvl` outside (0, 1); and any error of the functions called (for example negative counts in `yoy()`).

Value

A list of class `bricklayer_analysis` (with a print method that summarises it in a few lines):

`profile` `profcols()` of `data`.

`missingness` named numeric from `misssmry()`.

`change` the `yoy()` table; for counts with the added columns `p_value`, `padj` and `significant`, and with `env_*` and `compctlo`/`compctup` when publication bounds were requested.

`rates`, `ratechg` the `rtcalc()` and `ratechg()` tables when `population` is given, otherwise `NULL`.

`trend` data frame with one row per group: the `by` columns, `periods`, `first`, `last`, `tau`, `trndp`, `slope`, `rateratio`, `rateratiolo`, `rateratioup`.

`drift` `cpsdrft()` result against `prior`, or `NULL`.

`meta` list of the arguments used, `nrows`, the sorted `periods`, `prior` (logical) and `when` (a time stamp).

See Also

`rptanl()` writes the result as a Markdown or HTML report.

Examples

```
path <- system.file("extdata", "otis_a01_individuals.csv",
                    package = "rmoriebricklayer")
otis <- read.csv(path)
a <- anltab(otis, value = "individuals", prd = "year",
                  by = c("table", "group"))
a
```

```
a$change[, c("table", "group", "year", "pctchg", "padj")]
a$trend[, c("group", "tau", "trndp", "rateratio")]

# counts published rounded to base 5: the intervals widen
b <- anltab(otis, value = "individuals", prd = "year",
                  by = c("table", "group"), rnd = 5)
b$change[1:3, c("group", "year", "pctlo", "compctlo")]

# rates with an exposure column
d <- data.frame(year = rep(2019:2022, 2), site = rep(c("A", "B"), each = 4),
                n = c(10, 12, 15, 9, 30, 28, 35, 40),
                pop = rep(c(1000, 2000), each = 4))
r <- anltab(d, "n", "year", by = "site", pop = "pop", per = 1000)
r$ratechg[, c("site", "year", "rate", "rateratio", "padj")]

# a column that is not in the data is an error
try(anltab(otis, value = "count", prd = "year"))
```

`arccap` *Capture files the Internet Archive cannot reach, and publish them as an archive.org item*

Description

Downloads each URL from here, records its SHA-256, size and time in a manifest, timestamps the manifest with an RFC 3161 time-stamping authority, and, with archive.org keys, uploads the files, the manifest and its timestamp as one public archive.org item. The item is the mirror that `fetchvrf` falls back to, and the timestamp shows that these were the bytes on that date.

Usage

```
arccap(
  urls,
  dir,
  item = NULL,
  meta = list(),
  tsa = "https://freetsa.org/tsr",
  ak = NULL,
  sk = NULL,
  tmout = 600L,
  upload = NULL,
  reason = NULL,
  visibility = c("public", "private"),
  warc = TRUE
)
```

Arguments

urls	URLs to capture, or a wbsave result (its refused URLs are taken).
dir	A folder for the files and the manifest (created if needed).
item	Optional archive.org item identifier (5 to 100 letters, digits, dots, hyphens, underscores) to upload to; NULL (default) captures without uploading.
meta	Named metadata for a new item (title, creator, licenseurl, description, subject, ...); mediatype defaults to "data".
tsa	The time-stamping authority. "https://freetsa.org/tsr" is the default; NULL means no timestamp.
ak, sk	archive.org keys; default from the environment as wbsave reads them.
tmout	Seconds allowed for each download and each upload (default 600).
upload	Where else to keep the captures: "rmorie" (archive.rmorie.com, from this connection, with your key; see dlogin), "wayback" (ask the Wayback Machine to save the same addresses, with archive.org keys as for wbsave), both, or NULL (default: only the local folder and, with item, archive.org).
reason	Why the files are kept, one sentence of 20 characters or more; needed with upload = "rmorie", and written into each WARC file.
visibility	"public" (default: listed in the archive's search; anyone can download and check it) or "private" (only you, with your key).
warc	Write each capture as a WARC/1.1 file with a CDXJ index (dir/archive/, dir/indexes/index.cdxj), the layout pywb and warcio read and fetchvrf falls back to (default TRUE).

Details

Some servers refuse the Internet Archive's crawler while serving everyone else: Save Page Now then reports that the target "cannot find" the page, or "blocks access" to it, although the page is there. Those URLs cannot be put in the Wayback Machine, but they can be captured by anyone who can reach them, and the capture can be published. Pass a wbsave result as urls to capture just the URLs it failed on with such a refusal.

The manifest (manifest.json) lists, for each URL, the file, its SHA-256, its size and the time it was fetched. The timestamp (manifest.tsr) is the authority's signed statement that the manifest's SHA-256 existed at its time; tsvrf checks it, as does openssl ts -verify. Each file's digest in the manifest ties the file to that time.

The upload uses the Internet Archive's S3-like API: each file is sent with a PUT, streamed from disk, to
https://s3.us.archive.org/<item>/<file>.
The first request creates the item with the metadata given. The keys are the same as for wbsave, so they are read from IA_S3_ACCESS_KEY and IA_S3_SECRET_KEY, or their _FILE twins. They travel only in the authorization header and are never written or printed. Publish only what the source's licence lets you redistribute.

Value

A data frame of class `rmbl_archive_capture`, one row per URL: `url`, `file`, `sha256`, `bytes`, `fetched_utc`, `status` (`"captured"`, `"uploaded"`, `"archived"` or `"failed: ..."`), `mirror` (the archive.org download address, for `fetchvrf`), `warc` (the local WARC file), `capture_id` and `archive` (its page at archive.rmorie.com) and `wayback` (the Wayback Machine's answer); attributes `manifest`, `timestamp` (a `tsvrf` result, or `NULL`) and `item`.

References

Adams C, Cain P, Pinkas D, Zuccherato R (2001). Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP). RFC 3161. DOI 10.17487/RFC3161

Internet Archive. ias3: the Internet Archive's S3-like API. archive.org/developers/ias3.html

See Also

`fetchvrf`, `wbsave`, `tsvrf`.

Examples

```
# save from your own connection to archive.rmorie.com and the Wayback Machine
# at once
src <- "https://www150.statcan.gc.ca/n1/tbl/csv/35100177-eng.zip"
cap <- arccap(
  src,
  file.path(tempdir(), "cap"),
  upload = c("rmorie", "wayback"),
  reason = "Custody table cited in chapter 3"
)
cap$archive
w <- wbsave(urls)
lic <- "https://www.statcan.gc.ca/en/terms-conditions/open-licence"
meta <- list(
  title = "Statistics Canada tables (captures)",
  creator = "Statistics Canada",
  licenseurl = lic
)
cap <- arccap(
  w,
  file.path(tempdir(), "capture"),
  item = "my-statcan-tables-2026",
  meta = meta
)
fetchvrf(
  cap$url[1],
  tempfile(),
  sha256 = cap$sha256[1],
  mirrors = cap$mirror[1]
)
```

`arcfind` *Find saved copies of an address in archive.rmorie.com*

Description

Lists the captures the archive holds of a web address, oldest first: when each was taken, its HTTP status, media type, size and SHA-256, and the addresses to open or download it. An address ending in `*` lists every address that starts with it. Public captures are listed for anyone; with your key (see `dlogin`) your private ones are listed too.

Usage

```
arcfind(url, from = NULL, to = NULL, limit = 1000L)
```

Arguments

`url`	A web address, or the start of one followed by `*`.
`from`, `to`	Optional limits on the capture time, as `"YYYY"`, `"YYYYMM"` ... up to `"YYYYMMDDhhmmss"` (UTC).
`limit`	The most captures to list (default 1000).

Details

The archive answers the same CDX query as the Wayback Machine (`/cdx/search/cdx?url=...&output=json`), so tools written for it can read it as well. Set `MORIE_ARCHIVE_URL` to query another archive that speaks the same protocol, with its key in `MORIE_ARCHIVE_KEY`.

Value

A data frame of class `rmbl_archive_find`: `capture_id`, `captured` (UTC time), `url`, `status`, `mime`, `bytes`, `sha256`, `mode` (`"client"`: saved from the capturer's connection; `"worker"`: saved by the archive), `visibility`, `capture` (its page) and `replay` (its exact bytes).

See Also

`arcget`, `arccap`, `fetchvrf`.

Examples

```
caps <- arcfind("https://www150.statcan.gc.ca/n1/tbl/csv/35100177-eng.zip")
caps
arcfind("www150.statcan.gc.ca/n1/tbl/*")
```

`arcget` *Fetch the exact bytes of a saved copy from archive.rmorie.com*

Description

Downloads a capture's bytes and checks their SHA-256: against the archive's record, and against `sha256` when you give one (the pin from your analysis). A file that does not match is deleted.

Usage

```
arcget(x, dest, sha256 = NULL, at = NULL, tmout = 600L)
```

Arguments

`x`	A capture id (`"20261021133005-1a2b3c4d"`, from `arcfind` or a capture page) or a web address; for an address, the capture closest to `at` (default: the latest) is taken.
`dest`	The file the fetched bytes are written to.
`sha256`	Optional SHA-256 the bytes must have.
`at`	For an address, the time to be closest to, `"YYYY..."` up to 14 digits (UTC).
`tmout`	Seconds allowed for the download.

Value

`dest`, invisibly, with attributes `capture_id`, `url`, `captured` and `sha256`.

See Also

`arcfind`, `fetchvrf` (which falls back to the archive on its own).

Examples

```
src <- "https://www150.statcan.gc.ca/n1/tbl/csv/35100177-eng.zip"
f <- arcget(src, tempfile(fileext = ".zip"))
attr(f, "sha256")
```

`arcpush` *Send a capture kept on this computer to the archives later*

Description

`arccap()` keeps every capture on this computer first: the files, their SHA-256, a manifest and, by default, a WARC copy. When the Wayback Machine, archive.rmorie.com or archive.org is down, or there is no network but the source's, the capture is still complete there. `arcpush()` sends that same capture on later: the very files that were fetched, each checked first against the SHA-256 the manifest recorded, so an archive keeps what you captured, not a new download. Run it again to retry what failed; what an archive already took is not sent twice.

The Wayback Machine takes no uploads: with `upload = "wayback"` it is asked to capture each address itself, as the page is then. If the page has changed since, its copy differs from yours, and your copy, with the manifest's timestamp, remains the record of what you had.

A manifest with no timestamp yet (the authority was unreachable, or `tsa = NULL` at capture) is stamped now when `tsa` is given; that timestamp shows the files existed by now, not by the time they were fetched.

Errors: `dir` without the '`manifest.json`' that `arccap()` writes, or a manifest that lists no files; nowhere to send (neither `item` nor `upload`); as for `arccap()` for `item`, `meta`, `upload`, `reason`, `tmout` and the keys.

Usage

```
arcpush(
  dir,
  item = NULL,
  meta = list(),
  ak = NULL,
  sk = NULL,
  upload = NULL,
  reason = NULL,
  visibility = c("public", "private"),
  tsa = "https://freetsa.org/tsr",
  tmout = 600L
)
```

Arguments

`dir`	The folder of an `arccap()` capture.
`item`, `meta`, `ak`, `sk`, `upload`, `reason`, `visibility`, `tmout`	As for `arccap()`.
`tsa`	A time-stamping authority, asked only when the manifest has no timestamp yet (default as for `arccap()`); `NULL` asks none.

Value

An object of class `rmbl_archive_capture`, as `arccap()` returns: one row per file of the capture, with what each archive answered, this time or an earlier one. A file that is missing, or whose bytes no longer match the manifest, is `"failed: ..."` and is not sent. What each archive took is also kept in 'push.json' in `dir`, which the next run reads.

See Also

`arccap()` makes the capture; `wbsave()` and `wbrtry()` for the Wayback Machine alone; `fetchvrf()` falls back to the copies.

Examples

```
# capture now, on this computer only
src <- "https://www150.statcan.gc.ca/n1/tbl/csv/35100177-eng.zip"
d <- file.path(tempdir(), "cap")
arccap(src, d, tsa = NULL)
# later, when the archives answer: the same bytes, sent on
arcpush(
  d,
  upload = c("rmorie", "wayback"),
  reason = "Custody table cited in chapter 3"
)
```

---

asciifb *Use Text As-Is, Falling Back to ASCII When It Cannot Be Represented*

Description

Returns `x` unchanged when it is valid, well-formed text (so legitimate UTF-8 such as an accented name is preserved), and only transliterates to plain ASCII via `toascii()` when the text is not valid UTF-8 (an encoding error) or when `force = TRUE` (for ASCII-only destinations such as a package `DESCRIPTION`) . This lets author and supervisor names keep their accents wherever UTF-8 is supported while degrading gracefully instead of erroring where it is not.

Usage

```
asciifb(x, force = FALSE)
```

Arguments

`x`	A character vector (other types are converted with `as.character()`).
`force`	A single logical: `TRUE` always transliterates every element to ASCII; `FALSE` (the default) only the elements that are not valid UTF-8. Anything other than `TRUE` counts as `FALSE`.

Details

Each element is tested on its bytes with `validUTF8()`, so the outcome does not depend on the session locale. Valid elements (including plain ASCII and properly encoded accented text) are returned unchanged; invalid ones go through `toascii()`, which drops the bytes that are not UTF-8 and transliterates the rest. With `force = TRUE` every element goes through `toascii()`. A missing value is kept as `NA` unless `force = TRUE`, when it becomes `""`.

Value

A character vector the length of `x`: each element unchanged when it is valid UTF-8 (and `force` is not `TRUE`), otherwise its ASCII transliteration.

See Also

`toascii()`, which does the transliteration.

Examples

```
# By default valid UTF-8 is preserved (accents kept where supported).
asciifb("\u00c1ngela")              # "\u00c1ngela"

# force = TRUE always transliterates (for ASCII-only destinations
# such as a package DESCRIPTION).
asciifb("\u00c1ngela", force = TRUE)  # "Angela"

# Plain ASCII is returned unchanged either way.
asciifb("plain name")

# Vectorised; each element handled independently.
asciifb(c("caf\u00e9", "resume"), force = TRUE)

# Bytes that are not UTF-8 (a Latin-1 file read as UTF-8) are repaired.
bad <- rawToChar(as.raw(c(0x63, 0x61, 0x66, 0xe9)))
validUTF8(bad)
asciifb(bad)
```

---

attrfrac *Population attributable fraction*

Description

The share of all cases in a population that is due to an exposure: the fraction that would not occur if the exposure were removed and nothing else changed. It combines how strongly the exposure raises risk (the relative risk) with how common it is (the prevalence); multiplied by the number of cases it gives the attributable cases, as `polbrdn` does.

Usage

```
attrfrac(rr, expoprev, rrlo = NULL, rrup = NULL)
```

Arguments

`rr`	Relative risk among the exposed, a single finite positive number. A relative risk below 1 gives a negative fraction (a protective exposure); 0, negative, infinite and missing values are errors.
`expoprev`	Proportion of the population exposed, in `[0, 1]`.
`rrlo`, `rrup`	Optional confidence limits of `rr`; when both are given the fraction carries the limits that the same formula gives at them (Levin's formula is monotone in the relative risk, so the interval of the fraction is the image of the interval of the risk).

Details

Levin's formula (Levin 1953; Rothman, Greenland and Lash 2008, chapter 5):

$$PAF = p(RR - 1)/(1 + p(RR - 1)),$$

with p the proportion exposed. With everyone exposed it is $1 - 1/RR$. It assumes the relative risk is causal and unconfounded and that removing the exposure changes nothing else. A relative risk below 1 gives a negative fraction (the exposure protects).

When both confidence limits of the RR are given, the fraction's limits are the formula applied to them: the formula increases with the RR, so the interval of the fraction is the image of the interval of the risk. Refusals: an `rr` or limit that is not a single finite positive number (logical values included), a prevalence outside $[0, 1]$, only one of the two limits, or limits that do not enclose `rr`.

Value

A single number, the attributable fraction: in `[0, 1)` for `rr >= 1`, negative for a protective exposure. When the limits were given it carries an attribute `ci`, a named numeric vector (`lower`, `upper`). It has class `c("rmbl_attrfrac", "rmbl_measure", "numeric")`, so it prints as a box with its inputs (kept in attribute `"lakner"`); arithmetic, `[`, `c()` and `as.numeric()` give plain numbers.

References

Rothman, K. J., Greenland, S. and Lash, T. L. (2008). Modern Epidemiology, 3rd ed., chapter 5. Levin, M. L. (1953). The occurrence of lung cancer in man. Acta Unio Internationalis Contra Cancrum 9, 531-541.

See Also

`crfpm25` for a relative risk from a concentration, `polbrdn` for the attributable cases.

Examples

```
attrfrac(1.5, 0.4)
attrfrac(2, 1)      # everyone exposed: 1 - 1/RR

# With the confidence limits of the relative risk.
paf <- attrfrac(1.08, 1, rrlo = 1.06, rrup = 1.09)
attr(paf, "ci")

# A protective exposure gives a negative fraction.
attrfrac(0.8, 0.5)

# A prevalence above 1 is refused.
try(attrfrac(1.5, 40))
```

---

audcats *Audit the categorical columns of a data frame for coding hazards*

Description

Checks every categorical column of a data frame for the coding problems that silently change results: numbers that are really category codes whose labels were lost, labels still in foreign value-label form, labels that differ only by case or by an invisible space, empty strings or "NA" stored as a category, unused levels, and identifiers mistaken for categories. Run it once after every import, before any model. It also reports which level R will use as the reference category.

Usage

```
audcats(data, cols = NULL)
```

Arguments

data	A data frame whose categorical columns are audited.
cols	Character vector of column names to audit, default NULL: every factor, character or haven-labelled column. Columns of other types can be named explicitly and are audited by their distinct values.

Details

For each column the levels are levels(x) for a factor and the sorted distinct values (radix sort, as factor() would order them) otherwise. A hazard is reported when:

- the column is haven labelled (it still carries value labels and must be decoded with deccodes or declbld);
- every level looks like a number (^[0-9.]+$);
- every level starts with a code prefix such as "1. White";

- a level has leading or trailing whitespace, including non-breaking and other Unicode spaces, or two levels are equal once trimmed;
- a level is empty, or is a missing-value sentinel (`NA`, `N/A`, `NAN`, `NULL`, `NONE`, `.`, `-`, `?`, compared case-insensitively);
- the reference (first) level is empty, a sentinel, or padded;
- two different trimmed levels agree once lower-cased;
- a factor has levels that never occur;
- there are more than 50 levels.

The function only reports; it never changes `data`. It errors when `data` is not a data frame.

Value

A data frame of class `c("bricklayer_category_audit", "data.frame")`, one row per audited column, with character columns `column`, `storage` (the column's class), `levels` (up to eight levels joined with `"|"`), `reference` (the first level) and `hazards` (all hazards joined with `" ; "`, or `""`), and integer `n_levels`. The attribute `clean` is `TRUE` when no hazard was found. The print method lists each column and its hazards.

See Also

`deccodes`, `grdlvls`, `xfrvrf`.

Examples

```
# codes without labels, and a case-variant city name
a <- audcats(data.frame(
  race = factor(c("1", "2", "2", "3")),
  city = c("Toronto", "toronto", "Ottawa")[c(1, 2, 3, 3)]
))
a
attr(a, "clean")

# padded, empty and sentinel labels, and an unused level
audcats(data.frame(
  g = factor(c(" A", "A", "NA", ""), levels = c("", " A", "A", "NA", "B"))
))

# a clean frame; non-categorical columns are skipped by default
audcats(data.frame(sex = c("F", "M", "F"), age = c(31, 45, 27)))

try(audcats(1:3))
```

b64 *Base64 encoding*

Description

Encodes and decodes base64, in both the standard alphabet and the URL-safe variant. Computed in R with no dependency, so a capsule can embed binary content in a text manifest wherever this package runs.

Usage

```
jsb64enc(input)

jsb64dec(input)

jsb64uenc(input)

jsb64udec(input)
```

Arguments

input	For the encoders, a raw vector, or a character vector (its elements are joined with newlines and encoded as UTF-8 bytes), or NULL. For jsb64dec(), base64 text (a character vector, joined with newlines) or its raw bytes. For jsb64udec(), URL-safe base64 text, with or without = padding.

Details

jsb64enc() breaks its output into 72-character lines (a newline after each complete block of 54 input bytes), matching jsonlite's encoder byte for byte; the decoder ignores line breaks, other white space and = padding, so either form round trips. Any other character outside the alphabet is an error rather than being skipped, because skipping it would decode a corrupted body into a shorter valid-looking one.

The URL-safe variant substitutes - and _ for + and / and drops the = padding, which is what makes it safe in a URL path, a query string or a filename.

Base64 is an ENCODING, not encryption or a digest: it hides nothing and anyone can reverse it. Use csha256() to pin content and chmac() to authenticate it.

Errors: an encoder given anything but a raw vector, a character vector or NULL; jsb64dec() given anything but text or raw bytes, or text with a character outside the standard alphabet; jsb64udec() given anything but character text (raw bytes are refused), NA, or a character outside the URL-safe alphabet.

Value

The encoders return a character scalar ("" for empty input, NA_character_ for NULL); the decoders return a raw vector (use rawToChar() for text).

See Also

jsgzenc(), which composes this with gzip.

Examples

```
# Round trip through the standard alphabet.
b <- jsb64enc("hello capsule")
b
rawToChar(jsb64dec(b))

# Raw input works the same way.
jsb64enc(charToRaw("abc"))
jsb64dec(jsb64enc(charToRaw("abc")))

# Padding appears when the length is not a multiple of three.
jsb64enc("a")
jsb64enc("ab")
jsb64enc("abc")

# The URL-safe variant has no "+", "/" or "=" to escape.
jsb64uenc(as.raw(c(255, 224, 63)))
jsb64enc(as.raw(c(255, 224, 63)))
jsb64udec(
  jsb64uenc("path/safe?yes")
)

# Long input is wrapped, and the decoder ignores the breaks.
long <- jsb64enc(strrep("x", 200))
grepl("\n", long)
rawToChar(jsb64dec(long)) == strrep("x", 200)

# A character outside the alphabet is refused, not skipped.
try(jsb64dec("ab$c"))
```

---

bacondcmp *The Goodman-Bacon decomposition of a two-way fixed-effects estimate*

Description

Splits the two-way fixed-effects difference-in-differences estimate of a staggered adoption into the two-group, two-period comparisons it averages, with their weights: treated against never treated, earlier against later treated, and later against earlier treated (the comparisons that use already-treated units as controls).

Usage

```
bacondcmp(data, unit, time, y, trtd)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`trtd`	Column name of the treatment indicator (0 or 1).

Details

For a balanced panel with treatment that stays on and no covariates, the estimate of `plndid` is exactly $\hat{\beta} = \sum_k s_{kU}\hat{\beta}_{kU} + \sum_{k \neq l} s_{kl}\hat{\beta}_{kl}$, with weights that sum to one and grow with the groups' sizes n and the variance of their treatment over time $\bar{D}(1-\bar{D})$ (Goodman-Bacon 2021, theorem 1). Units treated from the first period act as an already-treated group. A large weight on "later vs earlier" comparisons, together with effects that grow over time, is the warning sign that the two-way estimate is biased.

Value

A data frame of class `rmbl_bacon`: `type` (`"treated vs never"`, `"earlier vs later"`, `"later vs earlier"`), `treated`, `control` (the adoption periods; `Inf` for never), `estimate`, `weight`; attribute `twfe` (the weighted sum, equal to the two-way estimate).

References

Goodman-Bacon A (2021). Difference-in-differences with variation in treatment timing. Journal of Econometrics 225(2):254-277. DOI 10.1016/j.jeconom.2021.03.014

See Also

`plndid`, `stgatt`.

Examples

```
p <- polex()
b <- bacondcmp(p, "district", "month", "crmrate", "reformed")
b
tapply(b$weight, b$type, sum)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
```

```
  0.2 * own$income + sin(seq_len(80))
bacondcmp(own, "town", "year", "rate", "bylaw")
```

`baltab` *A covariate balance table*

Description

Means of each covariate in the treated and comparison groups, their standardised mean difference, and Welch's test, optionally weighted (after matching or weighting).

Usage

```
baltab(data, grp, covs, w = NULL)
```

Arguments

`data`	A data frame, one row per unit.
`grp`	Column name of the 0/1 group (1 = treated).
`covs`	Column names to compare.
`w`	Optional column name of weights.

Details

The standardised mean difference is $(\bar{x}_1 - \bar{x}_0)/s_1$ with s_1 the unweighted standard deviation in the treated group (the convention for the effect on the treated, so that it does not move when weights change the comparison group); $|SMD| < 0.1$ is the usual threshold for balance. With weights the means are weighted and the test uses the weighted variances and Kish's effective sample sizes.

Value

A data frame of class `rmbl_balance`: `covariate`, `meantrtd`, `meanctl`, `smd`, `p_value`.

References

Welch BL (1947). The generalization of 'Student's' problem when several different population variances are involved. Biometrika 34(1-2):28-35. DOI 10.1093/biomet/34.1-2.28

Examples

```
d <- data.frame(t = rep(0:1, each = 50),
                age = c(stats::rnorm(50, 30), stats::rnorm(50, 33)))
baltab(d, "t", "age")
```

bandsens *How much a result depends on the open band's assumed cap*

Description

Recomputes a statistic of banded data (by default the Gini coefficient) for a range of assumed caps on the open top band, and reports how far the answer moves. Every statistic computed from banded data depends on where the open band is assumed to end; this function measures that dependence, so you can report a range instead of a single figure when the dependence is large.

Usage

```
bandsens(bands, cnts, stat = ginicoef, caps = NULL, rule = "midpoint")

## S3 method for class 'rmbl_band_sensitivity'
format(x, digits = 4L, ...)

## S3 method for class 'rmbl_band_sensitivity'
print(x, ...)
```

Arguments

bands	A data frame from prsbands, or labels to parse.
cnts	Numeric vector, one per band: how many units fall in each band. Non-integer counts are truncated by rep().
stat	A function that takes a numeric vector of per-unit values and returns one number. Default ginicoef.
caps	Numeric vector of caps to try, in the units of the labels. Default NULL: the geometric sweep described in Details. Values are sorted and de-duplicated.
rule	Character, default "midpoint": the placement rule passed to bandvals.
x	An rmbl_band_sensitivity object.
digits	Significant digits of the printed numbers.
...	Further arguments to print(); the band table ignores them.

Details

Let b be the smallest lower bound among the open top bands. The default caps are 12 points spaced evenly on the log scale from $1.05b$ to $20b$. For each cap c, bandvals(bands, rule, openupcap = c) gives one value per band, each value is repeated cnts times, and stat is applied to the result. A statistic that errors or does not return a single number gives NA; warnings are suppressed.

The span is $\max - \min$ over the finite values and the relative span is the span divided by the median absolute value (NA when any value is zero). The print method adds a note when the relative span exceeds 0.1.

Errors: cnts not numeric or of a different length from the bands; no open top band.

Value

A data frame of class `c("rmbl_band_sensitivity", "data.frame")` with numeric columns `cap` and `value`, and attributes `span`, `relspan` and `base` (the lower bound b of the open band).

Examples

```
b <- prsbands(c("1", "2 to 5", "6 to 10", "Greater than 10"))
counts <- c(1200, 430, 110, 38)

s <- bandsens(b, counts)
s
attr(s, "relspan")

# A statistic that barely moves has been measured; one that swings
# has not.
bandsens(b, counts, stat = mean, caps = c(12, 20, 50))

# nothing to vary without an open band
try(bandsens(c("1", "2 to 5"), c(10, 4)))
```

---

bandvals *Representative values for banded categories*

Description

Gives each band a single number that a calculation can use (a mean, a concentration index, a regression), under a rule you choose. For closed bands the rule places the value at the midpoint, the lower or upper bound, or the geometric mean of the bounds. An open band (`"50+"`, "under 18") has no midpoint, so its missing bound is filled from an explicit assumption, and the rows that rest on that assumption are flagged in the result instead of being absorbed into it.

Usage

```
bandvals(
  bands,
  rule = c("midpoint", "lower", "upper", "geometric"),
  openupcap = NULL,
  openupfac = 2,
  openloflr = NULL
)
```

Arguments

`bands`	A data frame from `prsbands`, or a character vector of labels to parse with its defaults.

rule	Character, one of "midpoint" (the default), "lower", "upper" or "geometric". The geometric mean suits quantities whose distribution inside a band is closer to log-uniform than uniform, which is usual for counts and durations.
openupcap	Number or NULL (the default). The upper bound assumed for every open top band, in the units of the labels.
openupfac	Number, default 2. When no cap is given, the open top band's upper bound is its lower bound times this factor (so "Greater than 10" becomes 11 to 22).
openloflr	Number or NULL (the default). The lower bound assumed for an open bottom band ("under 18"). NULL uses 0, or the band's own upper bound when that is below 0. A floor above a band's upper bound is refused.

Details

For a band with bounds $[l, u]$ the value is $(l+u)/2$ ("midpoint"), l ("lower"), u ("upper") or $\sqrt{lu}$ ("geometric"; when $l \leq 0$ the arithmetic midpoint is used instead, because the geometric mean is undefined there). The "lower" and "upper" rules bracket any value that could come from the bands.

An open upper band gets $u =$ openupcap if given, else $u = l \times$ openupfac. An open lower band gets $l =$ openloflr if given, else $\min(0, u)$. A cap below the band's lower bound, or a floor above its upper bound, is an error, because the value would lie outside the band. Unparsed bands give NA.

Use bandsens to see how much a result moves with the cap, and expbands to turn the values into one value per unit.

Value

The band table with two added columns: value (numeric, the representative value) and assumed (logical, TRUE for the open bands whose value rests on the cap or floor).

See Also

prsbands, bandsens, expbands

Examples

```
b <- prsbands(c("1", "2 to 5", "6 to 10", "Greater than 10"))
bandvals(b)

# The lower and upper rules bracket the truth.
bandvals(b, rule = "lower")$value
bandvals(b, rule = "upper", openupcap = 40)$value

# geometric placement, and an explicit floor for an open bottom band
bandvals(b, rule = "geometric", openupcap = 40)$value
bandvals(c("under 5", "5 to 9", "10+"), openloflr = 1)

# a cap inside the open band is refused
try(bandvals(b, openupcap = 5))
```

`benfordtst` *Benford first-digit test*

Description

Compares the distribution of leading significant digits in `x` with Benford's law, $P(d) = \log_{10}(1 + 1/d)$. Naturally occurring quantities that span several orders of magnitude follow it closely; figures that were rounded, truncated, capped, re-scaled, or invented typically do not. That makes it a cheap screen for a numeric column that arrived looking plausible but is not the measurement it claims to be.

It is a SCREEN, not a verdict. Columns with a narrow range, a unit floor or ceiling, or an assigned-identifier structure (postcodes, year fields, prices ending in 99) legitimately violate Benford's law. Treat a small p-value as a reason to look, never as evidence of fabrication.

Usage

```
benfordtst(x)
```

Arguments

`x`	Numeric vector; zero and non-finite values are dropped.

Details

Zeros and non-finite values have no leading significant digit and are excluded; the sign is ignored.

With n usable values, O_d the count with leading digit d and $E_d = n \log_{10}(1 + 1/d)$,

$$X^2 = \sum_{d=1}^{9} \frac{(O_d - E_d)^2}{E_d}$$

on 8 degrees of freedom. The smallest Benford share is 0.046 (digit 9), so the chi-square approximation needs about $n \geq 110$ for every expected count to reach 5; below that treat the p-value as rough. Digits are counted in compiled code in blocks of a million values, and a long vector shows a progress bar in an interactive session (silenced by `options(morie.quiet = TRUE)`). Errors: non-numeric input, or no finite non-zero value.

Value

A list of class `bricklayer_benford` with `counts` (observed digit frequencies 1–9), `expected`, `proportion`, `statistic`, `df`, `p_value`, `logpvalue` (the log of the upper tail, which keeps its information where `p_value` underflows to 0), and `n` (the number of values used). `counts`, `expected` and `proportion` are named numeric vectors with names `"1"` to `"9"`. The print method shows the verdict and a bar chart of observed against expected shares.

References

Benford F (1938). The law of anomalous numbers. *Proceedings of the American Philosophical Society* 78(4), 551–572.

See Also

drftchisq(), cpsdrft().

Examples

```
# A quantity spanning several orders of magnitude follows the law.
set.seed(3)
benfordtst(10^stats::runif(2000, 0, 6))

# Digits drawn uniformly do not.
benfordtst(as.numeric(paste0(sample(1:9, 2000, TRUE), "000")))

# The expected proportions are the closed form.
b <- benfordtst(10^stats::runif(500, 0, 5))
all.equal(unname(b$expected) / b$n, log10(1 + 1 / (1:9)))

# Zeros carry no leading digit and are excluded from n.
benfordtst(c(0, 0, 1, 2, 3))$n

try(benfordtst(c(0, NA)))
```

`blcli` *Run the rmoriebricklayer command line*

Description

Dispatches the verbs of the `rmoriebricklayer` launcher:

`login [-token] [-email ADDRESS [-code CODE]] [-no-browser]` sign in for the curated tables at data.rmorie.com (a free MORIE data key, issued at `www.rmorie.com/access/`): a key you paste with `--token` (read from the terminal or a pipe when KEY is omitted), a code sent to `--email`, or the GitHub device flow

`logout` forget the data key

`doctor` report the language-model routes available here (own endpoint, each provider with your key, local Ollama), in the order `ask` tries them

`models [-route ROUTE]` list the models your provider (or server) offers, the one in use marked

`data list` the curated tables at data.rmorie.com

`data pull db/table` download one table as CSV

`app [PATH] [-port N] [-no-browser]` open the analysis dashboard (rmblapp) on this machine, with PATH, a CSV file or a capsule directory, loaded; needs the shiny package

`config [setup | set KEY VALUE | unset KEY | get KEY | help]` show or change the settings of the language model (llmcfg): the route `ask` uses and the address, key and model of each route; `setup` walks through them

`ask [-route ROUTE] [-model NAME] [-fresh] PROMPT...` send a prompt to the model (or the named one, on the named route) and print the reply; the same question again is answered from this machine (`--fresh` asks anew). `aye` and `tellme` are the same verb, and a quoted question with no verb (`rmbl "how do I weight a survey?"`) is asked too

`explain-error MESSAGE...` where an error of these packages comes from (`explerr`), without a model or the network

`bundle REQUEST...` `agentbndl` from the shell

`pack DIR [-dest D] [-data FILE] [-version V] [-overwrite] [-secrets]` a reproducibility bundle of the project in DIR, or of a shipped example such as `otis-mrp`, with `mkbndl`: one zip a reviewer unzips and double-clicks (default `--dest`: the working directory; `--secrets` packs files that look like credentials, left out otherwise); `pack --new DIR [--title T] [--author A] [--contact C]` writes a starter project with `mkbndlproj`; `pack --check ZIP [--data FILE] [--synthetic] [--keep] [--container[=podman|docker]]` runs a built bundle as a recipient would and checks its results with `mkbndlchk` (exit status 1 when it fails). It RUNS THE BUNDLE'S CODE on this machine, with a minimal environment and a temporary HOME: check only bundles you trust. `--container` runs it in a podman or docker container; otherwise a PID namespace (Linux), a job object (Windows) or only a process group (macOS) contains it

`wayback FILE [-out FILE.jsonl] [-jobs N] [-within 30d] [-max N]` save every URL listed in FILE (one per line; blank lines and # comments skipped) in the Internet Archive's Wayback Machine with `wbsave`, results in `--out` (default `FILE.wayback.jsonl`), so a stopped run carries on; `wayback --retry FILE.jsonl [--include-gone]` retries the failures of an earlier run (`wbrtry`)

`archive save URL... -reason TEXT [-dir D] [-private] [-also wayback]` save from this connection to archive.rmorie.com with `arccap`, keeping the files, their manifest and WARC copies in D (default 'captures'); `archive push DIR [--reason TEXT] [--private] [--also wayback]` sends a capture kept in DIR on later, the same bytes, with `arcpush` (run it again to retry what failed); `archive find URL` and `archive get ID|URL -o FILE` list the captures of a URL and fetch one's exact bytes (`arcfind`, `arcget`)

`verify PATH [-json] [-fail-on LEVEL] [-public-key HEX]` check every capsule under PATH as the rmbl GitHub Action does (`cpschk`): each file against its JSON Schema (`schemapath`), `vrfcps` and `cpsbndlvrf`. The secret of an HMAC bundle is read from `RMBL_KEY`, which is then removed from the environment. Prints the Markdown report, or JSON with `--json`; exits 1 when a failed row is at or above `--fail-on` (`error`, the default, `drift` or `warn`). `--base MANIFEST|DIR` compares each manifest with its base copy, `--manifests GLOB` checks only the capsule files matching GLOB, `--snapshot` looks up each source's latest Wayback snapshot and `--no-footprint` leaves the footprint (`co2fp`) out of the report. `--seen-log FILE` adds the signatures an XMSS seen-log lists to those under PATH: an XMSS index one key signed twice fails its `xmss_index` row

`drift PATH [-json] [-fail-on LEVEL] [-html FILE] [-app]` verify, then fetch each source again and compare it column by column (`cpsdrft`, Benford): a column past `--psi-warn X` (0.1) is a warning, a flagged one past `--psi-fail X` (0.25) drift. `--allow-file` accepts `file://` sources, `--cache DIR` keeps each source and reads it when a fetch fails; the options of verify apply. `--html FILE` writes the interactive report (`drfthtml`) and `--app` opens it as a Shiny app (`drftapp`)

`refresh PATH` pin each source's new bytes (`cpsrfrsh`) and print what changed (`mnfdiff`)

`report PATH [-format md|json]` the verify report, exit 0; the options of verify apply

`sign PATH [-scheme S] [-key-file F] [-overwrite]` pin each provenance on its first run and sign each capsule into `capsule_bundle.json` (`cpsseal`), then verify as `verify` does. The key is `RMBL_KEY`, removed from the environment at once, or the contents of `--key-file`, and is never printed: the shared secret for `hmac` (the default), else the JSON of a `fipskg` key, or its secret hex with `--public-key`, or the JSON of a `pqckg` key for `xmss-sha256` (see the section XMSS keys). An existing bundle is refused without `--overwrite`

`functions [PATTERN]` list the exported functions with their one-line titles, optionally filtered by a regular expression

`describe NAME` title, usage, arguments and description of one function, from its help page

`examples NAME` print the examples of one function

`version` print the package version (also `--version`, `-v`)

`help [VERB]` this list, or one verb's usage (also `--help`, `-h`; `VERB --help` shows a verb's usage without running it)

The launcher installed by `instcli` calls this function with the shell's arguments and exits with its status. Calling it from R is the same as typing the command, which is how the examples below work.

Usage

```
blcli(args = commandArgs(trailingOnly = TRUE), out = cat)
```

Arguments

`args`	Character vector of arguments, verb first. The default is the command line, `commandArgs(trailingOnly = TRUE)`. An empty vector prints the help.
`out`	Function that receives the output text, one string at a time (default `cat`, the console). Pass a function that collects the strings to capture the output.

Details

An option a verb does not take is refused with exit status 2 rather than ignored or sent as part of a prompt; everything after a bare `--` is positional (for a prompt that itself starts with `--`) and only `ask`, `bundle` and `explain-error` accept it. A first argument that is plainly a question (two or more words, or ending in ?, with no path, line break or key-shaped word) and no verb is asked as `ask` would ask it.

Errors are not raised: every error is written to `out` as `"PROG: message"` and turned into the exit status. The status is 0 on success, 1 when the verb failed (an error, nothing found, no route), 2 for a usage error (an unknown verb or option, a missing value) and 130 when interrupted (Ctrl-C). The command name in messages is the environment variable `RMBL_PROG`, else `rmoriebricklayer`.

Side effects: during the call `options(morie.progress = TRUE)` is set, so downloads draw a progress bar; when `RMBL_PROG` is set (the launcher sets it) and `options(rmoriebricklayer.dcache)` is not, the data.rmorie.com manifest and tables are kept in `tools::R_user_dir("rmoriebricklayer", "cache")` rather than a fresh temporary folder. The verbs `login`, `logout` and `config` write the shared credentials and settings files; `ask`, `models`, `data`, `bundle` and `wayback` use the network, and `drift` and `refresh` fetch each capsule's source (`refresh` writes the capsule), `sign` writes

each capsule's provenance and bundle; `verify` and `report` only read the capsule files; `functions`, `describe`, `examples`, `explain-error`, `version` and `help` only read the installed help.

Value

The exit status (integer scalar), invisibly: 0 on success, 1 on failure, 2 for a usage error, 130 when interrupted.

XMSS keys

An XMSS key (pqckg) signs each of its `2^height` indices once; two signatures at one index let anyone forge a third. Its private key never changes, but its state, the indices already used, changes with every signature, and a copy of an old state signs used indices again (RFC 10033, "Hash-Based Signatures: State and Backup Management", sections 2 and 4). `rmbl sign --scheme xmss-sha256 --key-file KEY` therefore treats the key file as the state:

- Before anything is signed it rewrites KEY, under a lock, through a temporary file in the same directory that is flushed to the disk and renamed over it, with `next_index` advanced and the state in a `used` field: `highest` (the highest index used), `generation` (one more at every write), `writer` (a digest of the host name and boot id), the last `reserved` block and, once split, `sectors`. A crash after that write loses the reserved indices and never reuses one. A KEY that cannot be rewritten (not writable, a read-only file system) is refused.
- `--state-file FILE` keeps the state in FILE (an absolute path to a writable regular file, or a new file in a writable directory) and only reads KEY; a key in `RMBL_KEY` needs it. The command line never uses the configuration directory that cpssign uses in an R session; it reads and updates the record in `RMBL_XMSS_INDEX_FILE` only when that is set.
- `--sectors K --sector NAME` (K a power of two, at most half the indices) splits the indices into K disjoint ranges and signs only in one of them, which keeps its own highest index; the range of NAME is SHA-256 of NAME and the key fingerprint, modulo K. Two names land in one sector by chance, so for real use give each runner its own `--sector-index I`. A sector that is used up is refused. Once split, sign only through `rmbl sign`.
- `--reserve N` marks the next N indices used in one write (at least one per capsule signed); the capsules of the run are signed from that block, and the indices it does not use are lost, as RFC 10033, section 5.9 allows.
- A stale copy is refused: when a bundle under PATH, or the `--seen-log FILE` that every `rmbl sign` appends its signatures to, shows this key signed an index above what the state says, nothing is signed. A note says when another host wrote the state last.
- When N or fewer indices are left in the range (`--warn-left N`, by default a tenth of the range) a warning is printed in the report and on standard error (RFC 10033, section 3.4).

A CI secret is a fixed copy of the key that every job starts from, so each job would sign the same indices again: in CI sign with ML-DSA (fipskg), or keep the XMSS state in a file that persists between jobs and give each concurrent runner its own sector. `rmbl verify` shows each XMSS signature's index and key fingerprint, and xmssidxseen lists indices signed twice. NIST SP 800-208 approves XMSS only when the key is made and used inside a hardware cryptographic module that never exports it; a key file is software, so this follows the software guidance of RFC 10033 and is not a validated SP 800-208 deployment.

See Also

instcli to put the command on your PATH.

Examples

```
blcli("version")
blcli("help")

# one verb's usage
blcli(c("help", "data"))

# capture the output instead of printing it
con <- textConnection("lines", "w")
status <- blcli(c("functions", "^drbg_"), out = function(x) {
  cat(x, file = con)
})
close(con)
status
lines

# an unknown verb is a usage error: status 2, nothing run
blcli("nonsense")

# check a capsule as the rmbl GitHub Action does
d <- usecpstmpl(tempfile("capsule-"), ex = TRUE)
invisible(cpsrfrsh(d))
blcli(c("verify", d))


# these read every help page, which takes a few seconds
blcli(c("functions", "json"))
blcli(c("describe", "anltab"))
blcli(c("examples", "perdays"))
```

---

bldl *Download a file with live progress*

Description

Downloads one file from the web to a local path, showing a progress bar (percent, size and rate) while a person is watching. It is the single download routine of the package and its sibling packages, and it applies the package's network safety rules: HTTPS only, no local or private addresses, a size cap, and checked redirects. Use it when a script must fetch a source file and you want a visible, safe transfer.

Usage

```
bldl(
  url,
  dest,
  headers = NULL,
  label = basename(dest),
  size = NULL,
  tmout = 3600,
  quiet = NULL,
  tty = NULL,
  allowfile = FALSE,
  maxbytes = 2^31
)
```

Arguments

url	The URL, https. The option rmoriebricklayer.allowhttp set to TRUE also allows plain http. A loopback, link-local or private address is refused.
dest	Character string: the path to write.
headers	Character vector of request headers, every element named (name = header, value = its value), or NULL (the default).
label	Text shown in front of the bar; the file name by default.
size	Expected size in bytes when known (a percent bar instead of a spinner); otherwise the size the server announces is used.
tmout	Seconds allowed for the whole transfer: a whole number from 1 to 2147483647, default 3600. A transfer slower than 64 bytes a second for 30 seconds is ended before that.
quiet	TRUE, FALSE, or NULL (the default) to follow the session and options.
tty	Draw the live bar (TRUE) or print milestone lines (FALSE); NULL (the default) draws the bar when the console redraws a line in place, as described in Details.
allowfile	Logical, default FALSE: accept a file:// URL (for offline tests).
maxbytes	Most bytes the body may have (default 2 GiB, the transport's ceiling). A body past it, chunked or not, ends the transfer with an error and leaves nothing behind: a caller that expects a small file should say so.

Details

Progress is an animated bar, redrawn in place, when the console can redraw a line: a terminal, or the RStudio, Positron, R.app or RKWard console (never a knitr or Jupyter notebook; the options rmoriebricklayer.dynamic or cli.dynamic, or the environment variable R_CLI_DYNAMIC, override the choice). In the RStudio or Positron console the bar is drawn on stdout, elsewhere on stderr. Otherwise milestone lines go to stderr (every 10 percent, or every 50 MB when the size is unknown), and nothing is shown when quiet. By default output is shown in an interactive session or when options(morie.progress = TRUE) is set, and suppressed by options(morie.quiet = TRUE).

Network: the transfer runs in the package's compiled HTTP client. The URL is checked, its host resolved and pinned, and every redirect is checked again (no switch from https to http; credentials are not passed to another host). Plain `http://` is allowed only when `rmoriebricklayer.allowhttp`, an option, is `TRUE`. A transfer slower than 64 bytes a second for 30 seconds is abandoned. A `file://` URL is copied locally through the same progress display, only with `allowfile = TRUE`.

Errors: a URL that is not https (or allowed http/file), or that points at a loopback, link-local or private address; `headers` not a named character vector; `tmout` below 1 or above 2147483647; `maxbytes` below 1; a network failure; an HTTP status of 400 or more; a body larger than `maxbytes`. `dest` is overwritten. A server busy for a moment (HTTP 429, 500, 502, 503 or 504) is asked again up to three times, after the wait its `Retry-After` header asks for (at most a minute) or else 1, 2 and 4 seconds.

Value

`dest`, invisibly.

See Also

`blfetch` for a download with a Wayback Machine fallback.

Examples

```
# offline: a local file through the same routine
src <- tempfile(fileext = ".txt")
writeLines("hello", src)
dest <- tempfile(fileext = ".txt")
bldl(paste0("file://", src), dest, quiet = TRUE,
                    allowfile = TRUE)
readLines(dest)

# the safety checks run before any connection is made
try(bldl("http://example.org/table.csv", tempfile()))
try(bldl("https://127.0.0.1/table.csv", tempfile()))
try(bldl(paste0("file://", src), tempfile()))

# a real download needs the network


bldl("https://www.ontario.ca/some/open-data/table.csv",
                    "table.csv", maxbytes = 50e6)
```

`blfetch` *Fetch a URL to disk with an Internet Archive fallback (C++/libcurl)*

Description

The shared data-fetch foundation of the morie ecosystem. Downloads `url` to `dest`; if the live download fails (404, network), it retries the Wayback Machine snapshot – so a rotated or removed source file (CIHI, open-data portals) stays retrievable. Backed by the package's C++ kernel `rmbl_fetch_with_fallback` (libcurl), which **rmorie** and **morie** share through `LinkingTo`. The return value says which source served the file, so a report can note that it used an archived copy.

Usage

```
blfetch(url, dest, wb = "", tmout = 120L)
```

Arguments

`url`	Live source URL, a single `https` string.
`dest`	Destination file path, a single string; its directory must exist.
`wb`	Optional explicit Wayback snapshot URL, a string. `""` (default) or `NULL` auto-resolves one via `wbsnpurl`'s API when the live download fails.
`tmout`	Per-request timeout in seconds, a positive whole number (default 120).

Details

Before any network use the arguments are checked: `url` must be one `https` URL to a public host (plain `http` only when the option `rmoriebricklayer.allowhttp` is `TRUE`; loopback, private, link-local and metadata addresses are refused), the directory of `dest` must exist and be writable, and `tmout` must be at least one second. The compiled layer then resolves the host, pins the connection to the checked addresses and re-checks every redirect.

If the live download fails, the fallback is the `wb` URL when given, else a snapshot looked up through the Wayback availability API (as `wbsnpnat()` does). `dest` is overwritten. A spinner is drawn on stderr while waiting, unless `options(morie.quiet = TRUE)`.

Errors: invalid arguments as above; and, when both sources fail, an error saying whether the live URL could not be reached or answered an HTTP status, and whether the Wayback Machine had no snapshot or its copy could not be downloaded.

Value

Invisibly, `"live"` or `"wayback"`: the source that served the file. Throws on total failure.

Examples

```
# Inputs are validated before any network access:
try(blfetch("", tempfile()))          # empty url -> error
try(blfetch("ftp://example.org/data.csv", tempfile()))
try(blfetch("https://127.0.0.1/data.csv", tempfile()))
try(blfetch("https://example.org/data.csv",
                    file.path(tempdir(), "no-such-dir", "data.csv")))


# needs the network
```

```
dst <- tempfile(fileext = ".xlsx")

# Live download; auto-resolves a Wayback snapshot only if the live URL fails.
blfetch(
  paste0(
    "https://www.cihi.ca/sites/default/files/document/",
    "hospital-beds-2024-2025-data-tables-en.xlsx"
  ),
  dst
)

# Pin an explicit Wayback snapshot to fall back to, and a shorter timeout.
blfetch(
  "https://example.org/rotated-file.csv",
  tempfile(fileext = ".csv"),
  wb = paste0(
    "https://web.archive.org/web/2024id_/https://example.org/rotated-fil",
    "e.csv"
  ),
  tmout = 60
)

# The return value tells you which source served the file.
status <- blfetch("https://cloud.r-project.org/", tempfile())
status # "live" or "wayback"
```

`blpal` *Plot styling for the package's plot methods*

Description

Every `plot()` method in the package draws with base graphics and the same small set of style choices, which these functions expose. `blpal()` returns colour-blind-safe palettes: `"okabe"` (the Okabe-Ito categorical set), `"diverging"` (blue, grey, orange), `"sequential"` (light to dark blue) and `"grey"`. `plotopt()` sets the defaults every method uses, for the session: `palette`, `grid`, `cex`, `las` and `bg`. Each method also takes `main`, `col` (which overrides the palette), `palette` and `...` passed to the underlying base-graphics call, so a single plot can be changed without changing the session. Use `blpal()` also to colour your own plots so they match the package's.

Usage

```
blpal(name = c("okabe", "diverging", "sequential", "grey"), n = NULL)

plotopt(...)
```

Arguments

`name`	Character scalar, the palette: `"okabe"` (the default), or one of `"diverging"`, `"sequential"` and `"grey"`.

n	Integer scalar or NULL (the default: the palette's own colours). The number of colours wanted; "okabe" and "grey" recycle, "sequential" and "diverging" interpolate.
...	For plotopt(), named options to set: palette (a palette name), grid (logical, draw grid lines), cex (numeric text and symbol size), las (axis label orientation, as in par()), bg (background colour) and panel (the plot region's fill behind white grid lines, NA for none). With no arguments, the current values are returned.

Details

The palettes: "okabe" is the eight-colour set of Okabe and Ito (2008), distinguishable under the common forms of colour-vision deficiency. "diverging" has three anchors (blue #2166AC, grey #8C8C8C, orange-red #D6604D) and is used for changes (decrease, no change, increase). The palette "sequential" has five blues from light to dark; "grey" five greys from dark to light. With n given, "okabe" and "grey" repeat their colours (rep_len()) and "diverging" and "sequential" are interpolated with grDevices::colorRampPalette().

The session options are stored in options(rmoriebricklayer.plot); the defaults are palette = "okabe", grid = TRUE, cex = 1, las = 1, bg = "white" and panel = "#EEF1F6". Setting some options keeps the others.

Text is drawn at the session cex. Titles, axis titles, row labels and keys are made smaller to fit a small device, but never below 0.6 of that size, so that they stay readable; for larger text set, for example, plotopt(cex = 1.2), and for a table with many rows use a taller device.

Errors: an unknown palette name; n not a positive whole number; an option name other than the six above, or an unnamed value; an unknown palette option.

Value

blpal(): a character vector of hexadecimal colours. plotopt(): a named list of the six options in force (after the change), invisibly when setting.

References

Okabe, M. and Ito, K. (2008). Color Universal Design (CUD): how to make figures and presentations that are friendly to colorblind people. jfly.uni-koeln.de/color

Examples

```
blpal("okabe", 3)
# plotopt() returns the options in force, so keep them before a change
old <- plotopt()
plotopt(palette = "grey", grid = FALSE)
plotopt(palette = old$palette, grid = old$grid)

# an interpolated sequential scale, shown as swatches
cols <- blpal("sequential", 7)
barplot(
  rep(1, 7),
  col = cols,
```

```
  border = NA,
  axes = FALSE,
  names.arg = cols,
  las = 2,
  cex.names = 0.6
)

# the same palette in a base plot of your own
plot(1:8, pch = 19, cex = 3, col = blpal("okabe"))

# names are checked
try(blpal("viridis"))
try(plotopt(colour = "red"))
```

`blprog` *Live progress for long-running work*

Description

Makes a progress bar for a loop or a long computation: the animated bar of `bldl()`, redrawn in place at no more than 12 frames a second, when the console can redraw a line – a terminal, or the RStudio, Positron, R.app or RKWard console (never a knitr or Jupyter notebook; the options `rmoriebricklayer.dynamic` or `cli.dynamic`, or the environment variable `R_CLI_DYNAMIC`, override the choice). In the RStudio or Positron console the bar is drawn on stdout, elsewhere on stderr. Otherwise a milestone line goes to stderr at each tenth, and nothing is shown when quiet. With `total = 0` the work has no count, and the bar is a spinner with the time so far and the current stage (the `info` text), with a line at each new stage when the console cannot redraw.

Usage

```
blprog(label, total, quiet = NULL, color = "blue", delay = NULL)
```

Arguments

`label`	A short name for the work, shown before the bar.
`total`	The number of steps, or 0 when the work has no count.
`quiet`	`TRUE`, `FALSE`, or `NULL` to follow the session (see above).
`color`	The bar's colour: `"blue"`, `"green"`, `"red"`, `"violet"` or `"amber"`.
`delay`	Seconds of work before anything is drawn; `NULL` takes the package's 0.75.

Details

Progress is shown when a person is watching: an interactive session, or a command line that sets `options(morie.progress = TRUE)`. `options(morie.quiet = TRUE)` or `quiet = TRUE` silences it.

Nothing is drawn until the work has run for `delay` seconds, so a fast call never flashes a bar. A bar made inside a function while a bar made by a calling function is still running stays silent, so nested work shows one bar, for the outer loop.

Updates are cheap: when quiet an update only stores the value, and when drawing, frames are throttled, so `$tick()` can be called on every iteration of a loop. Call `$done()` through `on.exit(p$done(), add = TRUE)` right after making the bar, so the line is cleared and the bar is released even when the work fails or is interrupted:

```
p <- blprog("bootstrap", B)
on.exit(p$done(), add = TRUE)
for (b in seq_len(B)) {
  ...
  p$tick(info = sprintf("resample
}
```

`$done()` may be called more than once; only the first call acts.

Value

An object of class `rmbl_progress`, a list of functions: `update(done, info = NULL)` sets the steps done (and the text after the bar), `tick(by = 1, info = NULL)` adds by steps, `wait(seconds)` sleeps while keeping the animation moving, and `done(summary = NULL)` ends the bar, clears the line and, when the bar was shown, writes `summary` on its own line. Each returns the object invisibly.

Examples

```
p <- blprog("squares", 50, delay = 0)
for (i in 1:50) {
  x <- sum(seq_len(i)^2)
  p$tick(info = sprintf("step %d/50", i))
}
p$done("squares: 50 steps")

# work with no count: a spinner and the stage
f <- function() {
  p <- blprog("model", 0, delay = 0)
  on.exit(p$done(), add = TRUE)
  p$update(info = "fitting")
  p$wait(0.2)
  p$update(info = "writing")
  invisible(TRUE)
}
f()
```

---

bufasgn *Treated, comparison and excluded places around sites*

Description

Assigns each place (a block, an address) to the treated group when it lies within a radius of a site (a store that donated, a new facility), to the comparison group when it lies within the same radius of a comparison point and farther than a buffer from every site, and excludes it otherwise, so that spillover from the sites does not reach the comparison.

Usage

```
bufasgn(x, y, sitex, sitey, comx, comy, radius, buffer = 3 * radius)
```

Arguments

x, y	Coordinates of the places.
sitex, sitey	Coordinates of the sites.
comx, comy	Coordinates of the comparison points (for example the nearest major intersections).
radius	The treatment radius.
buffer	The distance from every site a comparison place must keep (default 3 * radius).

Details

Distances are Euclidean, so coordinates must be projected (metres or feet, not degrees). The radius and the buffer are in the same unit. The buffer should be several times the radius: the design it follows used 330 feet and 990 feet.

Value

A data frame of class rmbl_buffer with the columns group ("treated", "comparison" or "excluded"), site_distance and comparison_distance.

References

Applied to blocks near donor stores: Zorro Medina, Hackett, Green and Vargas (2025), see evstudy.

Examples

```
set.seed(4)
x <- runif(200, 0, 5000)
y <- runif(200, 0, 5000)
a <- bufasgn(x, y, c(1000, 3500), c(1200, 3000), c(2500, 4500), c(500, 4000),
                   radius = 330)
table(a$group)
```

---

capdeps *Where each loaded package came from*

Description

Records, for every package named, its version and enough about its provenance to find the same one again: the library it was loaded from, the repository it was installed from, and – for a package installed from a remote – the remote's URL and commit. Store it in a manifest next to capenv() when an analysis depends on packages installed from GitHub or a local build.

Usage

```
capdeps(pkgs = loadedNamespaces())
```

Arguments

pkgs	Character vector of package names. Defaults to the namespaces currently loaded.

Details

Why versions alone are not enough. Two installations can report the same version and differ: one built from CRAN, one from a fork, one from a local R CMD INSTALL of a working tree with uncommitted changes. A version number identifies an intention; the repository and commit identify what was actually loaded.

The fields are read with utils::packageDescription() from each package's installed DESCRIPTION: Repository (for example "CRAN"), RemoteUrl and RemoteSha (written by remotes, pak and devtools for packages installed from a Git host) and Built. Packages are sorted and de-duplicated. A progress bar is drawn on stderr for many packages unless options(morie.quiet = TRUE). Nothing is written. Error: pkgs that is not a character vector.

Value

A data frame with one row per package and character columns package, version, library (the library directory), repository, remote_url, remote_sha and built (R version, platform and time of the build). Unavailable fields are NA rather than omitted, so a reader can see that the information was absent rather than forgotten; for a package that is not installed, every field except package is NA, and a warning names it. An empty pkgs gives a data frame with no rows and the same columns.

See Also

capenv(), envdiff().

Examples

```
deps <- capdeps(c("stats", "utils"))
deps[, c("package", "version")]

# a package with no repository field records that fact
is.na(capdeps("stats")$repository)

# a package that is not installed is a row of NAs, not an error
capdeps("notapackage123")[, c("package", "version")]

# names must be character
try(capdeps(1))
```

capenv *Capture the Analysis Environment for a Manifest*

Description

Records the facts a replicator needs to rebuild the session in which a result was produced: the R version, platform and operating system, a UTC timestamp, the random number generator in use and whether it was seeded, and the versions of the packages involved. Store the result in a results manifest (for example with `wrmnfjs()`) next to the numbers it produced, and compare two captures with `envdiff()`.

Usage

```
capenv(pkgs = loadedNamespaces())
```

Arguments

`pkgs`	Character vector of package names to record. Defaults to every currently loaded namespace (`loadedNamespaces()`).

Details

Package versions come from `utils::packageVersion()`; a package that is not installed gets `NA` rather than an error. The packages are sorted and de-duplicated. The generator kinds come from `RNGkind()` (R has changed its default `sample()` algorithm before, so a recorded seed means nothing without the kind it was fed to), and `rng_seeded` says whether `.Random.seed` exists in the global environment. Nothing is written and the random state is not touched.

Value

A list of seven elements:

`r_version` character, e.g. `"4.5.1"`.

`platform` character, `R.version$platform`.

`os` character, the system name and release.

`captured_utc` character, ISO 8601 time `YYYY-MM-DDThh:mm:ssZ`.

`rng_kind` character, the three `RNGkind()` values joined by commas.

`rng_seeded` logical, whether `.Random.seed` exists.

`packages` named list of version strings, sorted by name (`NA` for a package that is not installed). A list, not a vector, so the names survive the manifest's JSON: a named vector is written as a bare array.

The list has class `c("rmbl_capenv", "list")`, which only changes how it prints (a box); its elements, and the JSON written from it, are those of a plain list.

Examples

```
# Record specific packages' versions alongside the session facts.
env <- capenv(c("stats", "utils"))
env$r_version
env$os
env$packages          # named list of versions

# Default captures every currently loaded namespace.
names(capenv())

# the generator and whether it was seeded
set.seed(42)
capenv("stats")[c("rng_kind", "rng_seeded")]

# a package that is not installed is recorded as NA, not an error
capenv(c("stats", "notapackage123"))$packages
```

`caseflow` *Case flow through a court: clearance, backlog and time to clear*

Description

Turns the counts a court publishes each period – cases received, cases disposed and, when it is published, cases pending – into the measures used to judge whether a court is keeping up: the clearance rate (cases disposed per case received), its rolling value over a window, the net flow into the inventory and, with the pending count, the number of periods the current inventory would take to clear at the recent pace of disposition.

Usage

```
caseflow(recv, disp, pend = NULL, prd = NULL, win = 12L, conflvl = 0.95)
```

Arguments

`recv`	Cases received (new cases), one per period, in time order.
`disp`	Cases disposed, one per period.
`pend`	Optional cases pending at the end of each period.
`prd`	Optional period labels: numbers, dates, or `"YYYY-MM"` strings (which the plot draws on a date axis).
`win`	Periods in the rolling clearance rate and in the pace behind `pertoclear`. Default 12, a year of months.
`conflvl`	Confidence level for the intervals. Default 0.95.

Details

A clearance rate above 1 means the court disposed of more cases than it received, so its inventory fell. The interval treats received and disposed as Poisson counts over the same period: given their sum, the disposed count is binomial, and the exact (Clopper-Pearson) limits for that proportion give exact limits for the ratio. The interval measures the period-to-period noise of counts this size; it does not make received and disposed the same cases, which they are not (a case disposed this month was usually received months before).

`pertoclear` divides the pending inventory by the mean number disposed over the last `win` periods (fewer at the start of the series). The published pending count need not equal the previous count plus the net flow – courts reactivate cases on bench warrants and transfer them – so `penchg` is reported next to `net` rather than derived from it.

The result carries attribute `"caseflow"`: `win`, `conflvl` and the totals received and disposed. Errors: `recv` empty; a count vector that is not numeric, has a missing, negative or infinite value, or a length other than `recv`'s; `prd` of another length, with missing or repeated values; `win` not a whole number of at least 1; `conflvl` outside (0, 1).

Value

A data frame of class `rmbl_case_flow`, one row per period, with `period`, `received`, `disposed`, `clearance` with `lower` and `upper`, `net` (received less disposed: positive grows the inventory), `rolclr` with `rollo` and `rolup` (`NA` until a full window is available), and either `pending`, `penchg` and `pertoclear`, or, without `pending`, `cumnet` (the inventory change since the first period).

References

Ostrom, B. J. and Hanson, R. A. (1999). *Efficiency, Timeliness, and Quality: A New Perspective from Nine State Criminal Trial Courts*. National Center for State Courts, Williamsburg, VA. (Clearance rate and time-to-disposition measures.)

Ulm, K. (1990). A simple method to calculate the confidence interval of a standardized mortality ratio. *American Journal of Epidemiology*, 131(2), 373-375. (Exact limits for a ratio of Poisson counts through the beta distribution.)

See Also

`lifetbl`, `dispmix`, `stkflow` for custody populations.

Examples

```
f <- read.csv(system.file(
  "extdata",
  "ocj_flow_ontario.csv",
  package = "rmoriebricklayer"
))
cf <- caseflow(f$received, f$disposed, pend = f$pending, prd = f$month)
cf
plot(cf)

# without the pending count, a quarterly rolling window and 90% intervals
cq <- caseflow(
```

```
  f$received,
  f$disposed,
  prd = f$month,
  win = 3,
  conflvl = 0.90
)
head(as.data.frame(cq)[, c("period", "clearance", "rolclr", "cumnet")])

# one value per period in every count
try(caseflow(c(100, 120, 90), c(95, 130)))
```

`cblake2b` *BLAKE2b digest, optionally keyed*

Description

Computes the BLAKE2b hash of text or bytes as lowercase hex, with any digest length from 1 to 64 bytes. With a key it becomes a message authentication code: only someone holding the key can produce or check the digest. Use it for fingerprints and keyed checks inside your own workflow; use `csha256()` or `chmac()` where other tools must reproduce the value.

Usage

```
cblake2b(x, key = NULL, length = 32L)
```

Arguments

`x`	A character vector or a raw vector.
`key`	Optional key: a single string or a raw vector of at most 64 bytes. `NULL` (default) gives the plain digest.
`length`	Digest length in bytes, 1 to 64 (default 32, matching SHA-256's width).

Details

BLAKE2b (RFC 7693): a modern cryptographic hash, faster than SHA-256 in software, that takes a key natively and produces any digest length from 1 to 64 bytes.

The native key is the interesting part. A keyed BLAKE2b IS a message authentication code, with no HMAC wrapper and so no doubled hashing – `cblake2b(msg, key = k)` does the job of `chmac()` at lower cost. Use SHA-256 or HMAC where interoperability with other tools matters; use this where it does not and speed does.

Character input is hashed element by element, as the bytes of each string; `NA` elements give `NA`. A raw vector is one message. The digest length is a parameter of BLAKE2b, so a 32-byte digest is not the first half of the 64-byte digest of the same input. Errors: `length` outside 1 to 64; a key that is not a single string or raw vector, or is longer than 64 bytes.

Value

A character vector of lowercase hex digests – one per element for character input, length-1 for raw input.

References

Saarinen MJ, Aumasson JP (2015). The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693. DOI 10.17487/RFC7693

See Also

csha256(), csha512(), chmac()

Examples

```
# The RFC 7693 test vector for BLAKE2b-512 of "abc".
cblake2b("abc", length = 64)

# 32 bytes by default, the same width as SHA-256.
cblake2b("abc")
nchar(cblake2b("abc"))

# Any digest length, which SHA-2 cannot do.
cblake2b("abc", length = 8)

# Keyed, so it authenticates without an HMAC construction.
cblake2b("manifest", key = "secret")
cblake2b("manifest", key = "secret") ==
  cblake2b("manifest", key = "other")

# Vectorised, and raw input hashes the same bytes.
cblake2b(c("a", "b"))
identical(cblake2b("abc"), cblake2b(charToRaw("abc")))

try(cblake2b("abc", length = 65))
```

---

cbootmean *Bootstrap replicate means (C backend)*

Description

B resamples of x, drawn with replacement and each the same length as x, with the mean of every resample returned. The spread of these replicate means estimates the standard error of the sample mean, and their quantiles give a percentile confidence interval, without assuming the data are normal (Efron and Tibshirani, 1993). The resampling uses the core's own 64-bit Mersenne Twister seeded by seed, NOT R's RNG, so a given seed reproduces the same replicates in every binding of the core and R's own random stream is left untouched.

Usage

```
cbootmean(x, B = 1000L, seed = 42L)
```

Arguments

x	Numeric (or logical) vector to resample.
B	Number of bootstrap replicates, a single whole number of at least 1 (default 1000).
seed	Seed for the core's generator, a single non-negative whole number (default 42). It does not use or change `set.seed()`'s stream.

Details

Replicate b draws n indices uniformly from $1, \ldots, n$ with replacement (C++ `std::mt19937_64` seeded with `seed`, then `std::uniform_int_distribution`, whose algorithm is the C++ library's own, so the replicates for a seed can differ between libstdc++ and libc++) and returns the mean of the drawn values, accumulated in extended precision. The replicates are a single stream: the first `B` replicates for a seed are the same whatever larger `B` is asked for later.

Edge cases: an empty `x` gives `B` values of `NaN`; an `NA` in `x` makes every replicate that draws it `NA`, so remove missing values first. `B` must be a whole number of at least 1 (`"B must be a single whole number >= 1"`); a character, factor or list `x` is refused (`"must be numeric"`). `seed` must be a whole number from 0 to below 2^{53}: `NA`, a negative or a fractional seed is refused.

Performance: the work is $B \times n$ draws (about 0.4 s for $B = 2000$, $n = 20000$ on a laptop). The kernel polls for a Ctrl-C after about every million draws, so a long run can be interrupted; while it runs, an interactive session (or one with `options(morie.progress = TRUE)`) shows a spinner on stderr from 0.75 s on, and `options(morie.quiet = TRUE)` silences it.

Value

A numeric vector of length `B`: the replicate means, in the units of `x`.

References

Efron B, Tibshirani RJ (1993). *An Introduction to the Bootstrap*. Chapman and Hall.

See Also

`exprspplr()` for a bootstrap of a model estimate.

Examples

```
set.seed(1)
x <- stats::rnorm(50, mean = 5)

reps <- cbootmean(x, B = 500, seed = 7)
length(reps)

# The replicates centre on the sample mean, and their spread estimates
# the standard error.
```

```
c(sample = mean(x), bootstrap = mean(reps))
c(bootstrapse = stats::sd(reps),
  formulase = stats::sd(x) / sqrt(length(x)))

# A percentile confidence interval for the mean.
stats::quantile(reps, c(0.025, 0.975))

# Reproducible: the same seed gives the same replicates, and R's own
# random stream is not consumed.
identical(cbootmean(x, 100, seed = 1),
          cbootmean(x, 100, seed = 1))

# A longer run continues the same stream.
identical(cbootmean(x, 10, seed = 1),
          cbootmean(x, 100, seed = 1)[1:10])

# At least one replicate is needed.
try(cbootmean(x, B = 0))
```

---

ccovm *Covariance matrix of a numeric matrix (C backend)*

Description

The covariance of every pair of columns of a numeric matrix or data frame, computed by the compiled core. Covariances describe how variables move together; the matrix is the input to principal components, Mahalanobis distances and multivariate drift checks. Column names are carried through to both dimensions of the result.

Usage

```
ccovm(x)
```

Arguments

x	A numeric matrix, or a data frame whose columns are all numeric (rows = observations, columns = variables).

Details

Entry (j, k) is $\frac{1}{n-1}\sum_i (x_{ij} - \bar{x}_j)(x_{ik} - \bar{x}_k)$, the n - 1 denominator of stats::cov(), with which it agrees to rounding. The diagonal holds the column variances.

Missing values are not removed: an NA in a column makes every covariance involving that column NA (pairwise or complete-case deletion is not done; use stats::cov() with use = for those). A single row gives NaN throughout. Refusals: a vector that is not a matrix ("must be a matrix or data frame") and a data frame with a non-numeric column ("every column of x must be numeric"). Logical and integer matrices are converted to double. The cost is $O(np^2)$ for n rows and p columns.

Value

A symmetric `ncol(x)` by `ncol(x)` numeric matrix, with the column names of `x` (if any) as both row and column names; entry units are the product of the two columns' units.

See Also

`ccorr()` for a single Pearson correlation, `cmom()` for one column's variance.

Examples

```
X <- cbind(a = c(1, 2, 3, 4), b = c(2, 4, 7, 8), c = c(5, 3, 2, 1))
ccovm(X)

# Agrees with stats::cov().
all.equal(ccovm(X), stats::cov(X))

# The diagonal is the column variances.
all.equal(diag(ccovm(X)), apply(X, 2, stats::var))

# Data frames are accepted.
ccovm(data.frame(u = 1:5, v = c(2, 1, 4, 3, 6)))

# A plain vector is not a matrix of variables.
try(ccovm(1:5))
```

---

ccrc32 *CRC-32 checksum (C backend)*

Description

Computes the CRC-32 checksum of character or raw input: the CRC of ITU V.42, zip and PNG (reflected polynomial `0xEDB88320`, initial value and final XOR `0xFFFFFFFF`). Use it as a fast first check that a large file or capsule member arrived intact, or to compare with the CRC a zip archive records for its members.

Usage

```
ccrc32(x)
```

Arguments

`x`	A character vector (one checksum per element) or a raw vector (one checksum of all its bytes).

Details

A CRC is NOT a cryptographic digest: it detects accidental corruption – a truncated download, a flipped bit on disk – but anyone can construct a different input with the same value, so it must never be used where csha256() is meant. It is here because it is far cheaper than SHA-256 over gigabyte-scale capsule members, which makes it the right first pass when the question is only "did this file arrive intact".

Character input is checked element by element, as its bytes, with NA giving NA; other atomic input is converted with as.character(). A raw vector gives one checksum. For files use crc32file().

Value

A numeric vector of unsigned 32-bit checksums (one per element for character input, NA for NA; length 1 for raw input). Returned as double rather than integer because values above 2^31 - 1 are not representable as an R integer.

See Also

csha256() for integrity against tampering rather than accident.

Examples

```
# The standard check value: CRC-32("123456789") is 0xCBF43926.
ccrc32("123456789")
ccrc32("123456789") == 0xCBF43926

# The empty input has checksum zero.
ccrc32("")

# Vectorised, and sensitive to a single changed byte.
ccrc32(c("brick", "brack"))

# Raw bytes work the same way.
ccrc32(charToRaw("123456789"))
```

---

certchnvrf *Verify a certificate chain*

Description

Builds the path from a leaf certificate to a trust anchor you supply, verifying each signature, each validity window, and the constraints that stop a certificate being used for something it was not issued for.

Usage

```
certchnvrf(
  leaf,
  trust,
  inter = list(),
  attm = Sys.time(),
  purp = NULL,
  crls = list(),
  pols = NULL,
  rvk = c("supplied", "fetch", "none")
)
```

Arguments

`leaf`	The certificate to validate, from `certprs()` or anything `certprs()` accepts.
`trust`	A list of trust anchors – the certificates you have decided to believe. A chain that does not reach one of these fails.
`inter`	Optional further certificates to build the path through. They are not trusted by being supplied; they still have to verify.
`attm`	The time to check validity windows at: a `POSIXct` or anything `as.POSIXct()` reads, taken as UTC. Defaults to now.
`purp`	Optional extended key usage the leaf must carry, as a name (`"timeStamping"`, `"codeSigning"`, ...) or an OID. Default `NULL`: no purpose is required.
`crls`	Optional list of CRLs, as raw DER or paths, to check the chain against. Default `list()`.
`pols`	Optional character vector of acceptable certificate policy OIDs; default `NULL`. Supplied, the path must yield at least one of them after policy mapping and the constraints in RFC 5280 section 6.1; omitted, policies are processed but only reported as a failure when a certificate in the path requires an explicit policy.
`rvk`	`"supplied"` (the default) checks only the CRLs given in `crls`. `"fetch"` additionally retrieves CRLs from the distribution points in the certificates and queries OCSP responders – which makes verification depend on the network and discloses to the responder which certificates are being checked, so it is never the default. `"none"` skips revocation entirely.

Details

`attm` is the point the validity windows are checked against, and it matters which one you pass. For a timestamp token the right answer is the time the token asserts: a token signed in 2020 by a certificate that expired in 2021 was validly signed, and checking it against today would reject it for no good reason. For a signature you are being asked to rely on now, pass now.

The path is built by following issuer names from the leaf through the intermediates until a certificate in `trust` is reached, at most ten steps (a cycle among supplied certificates cannot loop). Then every link is checked: the validity window at `attm`, the signature under the issuer's key (RSA PKCS#1 v1.5 with SHA-1, SHA-256, SHA-384 or SHA-512; ECDSA with SHA-256, SHA-384 or SHA-512 on P-256, P-384 and P-521), that every issuer is a CA with a path length that allows the chain,

name constraints, the policy rules of RFC 5280 section 6.1, the requested `purp` (extended key usage of the leaf), that the anchor is self-signed, and revocation as `rvk` says. Each check is one row of `checks`; `ok` is `TRUE` only when every row passed. A failed check is a row, not an error.

Errors: `trust` empty; a certificate that `certprs()` cannot read; an unknown `rvk`. With `revocation = "fetch"` the network is used (distribution points and OCSP responders named in the certificates); otherwise nothing is fetched.

Value

A list of class `bricklayer_certpath_check`: `ok` (logical, the verdict), `path` (the list of certificates from the leaf to the anchor, as built), `checks` (a data frame with `check`, logical `ok` and `detail`, one row per check), `attm` (the time used) and `policies` (the outcome of policy processing). `print()` shows the verdict and the checks.

See Also

`certprs()`, `tsvrf()`.

Examples

```
# The test chain shipped with the package: a self-signed P-256 root and a
# leaf it issued (a throwaway test authority; trust it for nothing). With
# your own certificates, pass their files (or certprs() results).
x509 <- function(f) {
  system.file("extdata", "x509", f, package = "rmoriebricklayer")
}
root <- certprs(x509("test-root.der"))
leaf <- certprs(x509("test-leaf.der"))

# the leaf chains to the root, checked a day after it was issued
res <- certchnvrf(leaf, trust = root, attm = leaf$not_before + 86400)
res$ok
res$checks[, c("check", "ok")]

# after the leaf expired, the same chain fails on validity
certchnvrf(leaf, trust = root, attm = leaf$not_after + 86400)$ok

# a purpose the leaf was not issued for
r <- certchnvrf(
  leaf,
  trust = root,
  attm = leaf$not_before + 86400,
  purp = "timeStamping"
)
r$checks[!r$checks$ok, ]

# a chain must end at something you trust
try(certchnvrf(leaf, trust = list()))


# with revocation looked up on the network
```

```
certchnvrf("tsa.crt", trust = "ca.crt", purp = "timeStamping",
                  rvk = "fetch")
```

`certprs` *Parse an X.509 certificate*

Description

Reads the fields a verifier needs: who issued it, who it is for, when it is valid, what key it carries, and what that key is allowed to do. Use it to inspect the certificate of a timestamping authority or a signer before you decide to trust it, or as the first step of `certchnvrf()`. The DER decoder is the package's own; no system library or network is used.

Usage

```
certprs(cert)
```

Arguments

`cert`	The certificate, DER or PEM: a raw vector of DER bytes, or the path of a `.der`, `.crt`, `.cer` or `.pem` file.

Details

The certificate is decoded as DER (RFC 5280 section 4.1); a file whose text contains a `-----BEGIN` line is read as PEM and its base64 body decoded first (only the first certificate in the file is used). Version 1 certificates, without the explicit version field, are accepted. Extensions read: basic constraints, key usage, extended key usage, subject alternative name, name constraints, certificate policies, policy mappings and policy constraints, and inhibit any-policy. The signature is not checked here: that needs the issuer's key, which `certchnvrf()` supplies.

Errors: `cert` neither a raw vector nor the path of an existing file (a directory is refused); bytes that are not well-formed DER or carry trailing bytes; a DER structure that is not a certificate or has no signature.

Value

A list of class `bricklayer_certificate` with these elements:

- `version` (integer) and `serial` (hex);
- `issuer` and `subject` (distinguished names as text);
- `not_before` and `not_after` (`POSIXct`, UTC);
- `signature_algorithm` (a name such as `"ECDSA-SHA256"`) and `signature_oid`;
- `signature` and `tbs` (raw; the signed part);
- `key` (a list: `type` `"RSA"` or `"EC"`, `bits`, and the key values, with `curve` for EC);
- `is_ca` and `path_len`;

- key_usage and extended_key_usage (character; EKU as OIDs);
- subject_rdns, issuer_rdns, san and name_constraints;
- the policy fields policies, policy_mappings and require_explicit_policy;
- the policy fields inhibit_policy_mapping and inhibit_any_policy;
- self_signed (logical: issuer equals subject);
- der (the bytes).

print() shows a summary.

See Also

certchnvrf(), tsvrf().

Examples

```
# Pass the certificate file you were given (.der, .crt, .cer or .pem), for
# example the one a timestamping authority publishes, or raw DER bytes.
# Here: a small test chain shipped with the package, a self-signed P-256
# root and a leaf it issued (a throwaway test authority; trust it for
# nothing).
x509 <- function(f) {
  system.file("extdata", "x509", f, package = "rmoriebricklayer")
}
root <- certprs(x509("test-root.der"))
root
leaf <- certprs(x509("test-leaf.der"))
leaf$subject
leaf$key$curve
c(leaf$not_before, leaf$not_after)

# the same root as PEM, the text form most authorities hand out
certprs(x509("test-root.pem"))$self_signed

# raw DER bytes work too
f <- x509("test-leaf.der")
der <- readBin(f, "raw", file.size(f))
identical(certprs(der)$serial, leaf$serial)

# bytes that are not a certificate
try(certprs(charToRaw("not a certificate")))
```

---

cgamcdf *Regularized incomplete gamma function (C backend)*

Description

The lower regularized incomplete gamma function $P(a, x)$, which is the CDF of a Gamma distribution with shape `a` and unit rate. Exposed because it is the building block of the chi-square tail used by `drftchisq()` and `benfordtst()`: the p-value of a chi-square statistic q on k degrees of freedom is $1 - P(k/2, q/2)$.

Usage

```
cgamcdf(shape, x)
```

Arguments

`shape`	Shape parameter a, a single positive number.
`x`	Numeric vector of quantiles at which the gamma CDF is evaluated.

Details

$$P(a, x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t}\, dt,$$

evaluated by the shared core (a series for small x, a continued fraction otherwise, the standard pairing), and agreeing with `stats::pgamma(x, shape)` to rounding. For a gamma distribution with rate r, pass `x * r`.

`shape` must be a single positive number (`"shape must be a single positive number"`). A negative `x` gives 0, a missing `x` gives `NA`, and `Inf` gives 1, as in `stats::pgamma()`.

Value

A numeric vector the length of `x`, each entry a probability in $[0, 1]$.

See Also

`stats::pgamma()`, the base R equivalent.

Examples

```
cgamcdf(3.5, c(0.5, 1, 4, 12))

# Identical to the unit-rate gamma CDF.
all.equal(cgamcdf(3.5, c(0.5, 1, 4, 12)),
          stats::pgamma(c(0.5, 1, 4, 12), shape = 3.5))

# Shape 1 is the exponential distribution.
all.equal(cgamcdf(1, c(0.5, 2)), stats::pexp(c(0.5, 2)))

# A chi-square tail on k degrees of freedom is 1 - P(k/2, q/2).
1 - cgamcdf(2 / 2, 5.99 / 2)      # about 0.05 on 2 df

# The shape must be positive.
try(cgamcdf(0, 1))
```

chgenv *Envelope of a difference, a percent change or a rate under publication bounds*

Description

Given the intervals from pubbnd() for the current and the previous cell, the set of differences and percent changes that the observed counts could have produced is an interval too: its ends are reached at the ends of the input intervals, because the difference and the ratio are monotone in each argument. This is interval arithmetic, not a confidence interval; it says nothing about sampling and everything about what the release withheld.

Usage

```
chgenv(now, prev, pop = NULL, prepop = NULL, per = 1000)
```

Arguments

now, prev	rmbl_bounds objects from pubbnd() (or data frames with numeric lower and upper) for the current and the earlier cell, with the same number of rows.
pop, prepop	Optional numeric exposures (for example residents) for a rate envelope of the current and the previous cell; the rate is per * count / population. Default NULL. prepop is used only with pop.
per	Numeric scalar, the rate base when exposures are given; default 1000 (per 1,000).

Details

With current interval $[a_l, a_u]$ and previous interval $[b_l, b_u]$: the change lies in $[a_l - b_u, a_u - b_l]$ and the percent change in $[100(a_l/b_u - 1), 100(a_u/b_l - 1)]$. An end of the percent interval is NA when its denominator could be zero (an upper bound needs $b_l > 0$, a lower bound $b_u > 0$), since a change from zero has no percentage. A rate interval is $[per \cdot a_l/pop, per \cdot a_u/pop]$; the population is taken as exact.

Errors when now and prev have different numbers of rows. The populations are not checked; give one per row (or one value).

Value

A data frame, one row per cell, with numeric columns chglo, chgup, pctlo, pctup (NA where the previous cell could have been zero); with population also ratelo and rateup; with prepop as well preratelo and prerateup.

See Also

pubbnd(), yoybnd().

Examples

```
now <- pubbnd(c(45, 120), rnd = 5)
prev <- pubbnd(c(40, 100), rnd = 5)
chgenv(now, prev)

# with rates per 1,000 residents in both years
chgenv(now, prev, pop = c(10000, 20000),
               prepop = c(10000, 19000))

# a suppressed previous cell could have been zero: no upper percentage
chgenv(pubbnd(12), pubbnd("<5"))

# the two sides must line up
try(chgenv(now, prev[1, ]))
```

---

chn *Tamper-evident chain of capsule manifests*

Description

Keeps a history of runs (one manifest per run) as a hash chain: each entry records the digest of the entry before it, so the chain's integrity covers the ORDER and COMPLETENESS of the history, not merely the contents of each manifest. Deleting an entry, inserting one, or editing one breaks the links from that point onward, and `chnvrf()` reports the first index where the break occurs. Use it when the record of which analyses were run, and in what order, has to be auditable.

Usage

```
chnnew()

chnapp(chain, entry, label = NA_character_)

chnhead(chain)

chnseal(chain)

chnvrf(chain)
```

Arguments

`chain`	A chain from `chnnew()` or `chnapp()`.
`entry`	Any object to record. It is digested through its own deterministic serialization, so lists and data frames are accepted as readily as strings.
`label`	Optional short label for the entry (the first element of `as.character(label)` is kept); `NA` by default. Covered by the entry's digest.

Details

chnnew() starts an empty chain, chnapp() adds an entry, chnhead() returns the last link, chnvrf() checks every link and chnseal() returns one digest to sign.

This is the structure behind an append-only audit log, and it is what a per-manifest digest alone cannot give you: individually valid manifests say nothing about whether any were removed.

Each entry stores payload, the SHA-256 of the entry (of its UTF-8 bytes for a single string, otherwise of serialize(entry, version = 2, xdr = TRUE) with the four bytes that name the writer's R version set to zero), prev, the previous link (64 zeros for the first), scheme (2) and digest = SHA-256 of "v2|<prev>|<payload>|<label>", where <label> is "0" for a missing label and "1" followed by the label otherwise. The entry itself is not kept, only its digest; the label is covered, so editing it breaks the link. Zeroing the writer-version bytes makes the payload the same on every R version, for any R object (a canonical JSON encoding would cover only the objects JSON can hold); the rest of a version-2 stream depends only on the object.

Entries written by 0.5.12 and earlier carry no scheme. Their digest is the SHA-256 of the string "<prev>:<payload>" (the label not covered, and the payload of a non-string entry taken over serialized bytes that include the writer's R version). chnvrf() and chnseal() still accept them under those rules, with a warning that they were sealed with the weaker scheme. Appending to such a chain adds scheme-2 entries.

chnvrf() walks the chain from the start: an entry fails when its prev is not the previous link, its scheme is neither 2 nor absent, or its digest does not recompute (compared in constant time with cdgsteq()).

What the head covers, and what it does not:

chnhead() is the STORED digest of the last entry. Deleting an entry from the MIDDLE leaves that value untouched – the stored digests do not change, only the links between them stop agreeing – so the head alone will not notice. chnvrf() will, and names the index.

Conversely, truncating from the END leaves a perfectly valid prefix that chnvrf() accepts, while the head changes.

The two failures are complementary, which is why chnseal() exists: it verifies the links AND folds the entry count and every link digest into one value (SHA-256 of the count and the links joined by "|"), so a single signature over the seal detects an edit, a deletion, a reordering, an insertion and a truncation alike. Sign the seal, not the head.

Signing is what turns tamper-EVIDENT into tamper-PROOF: without a signature an attacker who rewrites the whole chain leaves it internally consistent, because recomputing every link is cheap.

Error: a chain that does not come from chnnew() (for chnapp(), chnhead() and chnvrf()).

Value

chnnew() and chnapp() return an object of class c("bricklayer_chain", "list") with entries (a list of entries, each with label, payload, prev, scheme and digest) and genesis (64 zeros). chnvrf() returns a list of class bricklayer_chain_check with valid (logical), n (integer, entries), broken_at (integer index of the first bad entry, NA when intact) and head. chnhead() returns the head digest (64 hex characters; the genesis value for an empty chain). chnseal() returns a single digest over the verified chain's length and links, or NA if the chain does not verify.

See Also

cpssign() to sign the seal, mrkroot() for pinning the contents of one capsule rather than a history.

Examples

```
# keep this example's
op <- options(rmoriebricklayer.xmssidxfile = tempfile())
# one-time index record out of your configuration directory; leave the
# default in use
ch <- chnnew()
ch <- chnapp(ch, "manifest for run 1", label = "run-1")
ch <- chnapp(ch, "manifest for run 2", label = "run-2")
ch <- chnapp(ch, "manifest for run 3", label = "run-3")
ch

# An intact chain verifies.
chnvrf(ch)$valid

# Editing an entry breaks it, and names where.
edited <- ch
edited$entries[[2]]$digest <- csha256("something else")
chnvrf(edited)$valid
chnvrf(edited)$broken_at

# So does deleting one, which a per-manifest digest would not catch.
dropped <- ch
dropped$entries[[2]] <- NULL
chnvrf(dropped)$valid

# Sign the SEAL, which covers the links and the length together.
key <- pqckg(height = 2)
sig <- cpssign(chnseal(ch), key)
cpsvrf(chnseal(ch), sig, signpubkey(key))

# Truncating the chain still seals, but to a different value, so the
# signature no longer verifies.
truncated <- ch
truncated$entries[[3]] <- NULL
cpsvrf(chnseal(truncated), sig, signpubkey(key))

# A chain whose links disagree has no seal to present at all.
chnseal(dropped)

# any object can be an entry, such as a whole manifest list
ch2 <- chnapp(chnnew(), list(run = 1, mean = 2.5), label = "list entry")
chnhead(ch2)

# only chains are accepted
try(chnvrf(list(entries = list())))
options(op)
```

chnkfile *Split a file into fixed-size chunks*

Description

Reads a file as bytes and returns it as consecutive chunks of a fixed size, ready for `mrkroot()`, `mrklvs()` and `mrkprf()`. Use it to pin a large data file as a Merkle root, so that a later change can be traced to the chunk that moved.

Usage

```
chnkfile(path, chnkbytes = 1048576L)
```

Arguments

`path`	Path to an existing file, a single string.
`chnkbytes`	Chunk size in bytes, a positive whole number (default 1048576, i.e. 1 MiB).

Details

The file is read sequentially with `readBin()` in pieces of `chnkbytes`; every chunk has exactly that many bytes except the last, which holds the remainder. All chunks are held in memory at once, so the memory used is about the size of the file. The default 1 MiB chunk is a compromise: smaller chunks localise a change more precisely but make the tree and its proofs larger. An edit that changes the file's length shifts every later chunk boundary, so only edits that keep the length localise to one chunk.

Errors: `path` that does not name an existing file (or names a directory); `chnkbytes` that is not a positive whole number.

Value

A list of raw vectors, in file order. A zero-length file gives an empty list. Chunks are returned as bytes rather than strings because a file is bytes: an R string cannot hold a zero byte, so a character chunk could not represent an arbitrary binary file at all.

See Also

`mrkroot()`, `sha512file()`

Examples

```
p <- tempfile()
writeLines(rep("some capsule content", 50), p)

# One small chunk size to show the splitting.
ch <- chnkfile(p, chnkbytes = 128)
length(ch)
lengths(ch)
```

```
# The chunks pin the file as a Merkle root.
mrkroot(ch)

# They are the file's bytes, so they concatenate back to it exactly.
identical(unlist(ch), readBin(p, "raw", file.size(p)))

# Overwriting bytes in place changes exactly one leaf.
b <- readBin(p, "raw", file.size(p))
b[300] <- as.raw(0x58)
q <- tempfile()
writeBin(b, q)
which(mrklvs(ch) != mrklvs(chnkfile(q, chnkbytes = 128)))

# a chunk size must be positive
try(chnkfile(p, chnkbytes = 0))
unlink(c(p, q))
```

`chwkfit` *Fit a Hawkes process by maximum likelihood (analytic gradient, fast routes)*

Description

Fits a self-exciting (Hawkes) point process to a series of event times, such as the times of shootings, break-ins or calls for service, and returns the estimated background rate, the branching ratio (the share of events triggered by earlier events) and the shape of the triggering kernel, with AIC, BIC and a goodness-of-fit test. Use it to ask whether events cluster in time beyond what a steady or seasonal rate explains.

Usage

```
chwkfit(
  times,
  hor,
  kernel = c("exponential", "weibull", "gamma", "lomax"),
  base = c("constant", "sinusoidal"),
  meth = "auto",
  eps = 1e-09,
  start = NULL
)
```

Arguments

`times`	Numeric vector of event times, sorted increasingly, all in `[0, horizon]`, in any time unit (days are the natural unit for the sinusoidal baseline's yearly cycle). At least 50 events.

hor	End of the observation window, one positive number in the same unit as times; the window is [0, horizon]. With the sinusoidal baseline (an annual cycle, times in days) at most 4194304 days: that baseline is integrated a day at a time.
kernel	The triggering kernel: "exponential" (the default), "weibull", "gamma" or "lomax".
base	"constant" (the default) or "sinusoidal" (trend plus a 365.25-unit cycle).
meth	One of "auto" (the default), "exact", "soe", "truncate", "em", "inar". "auto" is "exact" for the exponential kernel, "truncate" for Weibull and "soe" for Lomax and gamma.
eps	Error level, one number in (0, 1), default 1e-9: the relative error per intensity of "soe" and the kernel tail mass dropped by "truncate". Ignored by the other routes.
start	Optional numeric vector of starting values in the order of theta: the baseline terms (a0, or a0..a3), then eta, then the kernel parameters. Values outside the parameter box are moved onto it. NULL (the default) uses a start from the data.

Details

The model is the univariate Hawkes process

$$\lambda(t) = \nu(t) + \eta \sum_{t_j < t} g(t - t_j)$$

with a constant baseline ($\nu = e^{a_0}$) or a sinusoidal one ($\nu(t) = \exp(a_0 + a_1 t/T + a_2 \sin(2\pi t/365.25) + a_3 \cos(2\pi t/365.25))$, a trend plus a yearly cycle when times are in days, its integral by the trapezoid rule on max(64,floor(T) + 1) points), branching ratio $\eta \in [0.001, 0.99]$ and one of four normalised kernels g: exponential (decay beta), Weibull (shape alpha, scale lambda), gamma (shape alpha, rate beta) or Lomax (shape alpha, scale c). The log-likelihood is $\sum_i \log \lambda(t_i) - \int_0^T \lambda(t)\, dt$. Every route uses the analytic gradient of the shared C++ core (the code morie's Python arm calls), so one likelihood evaluation per point instead of one per parameter. The optimiser is a projected BFGS inside a fixed parameter box ($a_0 \in [-15, 15]$, the other baseline terms in $[-5, 5]$, kernel parameters bounded as listed in the source), at most 2000 iterations.

"exact" Ozaki's O(n) recursion for the exponential kernel; for Weibull and gamma the double sum stops where the kernel underflows to exactly 0 (the same value as the full sum); Lomax the full O(n^2) sum.

"soe" the completely monotone kernels (Lomax; gamma with shape < 1) as a sum of exponentials (Beylkin & Monzon 2010): relative error eps on every intensity, O(n K); a gamma kernel with shape >= 1 is truncated at eps.

"truncate" each event excites only lags with kernel tail mass above eps: an approximation for light-tailed kernels, O(n w).

"em" the EM algorithm (Veen & Schoenberg 2008), finished by the projected BFGS on the exact likelihood from EM's point (EM's steps shrink before it reaches the maximum). It is an R loop of up to 500 iterations and much slower than the other routes (tens of seconds for a few hundred events); because it starts from the same point but moves differently, it can end at a different local maximum.

"inar" Kirchner's (2017) INAR(p) least-squares estimator on binned counts (constant baseline only): a different, fast, approximate estimator.

"auto" "exact" for the exponential kernel, "truncate" for Weibull, "soe" for gamma and Lomax.

For the Weibull, gamma and Lomax kernels without start, a second deterministic start at the exponential fit (the shape-1 member) is also run and the better optimum kept, because the likelihood is multimodal. The exponential kernel without start is also run from a second start (branching ratio 0.1 and a decay a quarter of the reciprocal mean gap), and the better optimum kept: on data with little clustering its likelihood can have more than one maximum, one of them with the decay on its bound. A parameter on its bound (see atbound) is still a reason to refit from another start and compare nll.

The reported nll is the exact likelihood at the estimate whatever the route, so AIC is comparable across routes. An event exactly at hor is part of the record and counted, as in morie's Python fit. The Kolmogorov-Smirnov test compares the time-rescaling residuals of chwkres() with the uniform distribution (exact p-value up to 10,000 events without ties, asymptotic otherwise).

Events at the same instant do not excite each other: the intensity sums over strictly earlier events, as the definition of a Hawkes process (a simple point process, with no two events at one time) has it. Tied times are fitted that way, with a warning: data recorded to a resolution, such as daily dates, carry ties the process itself never makes, and treating them as exact biases the fit (Filimonov & Sornette 2015), so spread them uniformly across their interval first with chwkjit().

Errors: times with NA, unsorted, or outside [0, horizon]; hor not one positive finite number; fewer than 50 events; an event rate n / horizon outside $[e^{-15}, 100e^{15}]$ per unit time (rescale the times); eps not in (0, 1); start of the wrong length or with a non-finite likelihood; method = "soe" with a kernel other than Lomax or gamma; method = "inar" with the sinusoidal baseline. Long fits show a progress spinner (EM a bar) on stderr unless options(morie.quiet = TRUE).

Value

A list:

theta numeric, all estimates in the order baseline, eta, kernel.

basepar numeric, a0 (constant) or a0..a3 (sinusoidal), on the log scale.

braratio numeric, η, the expected number of events each event triggers.

kerpar numeric, the kernel parameters (beta; alpha, lambda; alpha, beta; or alpha, c).

nll, aic, bic numeric: the exact negative log-likelihood, $2k + 2\,nll$ and $k \log n + 2\,nll$.

n, horizon, kpar the number of events (integer), the window end and the number of parameters (integer).

ksstat, kspva numeric, the Kolmogorov-Smirnov test of the time-rescaling residuals against the uniform.

kernel, baseline, method character, the choices used (method after "auto" is resolved).

eps numeric, eps for "soe" and "truncate", NA otherwise.

converged logical, the optimiser stopped at a maximum within the parameter box (before its iteration limit, at a finite likelihood). A maximum on the box counts: atbound names the parameters that lie on it.

`atbound` character, the names of parameters that lie on the box (`character(0)` for an interior maximum).

A warning is given when η reaches its upper bound 0.99 (a near-critical, probably non-stationary fit). An estimate on the box is not an interior maximum of the likelihood: day-dated (tied) times drive a shape or decay to its wall, and a Lomax fitted to exponential data runs to its exponential limit.

References

Ozaki T (1979). Maximum likelihood estimation of Hawkes' self-exciting point processes. *Ann. Inst. Statist. Math.* 31, 145–155. DOI 10.1007/BF02480272

Beylkin G, Monzon L (2010). Approximation by exponential sums revisited. *Appl. Comput. Harmon. Anal.* 28, 131–149. DOI 10.1016/j.acha.2009.08.011

Veen A, Schoenberg FP (2008). Estimation of space-time branching process models in seismology using an EM-type algorithm. *JASA* 103, 614–624. DOI 10.1198/016214508000000148

Kirchner M (2017). An estimation procedure for the Hawkes process. *Quant. Finance* 17, 571–595. DOI 10.1080/14697688.2016.1211312

Filimonov V, Sornette D (2015). Apparent criticality and calibration issues in the Hawkes self-excited point process model: application to high-frequency financial data. *Quant. Finance* 15(8), 1293–1314. DOI 10.1080/14697688.2015.1032544

See Also

`chwkjit()`, for times recorded to a resolution; `chwkres()` for the residuals.

Examples

```
set.seed(1)
times <- sort(stats::runif(400, 0, 100))
fit <- chwkfit(times, 100, "exponential")
fit$braratio
fit$atbound # an interior maximum: the better of two starts
# one start alone can stop with the decay on its bound, at a worse nll
chwkfit(times, 100, "exponential", start = c(log(2.4), 0.4, 4))$atbound
fit$nll
chwkfit(times, 100, "lomax", meth = "soe", eps = 1e-8)$nll
chwkfit(times, 100, "exponential", base = "sinusoidal")$aic

# the fast approximate INAR estimator
chwkfit(times, 100, meth = "inar")$braratio

# daily dates: spread the ties across each day first
days <- floor(times)
jit <- chwkjit(days, hor = 100)
chwkfit(jit, 100)$converged

# too few events, or times past the horizon, are refused
try(chwkfit(times[1:20], 100))
try(chwkfit(times, 50))
```

chwkjit *Spread event times recorded to a resolution across their interval*

Description

Daily (or hourly, or second) timestamps put every event of an interval at one time, but a Hawkes process never has two events at one instant, and fitting the rounded times as exact biases the estimates (Filimonov & Sornette 2015). This function spreads each recorded time uniformly over the interval it stands for, so the times can be passed to `chwkfit()`.

Usage

```
chwkjit(times, res = 1, seed = 42, hor = Inf)
```

Arguments

`times`	Numeric event times, each the start of its resolution interval (for example whole days since an origin).
`res`	The width of the interval a recorded time stands for, one positive number in the unit of `times`: 1 (the default) for daily dates in days, `1/24` for hourly times in days.
`seed`	Seed of the uniforms, as in `cunif()`; default 42.
`hor`	End of the observation window, when it cuts an interval short: an event recorded at `t` is spread over `[t, min(t + resolution,horizon)]`, so a time dated on the horizon itself stays there (`Inf`, the default: no window).

Details

Each time becomes `t + w * U`, with `w = min(resolution, horizon - t)` and `U` uniform on [0, 1), and the result is sorted; this is the remedy Filimonov and Sornette use. The times are sorted (radix order) before the uniforms are assigned, so the result does not depend on the input order. The uniforms are the splitmix64 stream of `cunif()`, so the result is reproducible and, on daily dates with `seed = 42`, the same as the jitter rmorie and morie (Python) apply before their TPS Hawkes fits. R's random number generator is not used.

Errors: `times` with `NA` or non-finite values; `res` not one positive finite number; `hor` not one number; a time past `hor`.

Value

A numeric vector of the jittered times, the same length as `times`, sorted increasingly.

References

Filimonov V, Sornette D (2015). Apparent criticality and calibration issues in the Hawkes self-excited point process model: application to high-frequency financial data. *Quant. Finance* 15(8), 1293–1314. DOI 10.1080/14697688.2015.1032544

See Also

chwkfit(), cunif()

Examples

```
days <- c(0, 0, 0, 1, 3, 3)
chwkjit(days)
anyDuplicated(chwkjit(days))

# a horizon keeps the last day inside the window
chwkjit(c(0, 1, 2), hor = 2)

# hourly resolution, in days, and another seed
chwkjit(c(0, 0, 0), res = 1 / 24, seed = 7)

# times past the horizon are refused
try(chwkjit(c(0, 5), hor = 3))
```

---

chwknll *Hawkes-process negative log-likelihood (C backend)*

Description

The negative log-likelihood of a univariate self-exciting Hawkes process with constant baseline on [0, horizon], for the event times times. A Hawkes process is the natural model for arrivals that trigger further arrivals – repeat calls to a service, aftershocks, retweet cascades, revisions to an open-data release.

Usage

```
chwknll(
  times,
  hor,
  kernel = c("exponential", "weibull", "lomax", "gamma"),
  par = NULL
)
```

Arguments

times	Numeric vector of event times, sorted increasing, each in [0, horizon], in the time unit of your data (days, hours, ...). May be empty.
hor	End of the observation window, a single positive finite number in the same unit; the window starts at 0.
kernel	The triggering kernel, one of "exponential" (the default), "weibull", "lomax" and "gamma". Partial names are matched.
par	Numeric parameter vector (required), as described above: length 3 (a0, eta, beta) for "exponential", length 4 (a0, eta and the kernel's two parameters) for the others. Rates are per unit of times.

Details

Four triggering kernels are available. `"exponential"` is memoryless and evaluates by an O(n) recursion; the other three are not, so they cost O(n^2).

Parameters are passed on the scales the kernel is defined on: `par = c(a0, eta, ...)` where `a0` is the LOG baseline intensity ($\nu = e^{a0}$) and `eta` the branching ratio in (0, 1) – the expected number of children per event, so the process is stationary only for `eta < 1`. The remaining entries are the kernel's own shape parameters: `beta` (exponential), `alpha, lambda` (Weibull: shape and scale), `alpha, c` (Lomax: shape and scale, $g(u) = \alpha c^{\alpha}(u+c)^{-(\alpha+1)}$), `alpha, beta` (gamma: shape and rate). This is the likelihood `chwkfit()` maximises, on the same parameters.

The conditional intensity is $\lambda(t) = \nu + \eta \sum_{t_i<t} g(t-t_i)$ with g a probability density on $(0,\infty)$ (so η is the branching ratio), and the value returned is

$$-\ell = -\sum_i \log \lambda(t_i) + \int_0^T \lambda(t)\,dt,$$

with T = `hor` (Hawkes, 1971). For the exponential kernel $g(u) = \beta e^{-\beta u}$. With no events the value is the compensator alone, νT.

The feasible region, outside which the sentinel `1e12` is returned instead of a value: $-20 < a_0 < 20$; $10^{-6} < \eta < 0.999$; and for the kernel parameters, exponential $0.05 \le \beta \le 30$; Weibull shape in $[0.05, 20]$ and scale in $[0.001, 1000]$; Lomax shape in $[0.05, 30]$ and scale in $[0.001, 100]$; gamma shape in $[0.05, 20]$ and rate in $[0.05, 30]$. A missing parameter also gives the sentinel.

Refusals (errors, not the sentinel): `times` with a missing value, not sorted, or outside $[0, T]$; a `hor` that is not a single positive finite number; a `kernel` not among the four; and a `par` of the wrong length for the kernel. The quadratic kernels poll for a Ctrl-C as they run, so a long evaluation can be interrupted.

Value

A length-1 numeric: the negative log-likelihood, to be MINIMISED. A parameter set outside the feasible region returns the sentinel `1e12` rather than erroring, so the value can be handed straight to `stats::optim()` without the optimiser walking off the domain.

References

Hawkes AG (1971). Spectra of some self-exciting and mutually exciting point processes. *Biometrika* 58(1), 83–90. DOI 10.1093/biomet/58.1.83

Examples

```
set.seed(4)
times <- sort(stats::runif(40, 0, 10))

# Exponential kernel: log-baseline -0.5, branching 0.3, decay 1.2.
chwknll(times, 10, "exponential", c(-0.5, 0.3, 1.2))

# Lower is better, so this is what an optimiser minimises.
nll <- function(p) chwknll(times, 10, "exponential", p)
fit <- stats::optim(c(-0.5, 0.3, 1.2), nll)
```

```
fit$par

# A branching ratio at or above 1 is not a stationary process, and is
# reported as the infeasible sentinel rather than a number.
chwknll(times, 10, "exponential", c(-0.5, 1.5, 1.2)) == 1e12

# The heavier-tailed kernels take two shape parameters.
chwknll(times, 10, "gamma", c(-0.5, 0.3, 2, 1.5))

# Unsorted times are an error, not the sentinel.
try(chwknll(rev(times), 10, "exponential", c(-0.5, 0.3, 1.2)))
```

---

chwkres *Time-rescaling residuals of a Hawkes process*

Description

Transforms the event times into residuals that are independent and uniform on (0, 1) when the fitted Hawkes model is right, by the time-rescaling theorem. Plot them, or test them against the uniform with `stats::ks.test()`, to check the fit of `chwkfit()`; departures show where the model misses clustering or a trend.

Usage

```
chwkres(
  times,
  hor,
  kernel = c("exponential", "weibull", "gamma", "lomax"),
  par,
  base = c("constant", "sinusoidal")
)
```

Arguments

`times`	Numeric vector of event times, sorted increasingly, all in `[0, horizon]`, in any time unit (days are the natural unit for the sinusoidal baseline's yearly cycle). At least 50 events.
`hor`	End of the observation window, one positive number in the same unit as `times`; the window is `[0, horizon]`. With the sinusoidal baseline (an annual cycle, `times` in days) at most 4194304 days: that baseline is integrated a day at a time.
`kernel`	The triggering kernel: `"exponential"` (the default), `"weibull"`, `"gamma"` or `"lomax"`.
`par`	Numeric vector of parameters, as `chwkfit()` returns them in `theta`: the baseline terms, `eta` and the kernel parameters. Its length is 3 for the exponential kernel with a constant baseline, one more for a two-parameter kernel, three more for the sinusoidal baseline.
`base`	`"constant"` (the default) or `"sinusoidal"` (trend plus a 365.25-unit cycle).

Details

With the compensator $\Lambda(t) = \int_0^t \lambda(s)\, ds$, the residual of event i is

$$U_i = 1 - \exp(-(\Lambda(t_i) - \Lambda(t_{i-1})))$$

with $t_0 = 0$ (Brown et al. 2002). The rescaled gaps $\Lambda(t_i) - \Lambda(t_{i-1})$ are unit-rate exponential under the model, so the U_i are uniform. The model and kernels are those of `chwkfit()`; the shared C++ core computes the residuals in O(n) for the exponential kernel and O(n w) for the others. Tied times give tied residuals.

Errors: `times` with `NA`, unsorted or outside `[0, horizon]`; `par` of the wrong length (2 + kernel parameters for the constant baseline, 5 + kernel parameters for the sinusoidal one).

Value

A numeric vector of length `length(times)`, values in (0, 1).

References

Brown EN, Barbieri R, Ventura V, Kass RE, Frank LM (2002). The time-rescaling theorem and its application to neural spike train data analysis. *Neural Comput.* 14, 325–346. DOI 10.1162/08997660252741149

Examples

```
set.seed(2)
times <- sort(stats::runif(200, 0, 50))
U <- chwkres(times, 50, "exponential", c(log(3), 0.2, 2))
stats::ks.test(U, "punif")$p.value

# residuals of a fitted model, and a quick uniform Q-Q check
fit <- chwkfit(times, 50, "weibull")
U2 <- chwkres(times, 50, "weibull", fit$theta)
summary(U2)
plot(
  stats::ppoints(length(U2)),
  sort(U2),
  xlab = "uniform quantile",
  ylab = "residual"
)
abline(0, 1)

# par must match the kernel and baseline
try(chwkres(times, 50, "weibull", c(log(3), 0.2, 2)))
```

chwksim *Simulate event times from a fitted Hawkes process*

Description

Draws a new event history from a `chwkfit()` result by its cluster representation: immigrant events from the baseline (a Poisson process, thinned from its upper bound for a sinusoidal baseline), then each event's offspring, a Poisson number with mean the branching ratio, at delays drawn from the fitted kernel, generation after generation, until none fall before the horizon.

Usage

```
chwksim(fit, hor = NULL, seed = 1L)
```

Arguments

`fit`	A `chwkfit()` result.
`hor`	Number, the end of the window; default the fit's horizon.
`seed`	Whole number, default 1, for this call only (the session's stream is left as it was).

Value

The sorted event times (numeric), with the attribute `generation` (0 for an immigrant, 1 for its children, ...).

See Also

`plothwk()`, `chwkfit()`.

Examples

```
set.seed(1)
times <- sort(stats::runif(300, 0, 100))
fit <- chwkfit(times, 100)
sim <- chwksim(fit, seed = 2)
length(sim)
table(attr(sim, "generation"))
try(chwksim(list(a = 1)))
```

`cipw` *Trimmed inverse-probability weights (C backend)*

Description

The inverse-probability-of-treatment weights $1/e$ for the treated and $1/(1-e)$ for the untreated, with the propensity score clamped into `[trimlo, trimhi]` FIRST. Clamping matters: an untrimmed score near 0 or 1 produces a weight large enough for one observation to dominate the entire estimate. These weights reweight a treated and an untreated group so that each resembles the whole sample on the covariates behind the propensity score, the basis of inverse-probability-weighted estimates of a treatment effect (Rosenbaum and Rubin, 1983).

Usage

```
cipw(treat, ps, trimlo = 0.01, trimhi = 0.99)
```

Arguments

`treat`	Numeric or logical treatment indicator, coded 0/1 (`1` / `TRUE` is treated); `NA` allowed.
`ps`	Numeric vector of propensity scores (the estimated probability of treatment), each in $[0, 1]$, the same length as `treat`; `NA` allowed.
`trimlo`, `trimhi`	Clamp bounds for the score, single numbers with `0 < trimlo < trimhi < 1` (defaults 0.01 and 0.99).

Details

For unit i with treatment T_i and propensity e_i, the clamped score is $\tilde{e}_i = \min(\max(e_i, l), u)$ with l = `trimlo` and u = `trimhi`, and the weight is $1/\tilde{e}_i$ when $T_i = 1$ and $1/(1 - \tilde{e}_i)$ when $T_i = 0$. With the defaults no weight exceeds 100.

Refusals: vectors of different lengths, a treatment coded other than 0/1 (`"treat must be coded 0/1"`, naming the offending values), a propensity outside $[0, 1]$, and bounds that do not satisfy $0 < l < u < 1$. A missing treatment or propensity gives a missing weight for that unit only. The weights are not normalised; divide by their sum within each arm for stabilised (Hajek) estimates.

Value

A numeric vector of weights the length of `treat`, each at least 1 (and at most `1 / trimlo` or `1 / (1 - trimhi)`), `NA` where the treatment or propensity is missing.

References

Rosenbaum PR, Rubin DB (1983). The central role of the propensity score in observational studies for causal effects. *Biometrika* 70(1), 41–55. DOI 10.1093/biomet/70.1.41

See Also

cwtd() to average an outcome with these weights.

Examples

```
treat <- c(1, 0, 1, 0)
e <- c(0.5, 0.25, 0.02, 0.9)

cipw(treat, e)

# A balanced score gives weight 2 to either arm.
cipw(c(1, 0), c(0.5, 0.5))

# Without trimming the third observation would carry weight 50; the
# default clamp holds it to 100 at the 0.01 floor, and a looser floor
# tames it further.
cipw(treat, e, trimlo = 0.10)

# Logical treatment indicators work too.
cipw(c(TRUE, FALSE), c(0.4, 0.4))

# A treatment coded 1/2 is refused rather than read as all-control.
try(cipw(c(1, 2), c(0.5, 0.5)))
```

---

citecps *Generate a Data Citation From Provenance*

Description

Builds a ready-to-paste data citation, as plain text and as a BibTeX `@misc` entry, from the `dataset` and `resource` blocks of a provenance record. Use it so that every report and paper built from a capsule cites the data set in the same words, with the retrieval date and licence that were actually pinned.

Usage

```
citecps(prov)
```

Arguments

prov	A provenance list as returned by loadprov(), or NULL.

Details

The fields used, with their fallbacks:

- author: `dataset$publisher`, else `"Unknown publisher"`;
- title: `resource$name`, else `dataset$package_slug`, else `"Untitled dataset"`;

- year: the first four characters of `captured_at_utc`, else the current year; retrieval date: its first ten characters (omitted if absent);
- link: `https://doi.org/<dataset$doi>` if a DOI is recorded, else `resource$direct_url`, else `dataset$catalogue_page`;
- licence: `dataset$licence_short`, else `dataset$licence_name`.

The text form is `Publisher (year). Title [Data set]. link Retrieved date. Licence: x.` The BibTeX key is the package slug with non-alphanumeric characters removed, followed by the year. In the BibTeX entry the licence is written in the `note` field only when a retrieval date is present. No character in the fields is escaped for LaTeX. Error: `prov` that is not a list.

Value

A list with `text` and `bibtex`, each a length-1 character string (the BibTeX entry has embedded newlines), or `NULL` if `provenance` is `NULL`. The list has class `c("rmbl_citation", "list")`, which only changes how it prints (a box); its elements, and the JSON written from it, are those of a plain list.

Examples

```
prov <- list(
  captured_at_utc = "2026-06-23T04:41:40Z",
  dataset = list(publisher = "Ontario Ministry of the Solicitor General",
                 licence_short = "OGL-Ontario",
                 package_slug = "data-on-inmates-in-ontario"),
  resource = list(name = "Restrictive Confinement - Detailed Dataset",
                  direct_url = "https://data.ontario.ca/example.csv")
)
cit <- citecps(prov)

# Plain-text citation ready to paste.
cat(cit$text)

# BibTeX @misc entry for LaTeX bibliographies.
cat(cit$bibtex)

# a recorded DOI replaces the URL
prov$dataset$doi <- "10.5683/SP3/EXAMPLE"
cat(citecps(prov)$text)

# NULL provenance returns NULL (composes safely).
citecps(NULL)

# anything else that is not a list is an error
try(citecps("data-on-inmates-in-ontario"))
```

clncolnms *Tidy the column names of an ingested table*

Description

Converts column names to a consistent, syntactically valid form: transliterated to ASCII, non-alphanumerics collapsed to a single separator, and duplicates disambiguated with a numeric suffix. The counterpart of `janitor::clean_names()`. Use it right after reading an open-data file, so the rest of the script can refer to columns without backticks.

Usage

```
clncolnms(
  data,
  case = c("snake", "lower_camel", "upper_camel", "screaming_snake", "none"),
  sep = "_",
  quiet = FALSE
)
```

Arguments

`data`	A data frame, or a character vector of names.
`case`	`"snake"` (default, `total_pop`), `"lower_camel"` (`totalPop`), `"upper_camel"` (`TotalPop`), `"screaming_snake"` (`TOTAL_POP`), or `"none"` to normalise separators only and keep the letters' case.
`sep`	Separator, a string, used between words for `"snake"`, `"screaming_snake"` and `"none"`, and before the number that disambiguates a duplicate in every case (default `"_"`).
`quiet`	Logical: `TRUE` skips the boxed list of the names that changed, which is otherwise sent through `message()` (so `suppressMessages()` also silences it); default `FALSE`.

Details

Open-data extracts arrive with names like `"Total Population (2021)"` and `"% change"`, which need backticks everywhere and break silently when a re-release renames `"% change"` to `"% Change"` or adds a trailing space. Normalising once at ingestion makes the schema stable against that.

The steps, in order: transliterate to ASCII (`iconv(..., "ASCII//TRANSLIT")`, keeping the original where that fails); replace `%` by `pct`, `&` by `and` and `#` by `n` as words; split camelCase at a lower-to-upper boundary; turn every run of other characters into one word break; an empty name becomes `x`; join the words in the chosen case; prefix `x` to a name starting with a digit; and give repeated names the suffixes `<sep>2`, `<sep>3`, ... in order of appearance (the first keeps its name). The transliteration depends on the platform's iconv, so an accented letter may come out slightly differently on different systems.

Record the mapping in the capsule manifest. Cleaning names changes what a downstream script must refer to, so an unrecorded cleaning is itself a reproducibility hazard – the returned object carries the original names in its `"orignms"` attribute for exactly that.

Value

For a data frame, the data frame with new names and an `"orignms"` attribute (character, the names before cleaning); for a character vector, the cleaned character vector.

Examples

```
clncolnms(c("Total  Population (2021)", "% change",
                     "Ville / City", "dup", "dup"))

# Applied to a data frame, with the original names retained.
df <- data.frame(`Total Pop` = 1:2, `% change` = 3:4,
                 check.names = FALSE)
cleaned <- clncolnms(df)
names(cleaned)
attr(cleaned, "orignms")

# Other cases.
clncolnms(c("Total Pop"), case = "lower_camel")
clncolnms(c("Total Pop"), case = "upper_camel")
clncolnms(c("Total Pop"), case = "screaming_snake")

# a different separator, and a name that starts with a digit
clncolnms(c("2021 count", "Rate per 1000"), sep = ".")

# an unknown case is an error
try(clncolnms("a b", case = "kebab"))
```

---

clrdcmp *Decompose a change in a clearance rate into rate and case-mix effects*

Description

An aggregate clearance rate – cases disposed per case received by a court, or offences cleared per offence reported to a police service – is a weighted mean of the rates of its parts (offence groups, courts, case types), weighted by each part's share of the denominator. It can therefore fall while the rate of every part rises, if the work shifts toward the parts that clear more slowly: Simpson's paradox. This function compares two periods (or two jurisdictions), splits the change in the aggregate exactly into the change in the parts' rates and the change in the mix, reports the rate standardised to a common mix, tests the within-part change, and says whether the comparison is a reversal.

Usage

```
clrdcmp(num, den, grp, cmp, base = NULL, conflvl = 0.95)
```

Arguments

num	Cases disposed (or cleared), one per part and comparison.
den	Cases received (or reported), one per part and comparison.
grp	The part each row belongs to: offence group, court, case type.
cmp	The comparison each row belongs to: exactly two values, such as two years or two jurisdictions.
base	Which value of cmp is the baseline. Default: the first in sorted order.
conflvl	Confidence level. Default 0.95.

Details

With part rates r_g and denominator shares w_g, the aggregate is $R = \sum_g w_g r_g$. Kitagawa's symmetric decomposition writes the change between comparisons A and B as

$$R_B - R_A = \sum_g \bar{w}_g (r_{gB} - r_{gA}) + \sum_g \bar{r}_g (w_{gB} - w_{gA}),$$

with $\bar{w}_g$ and $\bar{r}_g$ the averages over A and B: the first sum is the rate effect (the change the aggregate would show with the mix held at its average), the second the mix effect. The two add to the observed change with no residual. Because the shares' changes add to zero, each part's mix term is reported as $(\bar{r}_g - \bar{R})(w_{gB} - w_{gA})$, centred on the average aggregate rate $\bar{R}$: the total is the same, and a part's term is negative when work moved toward a part that clears below average.

The within-part change is estimated by the Mantel-Haenszel rate ratio, treating each part's numerator as a Poisson count over its denominator, with the Greenland-Robins variance; the p-value is for the hypothesis that the ratio is 1. Each part's own rate ratio has the exact conditional (binomial) interval. A court's clearance rate can exceed 1, so it is treated as a rate rather than a proportion; for police clearance, where it is a proportion, the Poisson treatment is conservative.

The decomposition is only as good as the parts: a different grouping (courts instead of offence groups) can tell a different story, and both can be right. Demers and Rossmo found 210 reversals in fifteen years of clearance rates for Canada's fifty largest police services, four of them reversed in every crime category, and advise comparing clearance rates only within comparable mixes.

Value

A list of class rmbl_clearance_decomposition:

groups one row per part: the counts, rates and denominator shares (weight) in both comparisons, the rate ratio with its exact interval, and the part's rateeff and mixeff.

overall the aggregate rates, their change, its rateeff and mixeff (which add to it exactly), the comparison rate standardised to the baseline mix (staonbase) and to the average mix (staonave).

rateratio the crude rate ratio and the Mantel-Haenszel rate ratio over the parts, each with its interval, and the p-value for the stratified ratio.

simpson "full" when every part moved one way and the aggregate the other; "mix" when the aggregate moved against the mix-standardised change (the mix outweighed the rates); "none" otherwise; with the number of parts that rose and fell.

References

Simpson, E. H. (1951). The interpretation of interaction in contingency tables. *Journal of the Royal Statistical Society, Series B*, 13(2), 238-241.

Kitagawa, E. M. (1955). Components of a difference between two rates. *Journal of the American Statistical Association*, 50(272), 1168-1194. DOI 10.2307/2281213

Das Gupta, P. (1993). *Standardization and Decomposition of Rates: A User's Manual*. U.S. Bureau of the Census, Current Population Reports P23-186. Washington, DC.

Greenland, S. and Robins, J. M. (1985). Estimation of a common effect parameter from sparse follow-up data. *Biometrics*, 41(1), 55-68.

Demers, S. and Rossmo, D. K. (2015). Simpson's paradox in Canadian police clearance rates. *Canadian Journal of Criminology and Criminal Justice*, 57(3), 424-434. DOI 10.3138/cjccj.2014-E29

See Also

`caseflow` for the clearance rate over time.

Examples

```
g <- read.csv(system.file(
  "extdata",
  "ocj_offence_groups.csv",
  package = "rmoriebricklayer"
))
g <- g[g$year %in% c(2014, 2022), ]
cd <- clrdcmp(g$disposed, g$received, g$offence_group, g$year)
cd
plot(cd)

# Simpson's paradox in four numbers: each part clears better in B, yet the
# aggregate falls, because B's workload moved to the part that clears less
toy <- clrdcmp(
  num = c(90, 20, 19, 60),
  den = c(100, 50, 20, 140),
  grp = c("minor", "serious", "minor", "serious"),
  cmp = c("A", "A", "B", "B")
)
toy

# every part must appear in both comparisons
try(clrdcmp(c(5, 6, 7), c(10, 10, 10), c("x", "y", "x"), c(1, 1, 2)))
```

`cmlrisk` *Cumulative risk of a first event by age*

Description

The chance that a person experiences an event (a first arrest, a first conviction, a first jail stay, a first child protection placement) by each age, from first events and the population by age, as a synthetic cohort; with mortality as a competing risk when deaths are given.

Usage

```
cmlrisk(age, fireve, pop, atrisk = NULL, dthrate = NULL, conflvl = 0.95)
```

Arguments

`age`	The ages, in order.
`fireve`	First events at each age.
`pop`	The population at each age.
`atrisk`	Optional: the persons at risk at each age, when known, in place of the synthetic-cohort assumption.
`dthrate`	Optional death rates by age (probability of dying within the age).
`conflvl`	Confidence level of the cumulative risk limits (default 0.95).

Details

With E_a first events at age a and N_a persons at risk (those who have not had the event), the hazard is $h_a = E_a/N_a$. When only the population P_a is known, the persons at risk are taken as $P_a(1 - CR_{a-1})$ (the synthetic-cohort assumption). Without mortality, $CR_a = 1 - \prod_{x \le a}(1 - h_x)$. With death rates q_x, the multiple-decrement table carries survivors $l_{x+1} = l_x(1 - h_x - q_x)$ and $CR_a = \sum_{x \le a} l_x h_x / l_0$, the risk of the event before death. The interval (without mortality) is Greenwood's on the log of the survival: $\mathrm{Var}(\log S_a) = \sum_{x \le a} h_x/(N_x(1 - h_x))$.

Counting first events needs records that link a person's events over time; an agency's counts of all events overstate first events. Events of people who do not live in the area, counted against its population, bias the risk upward.

Value

A data frame of class `rmbl_cumrisk`: `age`, `hazard`, `atrisk`, `cmlrisk`, `lower`, `upper`.

References

Bonczar TP (2003). Prevalence of imprisonment in the U.S. population, 1974-2001. Bureau of Justice Statistics, NCJ 197976.

Applied to arrests, convictions and jail in New York City: Hepburn P, Kohler-Hausmann I, Zorro Medina A (2019). Cumulative risks of multiple criminal justice outcomes in New York City. Demography 56(3):1161-1171. DOI 10.1007/s13524-019-00781-7

Examples

```
age <- 16:25
first <- c(900, 1400, 1700, 1600, 1400, 1100, 900, 700, 600, 500)
pop <- rep(100000, 10)
cmlrisk(age, first, pop)
```

---

cmom *Mean, variance, skewness and kurtosis (C backend)*

Description

The four standard descriptive moments of a numeric vector – where it is centred, how spread out it is, whether it leans to one side and how heavy its tails are – in one call to the compiled core. These are the four numbers a data description or a drift check usually reports together.

Usage

```
cmom(x)
```

Arguments

x	Numeric (or logical) vector; coerced with `as.numeric()`.

Details

The mean is base R's (an extended-precision sum followed by a corrective pass); the deviations from it are then divided by their largest magnitude before any power is taken, and the scale is restored at the end. That avoids both the cancellation of the textbook "sum of squares minus square of sum" formula and overflow of fourth powers for values beyond about 10^{77}. It reads the data three times (mean, largest deviation, moments), in $O(n)$ time. The variance uses the `n - 1` denominator, matching `stats::var()`. The shape statistics use the *sample moment* definitions

$$g_1 = m_3 / m_2^{3/2}, \qquad g_2 = m_4 / m_2^2 - 3,$$

with $m_k = n^{-1} \sum_i (x_i - \bar{x})^k$, so kurtosis is reported as EXCESS kurtosis: a normal sample sits near zero, not near three. These are the uncorrected moment ratios (type 1 in the classification of Joanes and Gill, 1998), not the bias-adjusted versions some software reports.

Edge cases are `NaN`, never an error: an empty vector gives four `NaN`; one value gives its mean and `NaN` for the rest; `skewness` needs at least 3 values and `kurtosis` at least 4; any `NA` or `NaN` in `x` makes every entry `NaN`. A constant vector has variance 0 and `NaN` shape statistics. A character, factor or list `x` is refused (`"must be numeric"`).

Value

A named length-4 numeric vector: `mean` (units of `x`), `variance` (squared units of `x`, `n - 1` denominator), `skewness` and `kurtosis` (both unitless; kurtosis as excess over the normal's 3).

References

Joanes DN, Gill CA (1998). Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society: Series D (The Statistician)* 47(1), 183–189. DOI 10.1111/1467-9884.00122

See Also

`csdev()`, `cqnt()` and `cmad()` for robust alternatives that one extreme value cannot move.

Examples

```
cmom(c(2, 4, 4, 4, 5, 5, 7, 9))

# The first two entries agree with base R.
m <- cmom(1:10)
all.equal(m[["mean"]], mean(1:10))
all.equal(m[["variance"]], stats::var(1:10))

# A symmetric sample has no skew; excess kurtosis is near 0 for normal
# data and positive for a heavy-tailed sample.
cmom(c(-2, -1, 0, 1, 2))[["skewness"]]
cmom(c(rep(0, 20), -8, 8))[["kurtosis"]] > 0

# Too short to define a shape statistic: NaN rather than a guess.
cmom(c(1, 2))
```

---

cnormlpdf *Normal log-density (C backend)*

Description

The logarithm of the normal (Gaussian) density at each value of `x`, computed directly from the formula rather than as `log(dnorm(x))`. Use it for log-likelihoods: summing log-densities is how a normal model is scored, and computing them directly keeps the sum finite far into the tails, where the density itself underflows to zero and its logarithm would be `-Inf`.

Usage

```
cnormlpdf(x, mean = 0, sd = 1)
```

Arguments

`x`	Numeric vector of quantiles (the observed values).
`mean`	The distribution's mean, a single number in the units of `x` (default 0).
`sd`	The distribution's standard deviation, a single positive number in the units of `x` (default 1).

Details

Each value is

$$\log \phi(x; \mu, \sigma) = -\log \sigma - \tfrac{1}{2}\log(2\pi) - \frac{(x-\mu)^2}{2\sigma^2},$$

evaluated by the shared core, and agrees with `stats::dnorm(x, mean, sd, log = TRUE)` to rounding.

`mean` and `sd` are single numbers: a vector of either is refused, so this is not vectorised over the parameters the way `stats::dnorm()` is. A non-finite `mean`, and an `sd` that is not a positive finite number, are errors, as in `cnormpdf()`; a missing `x` gives `NA`. A character, factor or list `x` is refused (`"must be numeric"`).

Value

A numeric vector the length of `x`: the natural-log density at each value (in log units per unit of `x`). Equivalent to `stats::dnorm(x, mean, sd, log = TRUE)`.

See Also

`cmom()` for estimating `mean` and `sd` from data.

Examples

```
cnormlpdf(c(-1, 0, 1))

# Identical to stats::dnorm(log = TRUE).
all.equal(cnormlpdf(-2:2, 0.3, 1.7),
          stats::dnorm(-2:2, 0.3, 1.7, log = TRUE))

# Still finite where the density itself underflows to zero.
stats::dnorm(50)                  # 0
cnormlpdf(50)            # about -1251

# A log-likelihood: the sum of the log-densities of a sample.
y <- c(4.8, 5.1, 5.6, 4.9, 5.3)
sum(cnormlpdf(y, mean = mean(y), sd = stats::sd(y)))

# A standard deviation of zero has no density, and the parameters are
# single numbers.
try(cnormlpdf(1, sd = 0))
try(cnormlpdf(1, mean = c(0, 1)))
try(cnormlpdf("1"))
```

`cnormpdf` *Normal density (C backend)*

Description

Evaluates the density of the normal distribution with a given mean and standard deviation at each value of `x`, in compiled code. It is the same kernel that other packages reach through `LinkingTo: rmoriebricklayer`, and for a positive standard deviation it agrees with `dnorm`.

Usage

```
cnormpdf(x, mean = 0, sd = 1)
```

Arguments

`x`	Numeric vector of quantiles at which the density is evaluated.
`mean`	Number, default `0`: the mean μ.
`sd`	Number, default `1`: the standard deviation σ; must be positive for a finite result.

Details

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

The function is vectorised over `x` only; `mean` must be a single finite number and `sd` a single positive number, and anything else is an error (`dnorm()` recycles them, and gives `Inf` or `NaN` for `sd = 0` or below). `NA` in `x` gives `NA`; `x = Inf` gives 0. Character, factor or list input for `x` is an error.

Value

A numeric vector the length of `x`.

See Also

`cstat`, `dnorm`.

Examples

```
# Standard normal density at a few quantiles.
cnormpdf(c(-1, 0, 1))

# Peak of the standard normal is 1/sqrt(2*pi) at x = 0.
cnormpdf(0)

# Shift and scale via mean and sd.
cnormpdf(c(0, 5, 10), mean = 5, sd = 2)

# Identical to stats::dnorm().
```

```
all.equal(cnormpdf(-2:2, 0, 1), stats::dnorm(-2:2, 0, 1))

# a non-positive sd and text are refused
try(cnormpdf(0, sd = 0))
try(cnormpdf("0"))
```

---

cnscwb *A Community Well-Being score from the census profile*

Description

Indigenous Services Canada's Community Well-Being (CWB) index (cwbscr) computed from the public 2021 Census Indigenous Population Profile, for each geography and group.

Usage

```
cnscwb(
  x,
  dwell = NULL,
  groups = c("First Nations", "Métis", "Inuit", "Non-Indigenous"),
  bounds = c(2650, 75000),
  resid = "Total - Residence on or off reserve",
  minpop = 250
)
```

Arguments

x	A cnsprof of a population table.
dwell	Optional cnsprof of the matching household table.
groups	The groups to score (default First Nations, Metis, Inuit and non-Indigenous people).
bounds	Income bounds of cwbscr.
resid	On or off reserve, in the tables that split it (default the total).
minpop	The smallest population scored (default 250; the Department uses 65).

Details

The Department computes the CWB from census microdata; the public profile gives close but not the same inputs, and this function says where it departs:

income per capita: the number of total income recipients aged 15 and over times their average total income, over the population in private households (all ages), as the CWB does.

education the shares with at least a high school diploma (all but those with no certificate, diploma or degree) and with a bachelor's degree or higher, both among those aged 25 and over: the profile has no age group of 20 and over, which the CWB uses for the first.

labour force the participation rate and the employment rate of participants (100 less the unemployment rate) of those aged 25 to 64: the CWB uses 20 to 64.

housing the shares of households with one person or fewer per room and needing no major repairs, from the household profile (`dwell`), which splits Indigenous from non-Indigenous households only, so First Nations, Metis and Inuit people all get the pooled Indigenous households' housing: the CWB counts persons, not households. Without `dwell` the housing component and the score are `NA`.

A group of fewer than `minpop` people in a geography gets no score. The Department scores communities of 65 people or more; the default here is 250, because the public profile's counts are rounded to multiples of 5 and its long-form counts rest on a few dozen households at that size. Geographies are matched by DGUID, so places that share a name stay apart. The inputs are those of the 2021 profile (98-510-X2021001 to -016): other editions name their characteristics differently. Read the result as a CWB-style description of socio-economic conditions, close to but not the Department's index, and never as a measure of the wellness of First Nations, Metis or Inuit people as they define it (see `cwbscr`).

Value

A data frame of class `rmbl_census_cwb`: `geo`, `dguid`, `group`; the components `income`, `education`, `housing`, `labour` and the score `cwb` (0 to 100, `NA` where not scored); and the inputs `incpercap` (dollars), `highsch25p`, `uni25p`, `population`, `participation`, `employment`, `nomajrep`, `notcro` (percentages).

References

Indigenous Services Canada (2019). Report on trends in First Nations communities, 1981 to 2016 (Community Well-Being index).
sac-isc.gc.ca/eng/1345816651029/1557323327644

See Also

`cwbscr`.

Examples

```
p <- cnsprof(indiex("censuspop"))
h <- cnsprof(indiex("censushh"))
cnscwb(p, h)[, c("geo", "group", "income", "education", "housing", "labour",
                     "cwb")]
```

`cnsprof` *Read the 2021 Census Indigenous Population Profile*

Description

Reads a table of the Indigenous Population Profile, 2021 Census of Population (Statistics Canada catalogue 98-510-X2021001 to 98-510-X2021016), as the data service mirrors it, into one row per geography, gender, age group, characteristic and group: First Nations, Metis, Inuit, the other identity groups and non-Indigenous people (or, in the household tables, Indigenous and non-Indigenous households), each in its own rows. The data service keeps these tables under `census2021_ipp/`.

Usage

```
cnsprof(x)
```

Arguments

`x`	A profile table as published (a data frame read from its CSV), or a data service key (`"census2021_ipp/x2021001_pop"`).

Details

The profile gives each characteristic in a hierarchy shown only by indentation ("Total - Highest certificate, diploma or degree", "High (secondary) school diploma or equivalency certificate", ...): the level, the parent and the topic's total (`rootid`) are worked out from it, so a share can be taken of its parent. A topic marked "25% sample data" comes from the long-form sample (`sample = "25%"`); the others from the full count (`"100%"`). Counts are randomly rounded to multiples of 5 (of 10 below 10) by Statistics Canada, and a cell suppressed to protect confidentiality (`x`), too unreliable (`F`), not available (`..`) or not applicable (`...`) has no value here; `E` values are kept and marked. The global non-response rates of the geography (`tnrsf`, `tnrlf`, in percent) are kept: Statistics Canada suppresses an area whose long-form rate is 50% or more, and a high rate means a larger error than the sample alone.

Value

A data frame of class `rmbl_census_profile`: `geo`, `dguid`, `geolvl`, `residence` (on or off reserve, in the tables that split it), `gender`, `age`, `chaid`, `characteristic`, `level`, `parid`, `rootid`, `sample`, `group`, `value`, `quality`, `tnrsf`, `tnrlf`.

References

Statistics Canada (2023). Indigenous Population Profile, 2021 Census of Population. Statistics Canada Catalogue no. 98-510-X2021001. Released June 21, 2023. Published at www12.statcan.gc.ca/census-recensement/2021/dp-pd/ipp-ppa/index.cfm?Lang=E (the site answers scripts with 403); the Way-back Machine holds a copy of that address captured on 8 March 2026 (web.archive.org, snapshot 20260308015010).

Statistics Canada (2022). Guide to the Census of Population, 2021. Catalogue no. 98-304-X.

See Also

`profshr`, `cnsstd`, `cnscwb`, `indiex`.

Examples

```
p <- cnsprof(indiex("censuspop"))
head(p[p$characteristic == "Foster children" & p$geo == "Canada", ])
```

---

cnsstd *An Indigenous standard population from the census*

Description

The age structure of a group in a geography of the 2021 Census Indigenous Population Profile, as the standard of `agestd` or `stdprev`.

Usage

```
cnsstd(
  x,
  grp = "Indigenous identity",
  geo = "Canada",
  ages = c("0 to 4 years", "5 to 9 years", "10 to 14 years", "15 to 19 years",
    "20 to 24 years", "25 to 34 years", "35 to 44 years", "45 to 54 years",
    "55 to 64 years", "65 to 74 years", "75 years and over"),
  resid = "Total - Residence on or off reserve"
)
```

Arguments

`x`	A `cnsprof`.
`grp`	The group (default `"Indigenous identity"`).
`geo`	The geography: its name (default `"Canada"`) or, where names repeat, its DGUID.
`ages`	The age groups, as the profile names them.
`resid`	On or off reserve, in the tables that split it (default the total).

Details

Comparisons of First Nations, Metis and Inuit health are standardised to the Indigenous population's own, younger, age structure (Kumar and Tjepkema 2019): a standard as old as Canada's gives weight to ages where few Indigenous people are. This takes the population in private households of each age group (the profile's first characteristic) for the group and geography given. Age groups that do not overlap are chosen by default (0 to 4, 5 to 9, 10 to 14, 15 to 19, 20 to 24, 25 to 34, 35 to 44, 45 to 54, 55 to 64, 65 to 74, 75 and over); collapse them to the age groups of the rates being standardised.

Value

A data frame: `age`, `population`, `weight` (shares summing to 1).

References

Kumar MB, Tjepkema M (2019). Suicide among First Nations people, Metis and Inuit (2011-2016): findings from the 2011 Canadian Census Health and Environment Cohort (CanCHEC). Statistics Canada catalogue no. 99-011-X2019001.

See Also

agestd, stdprev.

Examples

```
p <- cnsprof(indiex("censuspop"))
cnsstd(p)
```

---

cnttrnd *Trend in a count series, as a rate ratio per period*

Description

Estimates the trend in a series of counts (arrests, admissions, incidents per year) as a rate ratio per period: 0.85 means the count falls by 15% a year. It fits a Poisson log-linear model and, when the counts vary more than a Poisson model allows, widens the interval. An offset carries the denominator when the exposure varies, so the trend is then in the rate rather than in the count.

Usage

```
cnttrnd(y, x = NULL, offset = NULL, conflvl = 0.95)
```

Arguments

y	Numeric vector of counts (non-negative whole numbers), in period order.
x	Numeric vector of periods, default NULL (the position). The rate ratio is per unit of x.
offset	Optional numeric vector of exposures, one per period – a population, a number of admissions, a number of days. Positive. The trend is then in the rate rather than in the count.
conflvl	Number in (0, 1), default 0.95: the confidence level of the rate-ratio interval.

Details

The model is $\log \mu_t = \beta_0 + \beta_1 (x_t - \bar{x}) + \log o_t$, with o_t the offset (1 when none), fitted by iteratively reweighted least squares (at most 50 iterations). The rate ratio is e^{β_1}, with a Wald interval $e^{\beta_1 \pm z\,\mathrm{se}}$.

The dispersion is reported because a Poisson fit assumes it is one. It is the Pearson statistic $\sum (y - \mu)^2/\mu$ divided by $n - 2$. When it exceeds 1.5 the quasi-Poisson interval is returned instead: the

standard error is multiplied by the square root of the dispersion and the t distribution on $n - 2$ degrees of freedom replaces the normal, and `overdispersed` is set.

Missing values are dropped. Errors: negative or fractional counts; `x` and `y` of different lengths; infinite `x`; every count zero; an offset of the wrong length or not positive; fewer than three periods; a `conflvl` outside (0, 1). When the counts are zero at one end of the series only (`c(0, 0, 5)`) the likelihood rises without bound and there is no finite trend: a warning says so, the rate ratio is `Inf` (or 0) and the interval and p-value are `NA`. There is a `plot()` method for the result.

Value

A list of class `c("rmbl_count_trend", "list")` with `rateratio` (per period), `lower`, `upper`, `p_value` (test of no trend), `logslope` (β_1), `se` (its standard error, scaled when overdispersed), `fitted` (fitted counts), `dispersion`, `overdispersed` (logical), `n`, `data` (list of the `x` and `y` used) and `method`.

References

Bilder, C. R. and Loughin, T. M. *Analysis of Categorical Data with R*, 2nd edn. Chapman and Hall/CRC, on the quasi-likelihood treatment of an overdispersed Poisson fit: the variance is scaled by an estimated dispersion, which widens the interval while leaving the point estimate alone. That is the behaviour reported here through `dispersion` and `overdispersed`.

See Also

`trndtst`, `ratechg`.

Examples

```
# A count falling by about 15% a year.
set.seed(3)
y <- stats::rpois(8, lambda = 200 * 0.85^(0:7))
fit <- cnttrnd(y)
round(fit$rateratio, 3)
c(fit$lower, fit$upper)

# With a varying denominator the trend is in the rate.
cnttrnd(c(20, 25, 30), offset = c(1000, 1500, 2500))$rateratio

# Counts that swing far more than Poisson noise get the wider interval.
wild <- cnttrnd(c(10, 50, 5, 60, 8, 70))
wild$dispersion
wild$method

try(cnttrnd(c(1.5, 2, 3)))
```

`co2fp` *Energy and carbon footprint of a computation*

Description

Evaluates `expr` and reports the electricity it used and the CO2-equivalent that electricity implies, by one of two methods, for the grid it ran on, offline. Use it to report the energy and emissions of an analysis next to its results (as journals and funders increasingly ask), or to compare two ways of running the same computation. The result keeps the value of `expr`, and lists every figure that was assumed rather than measured.

Usage

```
co2fp(
  expr,
  loc = NULL,
  meth = c("auto", "rapl", "model"),
  cpupwrw = NULL,
  usage = "process",
  loadcur = c("linear", "codecarbon"),
  cores = NULL,
  memgb = NULL,
  mempwrwpergb = 0.3725,
  pue = 1,
  rapldir = "/sys/class/powercap",
  procstat = "/proc/stat"
)
```

Arguments

`expr`	The expression to evaluate. It is evaluated once, in the calling environment, when the function reaches it (lazily, as a promise).
`loc`	The location, passed to `co2int`. It is a location code, alpha-3 code or name, a number in gCO2e/kWh, or `NULL` (the default) to detect it.
`meth`	Character scalar: `"auto"` (the default; RAPL when readable, else the model), `"rapl"` (measured; an error when the counters are not readable) or `"model"` (always the Green Algorithms model).
`cpupwrw`	The CPU's total thermal design power in watts (the model's P_{cpu}); `NULL` uses codecarbon's unknown-CPU fallback.
`usage`	`"process"` (measured from this process's CPU time), `"machine"` (measured from the whole machine's CPU accounting), or a number in between 0 and 1. `NULL` means `"process"`.
`loadcur`	Character scalar: `"linear"` (the default; power proportional to utilisation, as Green Algorithms) or `"codecarbon"` (codecarbon's curves, see Details). Applies only when `cpupwrw` is given.

cores	Number of logical cores the process utilisation is measured against; NULL detects it.
memgb	Memory charged to the computation in GB; NULL detects the machine's total memory where it can, else assumes 8 GB.
mempwrwpergb	Memory power per GB (default 0.3725 W/GB, Green Algorithms; codecarbon v2 used 0.375).
pue	Power usage effectiveness: 1 for a laptop or desktop (default); 1.67 is the Green Algorithms default for a data centre, 1.56 the Uptime Institute 2024 global average. Must be a single number of at least 1.
rapldir	Character scalar, the powercap directory holding the RAPL domains; default "/sys/class/powercap" (Linux).
procstat	Character scalar, the kernel CPU accounting file read for usage = "machine"; default "/proc/stat" (Linux).

Details

Measured (method = "rapl"): the CPU package energy counters (RAPL, microjoules, read from rapldir) before and after, with wrap-around handled through max_energy_range_uj. This is what codecarbon does on Linux; the counters cover the CPU packages only, so memory is still modelled. Available when raplok() is TRUE.

Modelled (method = "model"): the Green Algorithms formula (Lannelongue, Grealey and Inouye 2021),

$$E = t\,(P_{cpu}\,u + M\,P_{mem})\,PUE$$

with t the running time, P_{cpu} the CPU's thermal design power in watts, u its utilisation, M the memory in GB, $P_{mem} = 0.3725$ W/GB and PUE the data-centre overhead. When the CPU's power is unknown (cpupwrw = NULL) the fallback is codecarbon's: 50% of a constant 85 W, that is 42.5 W charged for the whole running time whatever the measured utilisation (the utilisation is still reported, but not applied).

Utilisation is measured, not assumed, in one of two ways. usage = "process" (the default) is this R process's own CPU time, children included, divided by the wall time and the core count: codecarbon's process tracking mode, which charges only what the computation itself used. usage = "machine" is the whole machine's busy share over the run, from /proc/stat on Linux: codecarbon's machine tracking mode, right when the computation is the only thing the machine is doing or when the question is what the machine drew. Where /proc/stat is absent the process figure is used and the assumptions say so. A number between 0 and 1 fixes the utilisation, as the Green Algorithms calculator asks for it. loadcur chooses how power scales with that utilisation: linearly (Green Algorithms) or along codecarbon's curves, a floor at 10% of P_{cpu} for a process and $0.1 + 0.9u^3$ for a machine, so a figure can be made comparable with either tool.

Cross-checked against codecarbon 3.3.1 on one workload with the same forced inputs: memory energy identical (5.96 W for 16 GB), CPU energy within 8%. loadcur = "codecarbon" reproduces its power-from-load curves, so what remains of the difference is the load measurement itself (codecarbon samples the load during the run; this function integrates the CPU time).

Emissions are $E\times$ the carbon intensity of the location (co2int: a bundled table of every country at its latest published year and of sub-national zones, so no network access is needed anywhere). With location = NULL the location is detected from the environment, the time zone or the locale

(dtctloc) and the detection is recorded. Every default that stood in for a measurement is listed in assumptions, so a report can say what was measured and what was assumed.

The energy figures include the PUE factor (CPU, memory and total alike). The running time has a floor of the CPU time divided by the cores, so a computation faster than the clock's resolution does not come out at zero. Short computations (well under a second) give figures dominated by timer resolution; measure a representative workload.

Every argument is checked, and the carbon intensity looked up, before expr is evaluated, so a bad argument never costs the computation. Errors: an unknown loc; pue below 1 or not a single finite number; cpupwrw or cores not a single positive number (cores also whole); memgb or mempwrwpergb negative; usage other than "process", "machine" or one number in $[0, 1]$; method = "rapl" where the counters are not readable. An error inside expr propagates unchanged. Nothing is written and nothing is fetched.

Value

A list of class rmbl_footprint with elements value (the result of expr), durs (numeric, wall seconds), cputms (numeric, CPU seconds of this process and its children), usage (numeric in $[0, 1]$, the utilisation), usagemode ("process", "machine" or "fixed"), cores (integer), memgb (numeric), enekwh (a list of numerics cpu, memory and total, in kWh, PUE included), co2eg (numeric, grams of CO2-equivalent), co2int (the list returned by co2int, with its year and source), method ("rapl" or "model", the method actually used), assumptions (character vector of every default or fallback that stood in for a measurement) and citations (character vector). The print method shows a three-line summary.

References

Lannelongue, L., Grealey, J. and Inouye, M. (2021). Green Algorithms: quantifying the carbon footprint of computation. Advanced Science 8(12), 2100707. Courty, B. et al. (2024). CodeCarbon: estimate and track carbon emissions from machine learning computing. DOI 10.5281/zenodo.4658424. Lacoste, A., Luccioni, A., Schmidt, V. and Dandres, T. (2019). Quantifying the carbon emissions of machine learning. arXiv:1910.09700.

Examples

```
fp <- co2fp(sum(sqrt(seq_len(2e5))), loc = "CA-ON")
fp$enekwh$total
fp$co2eg
fp$co2int$year
fp$assumptions
print(fp)
# wherever you are, offline: the location is detected and recorded
co2fp(1 + 1, cpupwrw = 45, memgb = 8)$assumptions
# the whole machine's utilisation, on codecarbon's machine curve
co2fp(
  1 + 1,
  loc = "FR",
  usage = "machine",
  loadcur = "codecarbon",
  cpupwrw = 45,
  memgb = 8
```

```
)$usagemode
# a fixed utilisation and a data-centre PUE, with a carbon intensity given as
# a number
fp2 <- co2fp(
  sum(sqrt(seq_len(2e5))),
  loc = 30,
  usage = 0.5,
  cpupwrw = 65,
  memgb = 16,
  pue = 1.67,
  meth = "model"
)
fp2$enekwh
fpeqv(fp2$co2eg)
# a bad argument is refused before the expression runs
try(co2fp(Sys.sleep(10), pue = 0.9))
```

`co2int` *Carbon intensity of electricity by location, offline, anywhere*

Description

The grams of CO2-equivalent emitted per kilowatt-hour of electricity at a location, with the year and source of the figure. The table is bundled with the package, so it works offline in any country: every country at the latest year published by Our World in Data from Ember's yearly electricity data (the latest year each grid reports: 2025 or 2024 for nearly all, older for a few grids, Ukraine (2022) among them, named in `year`), sub-national zones from Electricity Maps' 2024 yearly data as redistributed in the Green Algorithms data release v3.1 (Canadian provinces and territories, US balancing authorities, Australian states, Indian and Japanese regions, Brazilian and Chilean systems, ...), and the world average. All figures are lifecycle intensities (generation plus upstream), the basis both sources publish.

Usage

```
co2int(loc = "WORLD")

co2inttab()
```

Arguments

`loc`	A location code, alpha-3 code or name; a number in gCO2e/kWh, returned as a user-supplied value; or `NULL` to detect the location from the environment, time zone or locale.

Details

A location is matched by its code (`"CA"`, `"CA-ON"`, `"US-CAL-CISO"`), its ISO 3166-1 alpha-3 code (`"CAN"`) or its name (`"Canada"`), case-insensitively. An unknown sub-national code falls back to its country and says so in `note`. `NULL` detects the location with `dtctloc`.

Value

`co2int()`: a list with `g_per_kwh`, `location`, `name`, `year`, `basis`, `source` and `note` (a character vector, empty unless a fallback or detection was involved). `co2inttab()`: the full table as a data frame with the same columns plus `iso3`.

Provenance and licence of the table

The country rows are Ember's yearly electricity data as published by Our World in Data (CC BY 4.0); the sub-national zones and the Canadian provinces come from the Electricity Maps yearly files redistributed in the Green Algorithms data repository (v3.1), under that repository's terms, and the default PUE from the same repository. Every row names its source and year. Values are reproduced unchanged: Ember reports three small diesel grids (Montserrat, Saint Helena, Uzbekistan) at exactly 1000 gCO2e/kWh and Turkmenistan's gas fleet at 1306, and the table does not round or cap them. Kosovo appears once, under the code `XK`.

References

Ember (2026). Yearly Electricity Data, via Our World in Data, ourworldindata.org/grapher/carbon-intensity-electricity. Electricity Maps (2025). 2024 Yearly Carbon Intensity Data, via Green Algorithms data v3.1. Lannelongue, L., Grealey, J. and Inouye, M. (2021). Green Algorithms: quantifying the carbon footprint of computation. Advanced Science 8(12), 2100707.

Examples

```
co2int("CA-ON")
co2int("Canada")$year
co2int("CA-XX")$note
co2int(120)$source
nrow(co2inttab())
# the Canadian provinces in the table, lowest intensity first
tab <- co2inttab()
ca <- tab[
  startsWith(tab$location, "CA-"),
  c("location", "name", "g_per_kwh", "year")
]
head(ca[order(ca$g_per_kwh), ])
# a location the table does not know is an error, not a silent world average
try(co2int("Atlantis"))
```

---

cohtrnd *Raw outcome trends by adoption cohort*

Description

The mean outcome of each adoption cohort by time since adoption, smoothed by a moving average and scaled to 1 in the period before adoption: a check, before any model, that the cohorts moved together before the policy.

Usage

```
cohtrnd(data, unit, time, y, ftreat, win = c(-12, 12), ma = 3L, norm = TRUE)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`ftreat`	Column name of the period a unit is first treated (`NA`, 0 or `Inf` for a unit never treated). Treatment is taken as staying on.
`win`	Periods before and after adoption to show (default -12 to 12).
`ma`	Width of the trailing moving average (default 3; 1 for none).
`norm`	Scale each cohort to 1 at period -1 (default `TRUE`).

Value

A data frame of class `rmbl_cohort_trends`: `cohort`, `evtm`, `value`. `plot()` draws one line per cohort, with the period before adoption marked.

Examples

```
p <- polex()
ct <- cohtrnd(p, "district", "month", "crmrate", "refmonth",
                  win = c(-10, 10))
plot(ct)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
cohtrnd(own, "town", "year", "rate", "first", win = c(-3, 3), ma = 1)
```

commidx *A community-weighted index of protective and risk factors*

Description

Combines published indicators into one index for each group or place, with weights the community sets: protective factors count for, risk factors against. The weights are never estimated.

Usage

```
commidx(
  data,
  grp,
  ind,
  est,
  lower,
  upper,
  w = NULL,
  qual = NULL,
  conflvl = 0.95
)
```

Arguments

`data`	A data frame, one row per group (or place) and indicator.
`grp`, `ind`, `est`, `lower`, `upper`	Column names; every indicator must be on one scale (percent, for example: a rate per 1,000 beside a percentage makes the weights meaningless).
`w`	Named weights, one per indicator used: positive for a protective factor, negative for a risk factor.
`qual`	Optional column of quality flags: `"E"` rows are marked use with caution, and suppressed ones (`"F"`, `"x"`, `".."`) get no estimate.
`conflvl`	Confidence level of each group's index limits (default 0.95).

Details

Inuit Tapiriit Kanatami's National Inuit Suicide Prevention Strategy (2016) names risk factors (among them crowded housing, food insecurity, childhood adversity, mental distress) and protective factors (among them cultural continuity, social equity, family strength, healthy child development). How much each matters is a judgement for the people the index describes, so the function takes the weights as given: positive for protective, negative for risk. The index is $\sum_i w_i p_i / \sum_i |w_i|$, with standard error $\sqrt{\sum_i w_i^2 s_i^2} / \sum_i |w_i|$ from the published intervals (independent estimates). It is a summary of the indicators under the weights given; with other weights it ranks places otherwise, and it is never a ranking of Nations or communities on their culture. For Inuit regions, use Inuit data only.

Value

A data frame of class `rmbl_community_index`: `group`, `index`, `lower`, `upper`, `usewithcau`; attributes `weights` and `conflvl`. `plot()` draws it.

References

Inuit Tapiriit Kanatami (2016). National Inuit Suicide Prevention Strategy. Ottawa. itk.ca/national-inuit-suicide-prevention-strategy

Carroll SR, et al. (2020), see `holbal`.

Examples

```
# weights a community might set for adults, 2019/2022 (table 13-10-0924-01);
# others would differ
h <- indiex("health")
h <- h[h$period == "2019/2022" & h$age == "Total, 18 years and over", ]
w <- c(2, 1, -2, -1)
names(w) <- c(
  "Sense of belonging to local community, somewhat strong or very strong",
  "Has a regular healthcare provider",
  "Perceived life stress, most days quite a bit or extremely stressful",
  "Heavy drinking"
)
ci <- commidx(
  h,
  "group",
  "indicator",
  "estimate",
  "lower",
  "upper",
  qual = "quality",
  w = w
)
ci
# an indicator the table does not publish for the period leaves the index NA,
# never filled: food insecurity is ".." for 2019/2022
w2 <- c(w, "Household food insecure" = -2)
commidx(
  h,
  "group",
  "indicator",
  "estimate",
  "lower",
  "upper",
  qual = "quality",
  w = w2
)$index
plot(ci)

# your own table and the weights a community chose: made-up percentages
own <- data.frame(
```

```
  community = rep(c("Lakeside", "Riverbend"), each = 3),
  item = rep(c("Speaks the language", "Has a doctor", "Food insecure"), 2),
  pct = c(38, 72, 21, 25, 64, 30),
  lo = c(33, 67, 17, 20, 58, 25),
  hi = c(43, 77, 25, 30, 70, 35)
)
commidx(own, "community", "item", "pct", "lo", "hi",
        w = c("Speaks the language" = 2, "Has a doctor" = 1,
              "Food insecure" = -1))
```

`concentration` *Concentration of a total across units*

Description

Measures how unevenly a total is spread across units: whether a few people account for most of the placements, or a few regions for most of the days. `ginicoef()` gives the Gini coefficient, `lorenzc()` the points of the Lorenz curve (for plotting, for example with `plotlrnz()`), and `topshr()` the share of the total held by the largest units ("the top 1% of people account for 20% of the days"). The three are computed in compiled code from the same sorted values.

Usage

```
ginicoef(x, na.rm = TRUE)

lorenzc(x, na.rm = TRUE)

topshr(x, fracs = c(0.01, 0.05, 0.1, 0.25), na.rm = TRUE)
```

Arguments

`x`	Numeric vector of non-negative values, one per unit (for example the days or the count of placements of each person). `NA` values are dropped; `Inf`, negative values and non-numeric input are errors.
`na.rm`	Logical. Missing values are dropped whatever its value; the argument exists so the call reads the same as base R's. Default `TRUE`.
`fracs`	Numeric vector of fractions of the units, each in `[0, 1]`: for each one the share of the total held by that fraction of units, taking the largest units first. Default `c(0.01, 0.05, 0.1, 0.25)`.

Details

Gini is the mean absolute difference between pairs of units over twice the mean, $G = \sum_i \sum_j |x_i - x_j| / (2n^2\bar{x})$, which is also twice the area between the Lorenz curve and the diagonal. Zero is a perfectly even spread; the maximum for `n` units is `1 - 1/n`, not 1, so a Gini near 1 requires many units as well as an uneven spread.

A Gini computed from banded or rounded data inherits that coarsening: compare with `bandsens()` when the values are bands. For a tail that a few units dominate, `hilltlidx()` says whether the largest values follow a power law.

Errors: `x` not numeric, with `Inf` or a negative value, or with no non-missing value; `ginicoef()` when every value is zero (the Gini of a zero total is undefined); `topshr()` when `fractions` is not numeric, is empty, has `NA` or lies outside `[0, 1]`.

Value

`ginicoef()`: a numeric scalar in `[0, 1 - 1/n]`. `lorenzc()`: a data frame with numeric columns `population` (cumulative share of units, from 0 to 1) and `value` (cumulative share of the total, units in ascending order), `n + 1` rows including the origin. `topshr()`: a data frame with columns `fraction` (the requested fraction of units), `units` (how many of the largest units that fraction is) and `share` (the share of the total those units hold). It has class `c("rmbl_gini", "rmbl_measure", "numeric")`, so it prints as a box with its inputs (kept in attribute `"lakner"`); arithmetic, `[`, `c()` and `as.numeric()` give plain numbers.

References

Hedderich, J. and Sachs, L. (2020). *Applied Statistics: Methods Using R*. Springer-Verlag, Berlin Heidelberg. Section 3.14, p. 117, gives the construction used here: the units are placed at equal intervals on the horizontal axis and the cumulated, ascendingly ordered shares of the total on the vertical one, so that the curve is the diagonal exactly when p percent of the units account for p percent of the total, and sags further the greater the concentration.

See Also

`hilltlidx()`, `bandsens()`

Examples

```
# Ten units holding one each: no concentration.
ginicoef(rep(1, 10))

# One unit holding everything: the maximum for ten units.
ginicoef(c(rep(0, 9), 1))
1 - 1 / 10

placements <- c(rep(1, 1200), rep(3, 430), rep(8, 110), rep(20, 38))
ginicoef(placements)

# What the most frequent few account for.
topshr(placements, c(0.01, 0.05, 0.1))

head(lorenzc(placements))

# missing values are dropped
ginicoef(c(1, NA, 3))

# a zero total has no Gini coefficient
try(ginicoef(c(0, 0, 0)))
```

`connassoc` *Connection across places: does one indicator rise where another does?*

Description

For two indicators published for the same places (regions, provinces, Inuit regions), their rank correlation across places, an exact permutation p-value, and an interval that carries the sampling error of the published estimates: for connection to land and wellness, community belonging and mental health, education and health.

Usage

```
connassoc(
  data,
  place,
  x,
  y,
  xlo = NULL,
  xup = NULL,
  ylo = NULL,
  yup = NULL,
  reps = 10000,
  seed = 1,
  conflvl = 0.95
)
```

Arguments

`data`	A data frame, one row per place.
`place`	Column of place names.
`x`, `y`	Columns of the two indicators.
`xlo`, `xup`, `ylo`, `yup`	Optional columns of their published limits (for the interval).
`reps`	Draws for the interval and, past 8 places, for the p-value (default 10,000).
`seed`	Seed for the draws (default 1).
`conflvl`	Confidence level of rho's drawn interval (default 0.95).

Details

Mashford-Pringle and Stewart (2019) describe the land as "an instrumental part of all aspects of Indigenous life including health and well-being"; Mashford-Pringle (2016) describes Indigenous health and education as interconnected. Across places, that connection has a testable shadow: where more people are on the land (or belong, or finish school), do more report wellness? This function

measures it by Spearman's ρ. Its p-value is exact when there are at most 8 places (over all $n!$ orderings) and a Monte Carlo estimate otherwise ($(b+1)/(R+1)$ for b of R = `reps` random orderings at least as extreme). Each estimate carries sampling error, so `rholo` and `rhoup` are the quantiles of ρ over `reps` draws of both indicators from normal distributions with their published standard errors. That is a sensitivity band, not a confidence interval for a correlation across places: it shows how much the published sampling error alone moves ρ for these places, it is pulled toward 0 (noise is added to estimates that are already noisy), and it ignores the sampling of the places themselves. Quality flags are not carried: filter suppressed rows out first.

This is an association across places (an ecological one). It cannot say that connection to land makes a person well, which needs data on persons; a handful of places gives little power; and a correlation is not a measure of the relationship the authors describe. It is a description of how the published indicators move together.

Value

A list of class `rmbl_connection`: `data` (the places used), `rho`, `p_value`, `exact`, `rholo`, `rhoup`, `n`.

Acknowledgement

This estimator puts to quantitative use the work of Professor Angela Mashford-Pringle (Timiskaming First Nation) and Professor Suzanne Stewart (Yellowknives Dene First Nation), both of the University of Toronto.

See the acknowledgements of the `first-nations-metis-inuit-health` vignette.

References

Mashford-Pringle A, Stewart SL (2019). Akiikaa (it is the land): exploring land-based experiences with university students in Ontario. Global Health Promotion 26(3_suppl):64-72. DOI 10.1177/1757975919828722

Mashford-Pringle A (2016). Aboriginal health and Aboriginal education are interconnected. Research in the Sociology of Health Care 34:85-96. DOI 10.1108/s0275-495920160000034005

See Also

`dualcont`.

Examples

```
# across regions, 2022 (table 41-10-0067-01): where First Nations people
# report excellent or very good mental health, do Metis people living in the
# same region too?
mh <- indiex("mentalhealth")
mh <- mh[mh$rating == "Excellent or very good" & mh$region != "Canada", ]
fn <- mh[mh$group == "First Nations", ]
me <- mh[mh$group == "M\u00e9tis", ]
d <- merge(fn, me, by = "region", suffixes = c("fn", "me"))
ca <- connassoc(
  d,
  "region",
  "estimatefn",
```

```
  "estimateme",
  "lowerfn",
  "upperfn",
  "lowerme",
  "upperme"
)
ca
plot(ca)

# your own places: made-up shares in six regions, without limits
own <- data.frame(
  region = c("North", "South", "East", "West", "Coast", "Interior"),
  language = c(41, 18, 27, 33, 22, 38),
  wellbeing = c(68, 55, 60, 66, 57, 63)
)
connassoc(own, "region", "language", "wellbeing", reps = 2000)
```

`cortbl` *Full pairwise correlation table*

Description

Computes the correlation of every pair of numeric columns and returns it in long form – one row per pair, which is easier to sort, filter and join than a matrix – with the number of complete pairs each figure rests on. The counterpart of `corrr::correlate()`.

Usage

```
cortbl(data, meth = c("spearman", "pearson"), minprs = 3L)
```

Arguments

`data`	A data frame; its numeric columns are correlated in pairs.
`meth`	`"spearman"` (default) or `"pearson"`. See `topcors()` for why the rank correlation is the default.
`minprs`	Minimum complete pairs required before a correlation is computed, a whole number of at least 2 (default 3). Below it the pair is `NA` rather than a number computed from almost nothing.

Details

Non-numeric columns are ignored. For each pair, rows where either value is missing are dropped (pairwise deletion), so different pairs can rest on different rows; `nprs` says how many. Spearman's rank correlation (the default) is the Pearson correlation of the ranks, robust to outliers and to monotone transformations; see `topcors()` for why it is the default. A pair with fewer than `minprs` complete rows, or with a constant column (no variance), gets `NA`. The pairs are listed in column order

(`utils::combn()`). A progress bar is drawn on stderr for many pairs unless `options(morie.quiet = TRUE)`.

Errors: `data` not a data frame; fewer than two numeric columns; `minprs` not a whole number of at least 2; an invalid `meth`.

Value

A data frame of class `c("bricklayer_cortable", "data.frame")` with one row per pair of numeric columns and the columns `x` and `y` (character, the column names), `correlation` (numeric, in $[-1, 1]$ or `NA`) and `nprs` (integer); the method is in `attr(, "method")`.

See Also

`topcors()` for just the strongest, `ccovm()` for the matrix form.

Examples

```
set.seed(1)
df <- data.frame(a = stats::rnorm(100), b = stats::rnorm(100))
df$c <- df$a + stats::rnorm(100, sd = 0.2)

cortbl(df)

# Pearson instead of Spearman
cortbl(df, meth = "pearson")

# nprs shows how much data each figure rests on.
gappy <- df
gappy$a[1:80] <- NA
cortbl(gappy)

# A pair with too little overlap is NA, not a number from nothing.
thin <- df
thin$a[1:99] <- NA
cortbl(thin)$correlation

# one numeric column cannot be correlated
try(cortbl(data.frame(a = 1:5, b = letters[1:5])))
```

`cpsatst` *Attest a capsule, binding a signature to what it covers*

Description

`cpsatst()` signs a manifest's canonical digest and records everything a verifier needs alongside it: the scheme, the public key, the context string, the pre-hash, a note and the time of signing. `cpschkatst()` checks the result against a manifest without being told anything further. Use them to publish a capsule together with proof that its manifest has not changed since you signed it.

Usage

```
cpsatst(man, key, ctx = NULL, prehash = "none", note = NULL)

cpschkatst(attest, man, keyexp = NULL, key = NULL)
```

Arguments

man	A manifest, as from mkmnf().
key	For cpsatst(), a signing key from fipskg() (ML-DSA, SLH-DSA and the other standardised schemes) or pqckg() (XMSS), or a shared secret (a string or raw vector) for "hmac". For cpschkatst(), the shared secret that checks an "hmac" attestation; it is not used for the other schemes, whose public key the attestation carries.
ctx	Optional context string (or raw vector), bound into the signature for the standardised schemes; see cpssign(). NULL (the default) means none.
prehash	Pre-hash, as in cpssign(). It is "none" (the default) or a hash name such as "sha256" or "sha512".
note	Optional free text recorded in the attestation – what the signature is meant to assert, in the signer's own words. It is covered by the signature, since it is part of the digest.
attest	An attestation from cpsatst().
keyexp	Optional public key the attestation must carry, as hex or as a key object from fipskg(), fipspubkey() or signpubkey(). Supply it when you know which key should have signed: without it the check confirms the attestation is internally consistent, which any key's holder could arrange.

Details

What this adds over cpssign(). A bare signature leaves three things implicit – which key, which scheme, and which bytes. A verifier who has to be told those out of band cannot check anything they were not already given, which makes the signature a formality. An attestation states them, so the check is cpschkatst(attestation, manifest) and nothing else.

What is signed: the payload digest (mnfdgst() of the manifest), scheme, public, ctx (hex for a raw context), prehash, note and signed_utc is put in canonical form (mnfcanon()) and its SHA-256 is signed with cpssign(). Every field, the note and the time included, is therefore covered: editing any of them breaks the signature. With an XMSS key from pqckg() the scheme is "xmss-sha256" and the context and pre-hash are not used by the signature; signing uses up one of the key's one-time indices, as every XMSS signature does.

With a shared secret (a string or raw vector) as key the scheme is "hmac": HMAC-SHA-256 under that secret, as cpssign(scheme = "hmac"). public is then a key id, the hex of HMAC-SHA-256 of the string "public" under the secret: it names the key, so keyexp can pin it, without revealing it. The secret itself is never recorded, so checking needs it again (key in cpschkatst()). context is "" and prehash "none", the bytes morie's capsule signer writes. HMAC is symmetric: anyone who can check the signature can also make one. It is not one of the post-quantum signature schemes and carries none of their guarantees.

The check, one row each: attestation_complete (every field is present), manifest_digest (the manifest's digest equals the attested one), key_as_expected (only when keyexp is given) and signature (the signature verifies under the attested public key; for "hmac", under key, whose key id must equal public). ok is TRUE when every row passes.

What it does NOT establish. That the public key belongs to whoever you think it does: an attestation is only as good as the channel the key arrived on, which is why keyexp exists. And that the manifest is true – only that it has not changed since it was signed. cpsfals() is for the other question. No chain seal is recorded; to attest a run history, sign chnseal() directly.

Errors: man not a list; key not from fipskg() or pqckg() and not a non-empty string or raw vector; an invalid prehash or ctx (from cpssign()), or either given with an HMAC key; attestation not from cpsatst(). A signature that fails to verify is a failed row, not an error.

Value

cpsatst() a list of class bricklayer_attestation with the character fields digest, scheme, public (hex), context, prehash, note and signed_utc (ISO 8601 UTC time), and signature (a bricklayer_signature from cpssign()).

cpschkatst() a list of class bricklayer_attestation_check with ok (logical) and checks, a data frame with columns check, ok and detail, one row per check.

See Also

cpssign(), mnfdgst(), cpsfals(), chnseal().

Examples

```
m <- mkmnf(list(dataset = "otis", rows = 1200L), env = FALSE)
key <- fipskg("ML-DSA-65")
att <- cpsatst(m, key, note = "counts as published")
att

# a verifier needs the attestation and the manifest, nothing else
res <- cpschkatst(att, m)
res$ok
res$checks

# any change to the manifest breaks it
m2 <- m
m2$meta$rows <- 1201L
cpschkatst(att, m2)$ok

# and so does presenting a different key
cpschkatst(att, m, keyexp = fipskg("ML-DSA-65")$public)$ok

# the note is signed too: editing it breaks the signature row
forged <- att
forged$note <- "counts corrected"
cpschkatst(forged, m)$checks[, c("check", "ok")]

# a context string and a pre-hash
```

```
att2 <- cpsatst(m, key, ctx = "release 2026-10", prehash = "sha512")
cpschkatst(att2, m, keyexp = key)$ok

# a shared secret signs with HMAC; `public` is its key id, not the secret
att3 <- cpsatst(m, "shared-secret")
att3$public
cpschkatst(att3, m, key = "shared-secret", keyexp = att3$public)$ok
cpschkatst(att3, m, key = "another-secret")$checks[, c(
  "check",
  "ok",
  "detail"
)]

# a key that is not a signing key is refused
try(cpsatst(m, 42))
```

---

cpsbndl *Bundle a capsule into one signed artifact*

Description

Records a digest of every file named, the manifest's canonical digest, and an attestation covering both, as a single JSON file. `cpsbndlvrf()` re-hashes the files on disk and checks everything against it.

Usage

```
cpsbndl(
  dir,
  man,
  key,
  files = NULL,
  ctx = NULL,
  prehash = "none",
  note = NULL,
  path = NULL
)

cpsbndlrd(path)

cpsbndlvrf(bundle, dir, man = NULL, keyexp = NULL, key = NULL, seen = NULL)
```

Arguments

`dir`	Character scalar, the directory the capsule lives in. For `cpsbndlvrf()`, the directory to check the files in.

`man` A manifest, as from `mkmnf()`. It is required by `cpsbndl()` and optional for `cpsbndlvrf()`, where it adds a check that the manifest still has the bundled digest.

`key` For `cpsbndl()`, a signing key from `fipskg()` or `pqckg()`, or a shared secret (a string or raw vector) for an `"hmac"` bundle such as morie writes: the attestation then records the key's id, never the secret (see `cpsatst()`). For `cpsbndlvrf()`, the shared secret that checks an `"hmac"` bundle's signature; not used for the other schemes.

`files` Character vector of paths relative to `dir`. Defaults to every regular file in `dir` (recursively, dot-files included), excluding the bundle itself.

`ctx`, `prehash`, `note`
As in `cpsatst()`.

`path` Character scalar. For `cpsbndlrd()`, the bundle file to read. For `cpsbndl()`, where to write the bundle (an existing file is overwritten).

`capsule_bundle.json` inside `dir` is the default for `cpsbndl()`.

`bundle` A bundle read back with `cpsbndlrd()`, or the path to one.

`keyexp` Optional public key hex (for `"hmac"`, the key id) the bundle's attestation must carry. Supply it when you know which key should have signed: without it the check confirms the bundle is internally consistent, which any key's holder could arrange.

`seen` For `cpsbndlvrf()`, the XMSS signatures seen elsewhere, as `xmssidxseen()` returns them, or `NULL`. For an XMSS bundle it adds an `xmss_index` row, failed (`"xmss index reused"`) when the same key signed the bundle's index with another signature: the key's state was copied or rolled back (RFC 10033, section 4).

Details

The file digests are inside the signed payload, not alongside it. That is the whole point: a list of hashes that is not itself signed can be rewritten to match whatever the files now say, and a recipient who re-hashes the files and compares them to that list learns only that the list is consistent with itself.

What it establishes: the named files have not changed since signing, the manifest has not changed, and both were signed together by the holder of that key. What it does not: that the key belongs to anyone in particular, or that the manifest's claims are true.

How it is built. Each file is hashed with SHA-256 (`sha256file()`); the payload is the list of `path`, `sha256` and `size_bytes`, the manifest's `mnfdgst()` and the UTC time; `cpsatst()` signs a manifest whose `meta` is that payload, and the bundle (payload plus attestation) is written as pretty JSON with doubles at 17 digits. With `files = NULL` every regular file under `dir` is hashed, recursively and including dot-files (a `.Rprofile` dropped into a capsule runs code), except the bundle file itself. A progress bar is shown when `options(morie.progress)` allows it.

How it is checked. `cpsbndlvrf()` re-hashes every listed file (a path that is absolute or climbs out of `dir` fails its row), reports every file on disk the bundle does not list (the bundle's own file excepted), checks the attestation with `cpschkatst()` (signature, digest, and `keyexp` when given; an `"hmac"` signature needs the shared secret as `key`), and, when `man` is given, compares its digest with the

bundled one. Every check is a row of checks; ok is TRUE only when all pass. A failed check is a row, not an error.

Errors: dir missing or not a directory; no files to bundle, or a named file not in dir; a key cpsatst() cannot sign with; cpsbndlrd() given a file that is not a bundle; cpsbndlvrf() given something other than a bundle or a path. Reading a bundle whose signature records a stateful key's state warns (see cpsatst()).

Value

cpsbndl(): the bundle, invisibly, a list of class bricklayer_bundle with files (a list of path, sha256, size_bytes), manifest_digest, bundled_utc and attestation (from cpsatst()), and the path it was written to as attribute "path".

cpsbndlrd(): the same structure read from the file.

cpsbndlvrf(): a list of class bricklayer_bundle_check with ok (logical) and checks (a data frame of check, logical ok and detail: one file:PATH row per listed file, no_unlisted_files, the attestation: rows, with a manifest manifest_digest, and with seen xmss_index). For an XMSS bundle the attestation:signature row's detail names the signature's index and the key fingerprint.

See Also

cpsatst(), vrfcps(), mnfdgst().

Examples

```
dir <- tempfile()
dir.create(dir)
write.csv(data.frame(x = 1:3), file.path(dir, "data.csv"),
          row.names = FALSE)
m <- mkmnf(list(dataset = "demo"), env = FALSE)
key <- fipskg("ML-DSA-44")

b <- cpsbndl(dir, m, key, note = "as published")
cpsbndlvrf(attr(b, "path"), dir, man = m)$ok

# touch a byte of the data and the bundle no longer holds
write.csv(data.frame(x = 1:4), file.path(dir, "data.csv"),
          row.names = FALSE)
res <- cpsbndlvrf(attr(b, "path"), dir, man = m)
res$ok
res$checks[!res$checks$ok, c("check", "detail")]

# read back, and checked against the key you expected to have signed
b2 <- cpsbndlrd(attr(b, "path"))
b2
cpsbndlvrf(b2, dir, keyexp = "00ff")$ok

# a file added beside the bundle is reported too
writeLines("x", file.path(dir, "extra.txt"))
cpsbndlvrf(b2, dir)$checks[, c("check", "ok")]
```

```
# a bundle signed with a shared secret (HMAC), as morie signs
dir2 <- tempfile()
dir.create(dir2)
writeLines("x,y", file.path(dir2, "d.csv"))
h <- cpsbndl(dir2, m, "shared-secret")
h$attestation$public  # the key id, not the secret
cpsbndlvrf(attr(h, "path"), dir2, key = "shared-secret",
           keyexp = h$attestation$public)$ok
# without the secret the signature row cannot pass
cpsbndlvrf(attr(h, "path"), dir2)$ok
unlink(dir2, recursive = TRUE)

# a directory that does not exist
try(cpsbndl(file.path(tempdir(), "no-such-capsule"), m, key))
unlink(dir, recursive = TRUE)
```

---

cpschk *Check every capsule under a folder*

Description

Runs the checks of the rmbl GitHub Action on every capsule (a folder holding `data_provenance.json` or `capsule_bundle.json`) under `path`: each capsule file against its JSON Schema (`schemapath`), then `vrfcps` and `cpsbndlvrf`. With `mode = "drift"` or `"refresh"` each source is fetched again and compared with the pin, column by column (`cpsdrft` and `benfordtst`); `"refresh"` then writes and pins new bytes. `rmbl verify`, `drift` and `report` call it. `format()` renders the Markdown report or the JSON, and `ghann`, `ghcmt` and `ghout` the GitHub pieces.

Usage

```
cpschk(
  path = ".",
  mode = c("verify", "drift", "refresh", "report"),
  key = "",
  pubkey = "",
  psiwarn = 0.1,
  psifail = 0.25,
  base = NULL,
  cache = NULL,
  allowfile = FALSE,
  snapshot = FALSE,
  wayback = FALSE,
  glob = "",
  fp = TRUE,
  seenlog = "",
  archive = FALSE
```

```
)

## S3 method for class 'rmbl_cpschk'
format(x, fmt = c("md", "json", "text"), failon = NULL, ...)

## S3 method for class 'rmbl_cpschk'
print(x, ...)
```

Arguments

path	Character scalar, the folder searched recursively (`.git` and `.rmbl-cache` folders skipped).
mode	`"verify"` (the default), `"drift"`, `"refresh"` or `"report"` (the verify checks).
key	Character scalar: the shared secret of HMAC bundles, or `""`.
pubkey	Character scalar: the public key (hex) every bundle must be signed with (for an HMAC bundle, its key id), or `""`. Given, a capsule with no signed bundle fails: there is nothing the key checks.
psiwarn, psifail	Non-negative numbers, `psiwarn` at most `psifail`: the PSI bands, by default 0.1 and 0.25, the bands `drftpsi` documents.
base	`NULL` (no comparison); the path of one manifest file each manifest is compared with; a folder holding the base copy of the tree under `path` (the same relative paths); or a function that takes a manifest's path and returns the path of its base copy, or `NULL` when the base has none (for a git ref, write what `git show` prints to a file). Its `"label"` attribute names it in the report.
cache	`NULL` (the default: the user cache folder, `tools::R_user_dir("rmoriebricklayer", "cache")`, in an interactive session, none otherwise), `FALSE` (none) or a folder.
allowfile	`FALSE` (the default), `TRUE` to accept `file://` sources and local files anywhere under `path`, or an existing folder whose files may be sources as well. Without it a local source must be a file inside its own capsule; a fetched one is `https://` only (see `cpsrfrsh`).
snapshot	Logical: look up the latest Wayback snapshot of each http(s) source, a `snapshot` row (a note).
wayback	Logical: with `"refresh"`, capture each changed http(s) source in the Wayback Machine first (archive.org keys, as `wbsave()` reads them).
glob	Character scalar: check only the capsule files whose path matches this glob (`*` also matches `/`), or `""`.
fp	Logical: measure the footprint of the checks (`co2fp`) for the last lines of the report.
seenlog	Character scalar: an XMSS seen-log (`xmssidxseen`; `rmbl verify --seen-log`) read with the bundles under `path`, or `""`. Every XMSS bundle gets an `xmss_index` row, failed when the same key signed its index with another signature in what was seen.

archive	Logical: use archive.rmorie.com when a source cannot be fetched. In "drift" and "refresh", a source that neither answers nor has its pinned Wayback snapshot is fetched from a copy kept there with the pinned SHA-256 (public copies need no key); with wayback = TRUE, a changed source the Wayback Machine refuses is saved there instead (wbsave(fallback = "rmorie"), which needs a key: dlogin() or MORIE_HOSTED_KEY).
x	An rmbl_cpschk result.
fmt	"md" (the Markdown report), "json" or "text" (the console summary print() draws as a box).
failon	Character scalar or NULL: the fail-on level, named in the report's last line.
...	Further arguments of format() and print(), ignored.

Details

Each check is one row with a severity: error, drift, warn or note (a note never changes a result). A capsule's result is the worst severity among its failed rows, and status the worst over all capsules: ok, warn, drift or error. Each failed row adds a note, and one more from explerr where it finds what the row means or where its message is raised.

Drift: when the fetched source's SHA-256 differs from the pin and the pinned data file is a CSV, cpsdrft() runs with psithr = psifail. A column is drift when cpsdrft() flags it and its PSI is above psifail, and warn when it is flagged below that or its PSI is above psiwarn; an added or removed column is drift. New bytes whose columns all stay within the bands are a note; new bytes that cannot be compared are a warning.

Refresh: the new bytes are written as the data file and pinned in the provenance as cpsrfrsh does (refused when they fail the schema the provenance pins, valsch); with wayback = TRUE an http(s) source is first captured with wbsave. The capsule is then verified again.

Manifests: with base, each capsule's manifest.json and results/manifest.json is compared with its base copy: the base's recorded results against this copy's observed values, each within the base's tolerance (mnfrcmp), and the resources of the two (mnfdiff). A manifest only one side has is added or removed.

Cache: a fetched source is kept in the cache folder under source- and the first 16 hex digits of the SHA-256 of its URL, one file per source overwritten by each fetch, and is read only when the live fetch fails.

Side effects: "drift" and "refresh" fetch each source (the network for http(s) ones), as snapshot = TRUE asks the Wayback Machine once per capsule (wbsnpurl); "refresh" writes the capsules; a cache folder is written to. Errors: path is not a directory; a bad argument. A capsule whose checks stop is one failed row, not an error.

Value

A list of class rmbl_cpschk: status, mode, path, glob, rows (a data frame of every check: capsule, part, file, check, ok, detail, severity and line, the line of the file where known), capsules (a list per capsule: capsule, source, pinned_sha256, fetched_sha256, changed, result, checks_passed, checks_total and notes, and where made drift with its columns table, drift_summary, snapshot, snapshot_utc and manifests), base (its label), psiwarn, psifail

and `footprint` (the `co2fp()` result without its value, or `NULL`). The `cpsdrft()` result of each capsule whose columns were compared, with its provenance and manifest diff, is in the attribute `drifts` (named by capsule), for `drfthtml` and `drftapp`. `format()` returns one string (`"md"`, `"json"`) or the lines of the console summary (`"text"`); `print()` draws that summary as a box.

See Also

`blcli` (the verbs), `ghann`, `cpsrfrsh`, `explerr`.

Examples

```
root <- tempfile("capsules-")
d <- usecpstmpl(file.path(root, "one"), ex = TRUE)
invisible(cpsrfrsh(d))
x <- cpschk(root, fp = FALSE)
x$status
x$rows[, c("check", "ok")]
x

# the data changed: the digest row fails, and the notes say what it means
p <- loadprov(file.path(d, "data_provenance.json"))
cat("1,2\n", file = file.path(d, p$resource$filename), append = TRUE)
x <- cpschk(root, fp = FALSE)
x$status
x$capsules[[1]]$notes
substr(format(x, fmt = "json"), 1, 120)

# drift: fetch the source again (here a local file) and compare
cpschk(root, "drift", psiwarn = 0.05, fp = FALSE)$status

try(cpschk(tempfile()))
try(cpschk(root, psiwarn = 1))
```

`cpsdrft` *Compare a fetched data frame with the one a capsule was pinned against*

Description

Checks whether the data in a freshly downloaded table still look like the data an analysis was built on, column by column. A digest only says whether the bytes changed; this says whether the distributions changed, which is what invalidates a result. Use it when re-running an analysis on a re-released extract.

Usage

```
cpsdrft(
  ref,
```

```
  cur,
  alpha = 0.01,
  psithr = 0.25,
  psiminn = 1000L,
  bins = 10L,
  idelvls = 100L,
  seed = 1L
)
```

Arguments

ref	Data frame the capsule was built from.
cur	Data frame just fetched.
alpha	Significance level for the drifted flag (default 0.01; deliberately stricter than 0.05 because a wide table runs many tests).
psithr	PSI above which a numeric column is flagged even when its p-value is not significant (default 0.25, the conventional "material shift" line).
psiminn	Minimum size BOTH samples must reach before psithr is allowed to flag a column on its own (default 1000). The PSI bands are large-sample heuristics with no calibrated null distribution: on a few hundred rows, binning noise alone routinely pushes the index past 0.25, so applying the threshold there manufactures drift. Below this size the flag rests on the Kolmogorov-Smirnov p-value, which is calibrated, and the PSI is still reported as an effect size.
bins	Bins passed to drftpsi() (default 10).
idelvls	A character column with more distinct values than this (or at least 20 distinct values covering more than a fifth of the rows) is an identifier, not a distribution: it gets type = "identifier", its unseen_share (the fraction of current rows whose value the reference never had) and drifted = NA. Character columns that parse as dates are compared as dates (type = "date").
seed	Seed for the Monte Carlo branch of the homogeneity test, drfthomog(). The caller's RNG stream is left untouched.

Details

Runs the appropriate drift test on every shared column and collects the verdicts in one report: drftks() plus drftpsi() for a numeric column, drfthomog() for a categorical one. Columns that appeared or vanished are listed separately, since no test applies to them.

The categorical test is the two-sample homogeneity test, NOT drftchisq() 's goodness-of-fit against a known distribution: the reference here is itself a finite sample, and ignoring its sampling error would report drift too readily.

This is the check that a byte-level digest cannot make. A re-released extract legitimately has a different SHA-256 while being the same data statistically; conversely a column can keep its name, type and row count while having been silently rescaled. cpsdrft() asks whether the DATA moved.

Column types: a column numeric on both sides is "numeric"; a character or factor column whose every value is an ISO date (YYYY-MM-DD) or date-time on both sides is converted and tested as "date"; a categorical column with more than idelvls distinct values, or with at least 20 distinct

values that number more than a fifth of the rows, is an "identifier" and is not tested; anything else is "categorical". Missing values are dropped. For a categorical column the PSI is computed over the reference's categories with every unseen current value pooled into one extra bin.

A column is drifted when its p-value is below alpha, or when its PSI exceeds psithr and both samples have at least psiminn values. anydrft is also TRUE when a column was added or removed. Each column runs one or two tests, so with many columns some will fall below alpha by chance; see scanadj() for a multiplicity correction and drftcal() for an empirical false-alarm rate. A progress bar is shown in an interactive session for many columns.

Errors: either argument not a data frame; alpha not strictly between 0 and 1; psiminn negative; idelvls below 2.

Value

A list of class bricklayer_drift with elements:

columns a data frame, one row per shared column, with column, type ("numeric", "date", "categorical" or "identifier"), statistic (KS D or chi-square), p_value, psi, jsdiv, unseen_share (share of current values the reference never had; categorical and identifier columns only), drifted (logical, NA when no test applies) and note (why a column was not tested, or "");

added, removed character vectors of column names;

nref, ncur row counts;

alpha, psithr, psiminn the settings used;

bands the PSI bands the plots and reports colour by: warn (0.1, or psithr when lower) and fail (psithr);

summ one binned summary per shared column, for plot(), drfthtml() and drftapp(): missing shares, sizes, summary statistics or the ten largest categories, the PSI bins with each bin's term of the index, a histogram and first-digit counts. Never the rows;

anydrft logical.

The print method lists the per-column verdicts, drifted columns first.

See Also

valsch() for the structural check, which this complements rather than replaces.

Examples

```
set.seed(5)
ref <- data.frame(
  value = stats::rnorm(300),
  size = stats::runif(300, 1, 10),
  grade = sample(c("a", "b", "c"), 300, TRUE)
)

# A fresh draw from the same process: no drift.
same <- data.frame(
  value = stats::rnorm(300),
```

```
  size = stats::runif(300, 1, 10),
  grade = sample(c("a", "b", "c"), 300, TRUE)
)
d <- cpsdrft(ref, same)
d$anydrft

# A silently rescaled column, and a new category, are both caught.
moved <- same
moved$size <- moved$size * 3
moved$grade[1:100] <- "z"
cpsdrft(ref, moved)

# The PSI is always reported as an effect size, but on a sample this
# small it is not allowed to raise the flag by itself -- binning noise
# alone would clear 0.25. Lower psiminn to override that.
cpsdrft(ref, same)$columns$psi

# Structural changes are reported rather than tested.
cpsdrft(ref, same[, c("value", "grade")])$removed

try(cpsdrft(ref, as.matrix(same)))
```

`cpsfals` *Run falsification controls against a statistic*

Description

Recomputes `stat` under conditions in which its value is known in advance, and reports whether it behaved. Four controls, each answering a different way of being wrong:

Usage

```
cpsfals(data, stat, trt = NULL, n = 199L, subfrac = 0.8, seed = NULL)
```

Arguments

`data`	A data frame that the statistic `stat` is computed on.
`stat`	A function of one data frame returning a single finite number.
`trt`	Name of the column the finding is about, permuted for the permutation and placebo controls. Optional: without it those two controls are skipped and reported as such rather than silently omitted.
`n`	Integer scalar, the number of permutations and of subsets, at least 9; default 199. The permutation p-value cannot be smaller than `1 / (n + 1)`, which is reported.
`subfrac`	Numeric scalar strictly between 0 and 1, the fraction of rows kept by the subset control; default 0.8.
`seed`	Optional integer seed. Supplied, the result is reproducible; omitted (`NULL`, the default), the current random number stream is used and only the generator kind is recorded.

Details

- **permutation** – shuffle `trt` and the association it carries is destroyed, so the statistic should fall to its null distribution. Reports where the observed value sits in that distribution, and the smallest p-value the number of permutations could have produced.
- **random common cause** – add a column of noise. It cannot possibly matter, so a statistic that moves is reading something other than the data.
- **placebo treatment** – replace `trt` with a random draw of the same shape. The effect should vanish; if it does not, the statistic is picking up structure that has nothing to do with the exposure.
- **subset stability** – recompute on random subsets. Wide spread means the finding rests on particular rows, which is worth knowing before it rests on a conclusion.

What a pass means. Only that these controls did not catch anything. They are falsification tests, so they can refute and cannot confirm: passing all four is consistent with a statistic that is wrong for a reason none of them probes.

When each control passes:

- permutation – the two-sided permutation p-value $(1 + \#\{|s_b| \geq |s_{obs}|\})/(1 + B)$ over the B usable permutations is below 0.05 (the observed statistic is unlike its null);
- placebo – the statistic under one more permuted treatment lies within the 95th percentile of the absolute permutation values;
- random common cause – adding a column `.rmbl_random_common_cause` of standard normal noise changes the statistic by at most $10^{-8} \max(1, |s_{obs}|)$;
- subset stability – the observed value lies inside the middle 95% of the statistic over `n` random subsets of `round(subfrac * nrow(data))` rows (at least two).

A permutation or subset on which the statistic errors or is not finite is dropped from that control. The statistic is evaluated `2 n + 3` times, so its cost sets the run time; a progress bar is shown when `options(morie.progress)` allows it. With `seed` the random number stream is set for the call and the caller's stream restored afterwards.

Errors: `data` not a data frame or with fewer than 4 rows; `stat` not a function, or failing or not returning one finite number on the data as given, the placebo data or the noise-augmented data; `n` below 9; `subfrac` not strictly between 0 and 1; `trt` not a column of `data`.

Value

A list of class `bricklayer_falsification`: `observed` (the statistic on the data as given), `controls` (a data frame with one row per control: `control`, logical `passed` (`NA` when skipped), numeric `value` (the p-value, the placebo or noise statistic, or the subset spread) and `detail`), `permutation` (the vector of permuted statistics, `NULL` without `treatment`), `n`, `treatment`, `rng_kind` and `seed` (`NA` when none was given). `print()` shows one line per control; `plot()` draws the permutation distribution.

See Also

`cpspwr()` for the positive control – whether a real effect would have been seen at all; `cpsatst()` for the different question of whether the record is intact rather than whether the finding survives.

Examples

```
set.seed(1)
d <- data.frame(x = rnorm(200))
d$y <- 0.8 * d$x + rnorm(200)
# a real association survives its controls
real <- cpsfals(d, function(z) cor(z$x, z$y),
                        trt = "x", n = 199, seed = 42)
real$controls[, c("control", "passed")]

# and a statistic that ignores the data fails the ones that can see it
fake <- cpsfals(d, function(z) 0.5, trt = "x",
                        n = 199, seed = 42)
fake$controls[fake$controls$control == "permutation", "passed"]

# without a treatment column two controls are skipped, and say so
fit <- cpsfals(d, function(z) mean(z$y), n = 49, seed = 1)
fit$controls[, c("control", "passed")]

# the statistic must return one finite number
try(cpsfals(d, function(z) c(1, 2), n = 49))
```

---

cpsplant *Does the Analysis Recover a Known Effect?*

Description

Plants effects of known size into the real data and checks that the statistic gets them back. Each planted data set keeps every column as it is, except the outcome: it is shuffled, which removes whatever real association it had with the treatment, and then the planted effect is added, `effect * treatment`. A statistic that estimates the treatment's effect on the outcome should return about `effect` on every one.

Usage

```
cpsplant(
  data,
  stat,
  y,
  trt,
  effects = c(0, 0.5, 1),
  n = 100L,
  tol = NULL,
  seed = NULL
)
```

Arguments

data	A data frame with at least 4 rows.
stat	A function of one data frame returning one number: the estimated effect of trt on y.
y	The outcome column's name. It must be numeric.
trt	The treatment column's name.
effects	The effects to plant. Include 0, so a statistic that finds an effect where there is none fails.
n	Planted data sets per effect.
tol	How far an average estimate may sit from the planted effect. The default is 10% of the largest planted effect.
seed	Optional seed, for a reproducible check. The session's own random stream is left as it was.

Details

This is evidence about the statistic as it is applied to this data (its missing values, its skew, its group sizes), which a simulation on made-up data cannot give. A statistic that recovers 0, 0.5 and 1 correctly is very unlikely to be computing the wrong thing.

The treatment column must be numeric, logical or a factor with two levels (the second is coded 1). Effects are on the outcome's own scale per unit of treatment, so the statistic must estimate an additive effect: a difference in means, a regression slope, an average treatment effect. An effect is recovered when the average estimate is within tol of it, after allowing one Monte Carlo standard error, and that standard error is itself no larger than half of tol: a statistic too noisy for the check to mean anything is not recovered (raise n). The allowance is so at most 1.5 times tol. Without the second condition the allowance grew with the noise, and a statistic that returned pure noise passed; with up to four times tol allowed, an estimator that returned 0.7 of the effect with some noise passed.

Value

A list of class rmbl_plant: ok (every effect recovered), table (one row per effect: effect, recovered (the average estimate), bias, se, passed), observed (the statistic on the real data), n, tol, y, trt and seed. It prints as a box and goes into a manifest with recevid().

See Also

cpsspec() for agreement across specifications, cpsfals() for falsification controls, recevid() to record them all.

Examples

```
set.seed(1)
d <- data.frame(x = rbinom(200, 1, 0.5), z = rnorm(200))
d$y <- 2 * d$x + d$z + rnorm(200)

# A difference in means recovers what is planted.
```

```
dm <- function(df) mean(df$y[df$x == 1]) - mean(df$y[df$x == 0])
cpsplant(d, dm, y = "y", trt = "x", n = 30, seed = 1)

# A statistic with a bug (it doubles the effect) does not.
bad <- function(df) 2 * dm(df)
cpsplant(d, bad, y = "y", trt = "x", n = 30, seed = 1)$ok
```

`cpspwr` *Detect an injected effect of known size*

Description

Adds an effect of each given size to the data, reruns the whole detection procedure, and reports how often it was found. The result is a power curve, and the smallest size detected reliably is the smallest effect the analysis could have seen. Use it before reporting a null result, to show whether the analysis could have found an effect of the size that matters.

Usage

```
cpspwr(
  data,
  stat,
  inject,
  sizes = c(0, 0.2, 0.5),
  trt = NULL,
  n = 99L,
  reps = 10L,
  alpha = 0.05,
  seed = NULL
)
```

Arguments

`data`	A data frame, passed to `stat` and `inject`.
`stat`	A function of one data frame returning a single finite number (for example a correlation or a difference in means).
`inject`	A function of `(data, size)` returning the data with an effect of that size added. It is the caller's, because what counts as an effect of size 0.2 is a modelling decision and not something this function can guess.
`sizes`	Numeric vector of effect sizes to try (default `c(0, 0.2, 0.5)`). `0` is added if absent, since the detection rate there is the false-positive rate and belongs on the same curve. Duplicates are removed and the sizes sorted.
`trt`	Name of the column permuted to build the null, as in `cpsfals()`. Required.
`n`	Permutations per test, a whole number of at least 9 (default 99).

reps	Repetitions per size, a whole number of at least 1 (default 10). The detection rate at each size is out of this many, so its resolution is 1 / reps.
alpha	Significance threshold for counting a detection (default 0.05).
seed	Optional integer seed; NULL (the default) uses the session's random number state.

Details

This is the positive control that cpsfals() lacks: its four are all negative. Those establish that the finding is not an artefact; this establishes that a real effect would not have been missed. A study that passes every falsification control and has no power against the effect it was looking for has not found that the effect is absent – it has found nothing either way, and the two are routinely reported as the same thing.

For each size (0 is always added) and each of reps repetitions, the statistic is computed on the data passed through the injector, as inject(data, size), and a permutation null is built by shuffling the trt column n times and recomputing the statistic. The two-sided permutation p-value is

$$p = \frac{1 + \#\{|T^*| \geq |T|\}}{1 + n_{ok}}$$

over the n_{ok} permutations with a finite statistic (Phipson and Smyth's correction, so p is never 0), and the effect counts as detected when $p \leq \alpha$. A repetition whose injection or observed statistic fails (error or non-finite value) is skipped and not counted in reps. Since the data are the same in every repetition, inject should add randomness of its own (new noise, a resampled effect) when repetitions are meant to differ; otherwise only the permutations vary.

The cost is $|sizes| \times reps \times (n + 1)$ evaluations of stat. A progress bar is drawn on stderr unless options(morie.quiet = TRUE). With seed, R's random number state is set for the run and restored afterwards; without it, the run uses and advances the session's generator.

Errors: data not a data frame of at least 4 rows; stat or inject not a function; trt missing or not a column; n below 9; reps below 1; n so small that $1/(n + 1) > \alpha$, so that nothing could ever be detected.

Value

A list of class c("bricklayer_power", "list"):

curve data frame with one row per size: size, detected (integer), reps (integer, the usable repetitions) and rate (detected / reps, NA when none was usable).

alpha, n, reps, treatment the settings used.

smadet the smallest positive size found at a rate of at least 0.8, or NA when no size reached it.

false_positive_rate the rate at size 0.

See Also

cpsfals() for the negative controls.

Examples

```
set.seed(1)
d <- data.frame(x = rnorm(120), y = rnorm(120))
# inject a linear effect of x on y
inj <- function(z, size) {
  z$y <- z$y + size * z$x
  z
}
pw <- cpspwr(d, function(z) cor(z$x, z$y), inj,
                    sizes = c(0, 0.3, 0.6), trt = "x",
                    n = 99, reps = 5, seed = 42)
pw$curve
pw

# The rate at size 0 estimates the false-positive rate. With few
# repetitions it is usually 0, because alpha is small -- five draws
# at 0.05 come up empty about three times in four -- so read it as a
# sanity check that it is not LARGE, not as an estimate of alpha.
pw$curve[pw$curve$size == 0, "rate"]

# an injection with fresh noise each time, and effects too small to be
# detected in four repetitions: the smallest detected size is NA
inj2 <- function(z, size) {
  z$y <- z$y + size * z$x + stats::rnorm(nrow(z), sd = 0.5)
  z
}
cpspwr(d, function(z) cor(z$x, z$y), inj2, sizes = c(0.1, 0.2),
              trt = "x", n = 49, reps = 4, seed = 7)$smadet

# too few permutations to ever reach alpha = 0.05
try(cpspwr(d, function(z) cor(z$x, z$y), inj, trt = "x", n = 9))
```

`cpsrfrsh` *Fetch a capsule's source again and pin the new bytes*

Description

Downloads the source named in a capsule's `data_provenance.json` (`resource$direct_url`, else `source$url`; an `https://` URL, or a local file inside the capsule), and when its bytes differ from the pin writes them as the capsule's data file and pins them in the provenance, in the form the `analysis.R` of `usecpstmpl` writes on its first run: `source$sha256`, and `resource$filename`, `sha256`, `size_bytes` and `retrieved_utc` (`row_count_data_rows` too when it was recorded). `vrfcps` then verifies the capsule against the new data. From the shell: `rmbl refresh PATH`.

Usage

```
cpsrfrsh(dir, allowfile = FALSE)
```

Arguments

dir	Character scalar, the capsule directory.
allowfile	FALSE (the default), TRUE to accept a file:// source inside dir, or an existing folder whose files may also be read as a source (a local path or a file:// URL).

Details

Nothing is written when the bytes are unchanged. A download goes through fdl, with the provenance's wayback$csv_snapshot as its fallback (no other Wayback copy is looked up). A source is fetched over https:// only. A local path (relative to the capsule) or a file:// URL is read only when it resolves, symbolic links followed, to a file inside the capsule, inside the allowfile folder, or in the package's extdata; never under /proc, /sys or /dev. A gzip source may inflate to at most the option rmoriebricklayer.maxcsvbytes (1 GiB by default) when it is read. A capsule bundle is not signed again: that needs your key (cpsbndl). Side effects: writes the data file and data_provenance.json in dir. Errors: dir is not a directory or holds no provenance; the provenance names no source, or a TODO one, or a file:// one without allowfile; a URL that is not https://; a local source outside those folders; the source cannot be fetched, or inflates past the limit; resource$filename is not a plain relative path inside dir; data_provenance.json is a link out of dir; the new data fail the schema the provenance pins (valsch).

Value

The mnfdiff of the provenance before and after: all "unchanged" when the source still matches its pin.

See Also

mnfdiff, vrfcps, usecpstmpl.

Examples

```
d <- usecpstmpl(tempfile("capsule-"), ex = TRUE)
cpsrfrsh(d)            # the first pin: the data file is added
vrfcps(d)$ok
cpsrfrsh(d)$change     # the source has not changed since

try(cpsrfrsh(tempfile()))
```

---

cpsrpt *Assess a capsule in one call*

Description

Runs the checks this package provides over one data frame and collects the findings into a single report: the structural schema check, the distributional comparison against a reference, the missingness picture, the multivariate outliers, a Benford screen on the wide numeric columns, and the integrity digests. Use it as the first look at a capsule or a fresh extract, before any analysis, and keep the Markdown version (rptmd()) with the outputs.

Usage

```
cpsrpt(
  data,
  ref = NULL,
  schema = NULL,
  rules = NULL,
  chunks = NULL,
  sig = NULL,
  key = NULL,
  alpha = 0.01,
  maxrowsoutl = 20000L
)
```

Arguments

data	The data frame to assess.
ref	Optional data frame the capsule was pinned against. Supplying it enables the drift comparison, which is the check a digest cannot make.
schema	Optional schema from infsch(), or a provenance list containing one. Supplying it enables the structural check.
rules	Optional list of rul() objects.
chunks	Optional character vector of capsule chunks (see chnkfile()) to pin with a Merkle root.
sig, key	Optional signature and verifying key, as from cpssign(), checked against the data's own digest.
alpha	Numeric scalar, the significance level passed to cpsdrft(); default 0.01.
maxrowsoutl	Integer scalar: skip the outlier scan above this many rows (default 20000), since it factors a covariance per call.

Details

Nothing here is new arithmetic. The value is that the answers arrive together and ordered by severity, because the failure mode this is built against is a person running one check, seeing it pass, and concluding the data is fine.

Value

A list of class c("bricklayer_report", "list"): findings (a data frame of character columns severity, check ("schema", "rule", "drift", "missing", "shape", "outliers", "benford" or "signature"), subject (the column, rule or row count concerned) and detail), verdict (the worst severity: "fatal", "warn", "note", or "clean" when there is no finding), profile (from profcols()), missingness (from misssmry()), drift (the cpsdrft() result, or NULL without a reference), digest (a list with data_digest, and with chunks also merkle_root, merkle_scheme (the mrkroot() hashing scheme, 2) and n_chunks, and, with a signature and key, the logical signature_valid), and nrows / n_cols. print() shows the verdict and the findings; summary() gives the verdict and the count of each severity.

What `severity` means

- `fatal` – nothing downstream can run as it stands: a schema check or a rule of severity `"fatal"` failed (for example a required column is missing), or a column of the reference is absent from the data.
- `warn` – something moved or is badly incomplete: a schema check, or a rule of severity `"warning"`, failed (a value left its pinned range, a type changed), a column drifted from the reference, a column is entirely missing or more than half missing, or a signature did not verify (or came with no key).
- `note` – worth a look but not necessarily wrong: a column not in the reference, a column whose drift could not be tested, a contiguous run of missing values (one outage rather than scattered gaps), a constant column, values beyond the Tukey fences, multivariate outliers or collinear columns, a Benford departure.

How each check runs

The drift comparison is `cpsdrft()` at level `alpha`. The missing run is the longest run from `missruns()` of at least `max(2, nrow(data) %/% 20)` rows. The outlier scan is `mahaoutl()` on the numeric columns that are not constant or entirely missing, run only when there are at least two of them, at most `maxrowsoutl` rows and more complete rows than columns. The Benford screen (`benfordtst()`) runs on a numeric column with at least 200 non-zero finite values whose largest absolute value is more than 1,000 times its smallest, and reports p below 0.01. The digest is `dgstobj()` of the data; a signature is checked with `cpsvrf()` against that digest.

Findings are sorted by severity, then check, then subject. Nothing is written and nothing is fetched. Errors when `data` is not a data frame or has no columns, and when `schema` or `rules` are malformed (from `valsch()` and `valrules()`).

A clean report is not proof the data is correct. It means these particular checks found nothing, and every one of them has a stated blind spot – see `cpsdrft()` on statistical power and `benfordtst()` on why a departure is a screen rather than a verdict.

See Also

`cpsdrft()`, `profcols()`, `valsch()`, `rptmd()` to write it out.

Examples

```
set.seed(1)
ref <- data.frame(
  id = 1:200,
  score = stats::runif(200, 0, 10),
  grade = sample(c("a", "b", "c"), 200, TRUE),
  stringsAsFactors = FALSE
)

# A fresh extract from the same process: nothing to report.
cur <- ref
cur$score <- stats::runif(200, 0, 10)
cpsrpt(cur, ref = ref, schema = infsch(ref))
```

```
# One rescaled column and a new category: both surface, worst first.
bad <- cur
bad$score <- bad$score * 5
bad$grade[1:80] <- "z"
r <- cpsrpt(bad, ref = ref, schema = infsch(ref))
r
r$verdict
r$findings[, c("severity", "check", "subject")]

# A missing column is fatal, because nothing downstream can run.
cpsrpt(bad[, c("id", "score")], ref = ref, schema = infsch(ref))$verdict

# validation rules join the findings, and chunks are pinned with a Merkle
# root
r2 <- cpsrpt(
  bad,
  rules = list(ruleinset("grade", c("a", "b", "c"))),
  chunks = c("part one", "part two")
)
summary(r2)
r2$digest$merkle_root

# a data frame is required
try(cpsrpt(1:10))
```

`cpsseal` *Pin and sign every capsule under a folder*

Description

Seals each capsule (a folder holding `data_provenance.json`) under `path`, so that a program in any language only has to write the data file and a plain provenance naming it (`{"resource": {"filename": "data.csv"}}`): on its first run the provenance is pinned to the data file's bytes as `cpsrfrsh` pins it (`resource$sha256`, `size_bytes`, `retrieved_utc`, the row count when one is recorded), then `cpsbndl` signs every file of the capsule into `capsule_bundle.json`, over its `manifest.json` when it has one. `rmbl sign` calls it and then verifies (`cpschk`).

Usage

```
cpsseal(path = ".", key, overwrite = FALSE, glob = "")
```

Arguments

`path`	Character scalar, the folder searched recursively.
`key`	The signing key, as `cpsbndl()` takes it: a shared secret (a string or raw vector) for an `"hmac"` bundle, or a key from `fipskg` or `fipskey`, or an XMSS key from `pqckg`, which signs one capsule after another at consecutive indices (each reserved as `cpssign` reserves it).

overwrite	Logical: sign a capsule that already has a bundle again.
glob	Character scalar: only the provenance files whose path matches this glob (* also matches /), or "".

Details

A provenance already pinned (one with resource$sha256) is left as it is. Side effects: writes data_provenance.json (first run) and capsule_bundle.json in each capsule. Errors, before anything is written: no provenance under path; a capsule that already has a bundle, unless overwrite; then, per capsule, a resource$filename that is not a plain relative path to a file in the capsule, and the refusals of cpsbndl() (a key it cannot sign with).

Value

A data frame, one row per capsule: capsule, pinned (whether the provenance was pinned now), files (the number signed) and public (the HMAC key id or the public key; never the secret).

See Also

cpsbndl, cpschk, blcli (rmbl sign).

Examples

```
root <- tempfile("capsules-")
d <- file.path(root, "one")
dir.create(d, recursive = TRUE)
utils::write.csv(data.frame(x = 1:3), file.path(d, "data.csv"),
                 row.names = FALSE)
writeLines('{"resource": {"filename": "data.csv"}}',
           file.path(d, "data_provenance.json"))
s <- cpsseal(root, "a shared secret")
s
cpschk(root, key = "a shared secret", fp = FALSE)$status

# a second run needs overwrite, and a post-quantum key signs too
try(cpsseal(root, "a shared secret"))
cpsseal(root, fipskg("ML-DSA-44"), overwrite = TRUE)$pinned
```

---

cpssign *Sign a capsule manifest*

Description

Authenticates msg – normally a manifest digest, or the whole manifest text – so a verifier can tell that it came from the holder of the key and has not been altered since. A digest alone shows that data were not corrupted; a signature also shows who vouched for them, because only the key holder can make it. Verify with cpsvrf().

Usage

```
cpssign(
  msg,
  key,
  scheme = NULL,
  ctx = NULL,
  determ = FALSE,
  prehash = "none",
  masked = getOption("rmoriebricklayer.masked", TRUE)
)
```

Arguments

msg	Length-1 character vector (or raw vector) to sign.
key	A shared secret (character/raw) for "hmac", a bricklayer_signing_key from pqckg() for "xmss", or a bricklayer_fips_key from fipskg() for a standardised scheme (in which case scheme is taken from the key and ignored), or a key from fipsmkg(), which signs inside the FIPS 140-3 module that made it.
scheme	"xmss" (post-quantum, asymmetric) or "hmac" (symmetric). Inferred from key when not given.
ctx	Optional context string (character or raw, at most 255 bytes) for the standardised schemes, and ignored by the others. FIPS 204 and FIPS 205 both bind it into the message encoding, so a signature made under one context does not verify under another – which is how the same key is safely used for two purposes.
determ	For the standardised schemes, use the deterministic variant rather than drawing fresh randomness per signature. The signature then depends only on the key, message and context, which is what the standards' test vectors rely on.
prehash	For the standardised schemes, sign a digest of the message rather than the message itself – HashML-DSA (FIPS 204 section 5.4) or HashSLH-DSA (FIPS 205 section 10.2.2). One of "none" (the default, the pure variants) or any hash function the two standards approve. The SHA-2 hashes are "sha224", "sha256", "sha384", "sha512", "sha512_224" and "sha512_256"; the SHA-3 hashes are "sha3_224", "sha3_256", "sha3_384" and "sha3_512"; the SHAKE ones are "shake128" (256-bit output) and "shake256" (512-bit output). The identifier of the pre-hash is bound into the signature, so a pre-hashed signature is never interchangeable with a pure one over the same digest.
masked	TRUE (the default) for the first-order masked ML-DSA signer, FALSE for the plain constant-time one; both give the same signature for the same randomness. The option rmoriebricklayer.masked sets the default for a session, so chain- and bundle-heavy code that has no physical attacker in its threat model can take the plain path everywhere with options(rmoriebricklayer.masked = FALSE). SLH-DSA and XMSS signing are not masked and ignore it. Masking costs about 2.3 to 3.4 times the plain time on ML-KEM decapsulation (4 ms for ML-KEM-768 on this package's test machine) and 7 to 10 times on ML-DSA signing (26 ms for ML-DSA-65), depending on the CPU.

Details

Three kinds of key are accepted: a shared secret (HMAC), an XMSS key from `pqckg()`, or a standardised ML-DSA or SLH-DSA key from `fipskg()`. For new work the standardised keys are the simplest: they sign any number of messages with no state to keep.

With a standardised key the signature is FIPS 204 `ML-DSA.Sign` or FIPS 205 `slh_sign` over the message with its context (hedged by default, deterministic on request). ML-DSA signs in milliseconds; SLH-DSA from a few hundred milliseconds (the `f` sets) to several seconds (the `s` sets at 192 and 256 bits); an XMSS signature rebuilds the key's tree, about 3 s at the default height 10. Slow signatures show a spinner on stderr after 0.75 s when a person is watching (an interactive session or `options(morie.progress = TRUE)`; `options(morie.quiet = TRUE)` silences it), and Ctrl-C stops them.

Refusals: a message that is not one string or a raw vector; a key of the wrong kind for the scheme; a context over 255 bytes; an unknown pre-hash; an exhausted XMSS key, an index the used-index record shows as used, a key whose record cannot be read or written, or an XMSS key the record does not know (see `xmssadv()` and `xmssregkey()`).

With `scheme = "hmac"` the `key` is a shared secret string and the result is an HMAC-SHA-256 tag. Symmetric, so anyone who can verify can also sign.

With `scheme = "xmss"` the `key` is a `pqckg()` object and the result is a post-quantum one-time signature under the key's Merkle root. Asymmetric: a verifier holding only the public root cannot forge.

Value

A list of class `bricklayer_signature`: `scheme`, `signature`, and for XMSS also `auth`, `index`, `root`, `height`, `randomizer` (the per-signature `R` of RFC 8391, which a verifier needs and which therefore travels with the signature) and `wire` – the signature in the RFC's own byte order, `index || R || WOTS || auth`, hex-encoded. `wire` is byte-identical to what the XMSS reference implementation produces from the same key material, so it can be handed to another implementation as bytes.

The key must be advanced after every signature

An XMSS signature consumes a leaf, and **subsequent signing must use the advanced key** – reusing an index breaks the scheme. The signature object carries no key material (it is what verifiers are handed); the advanced key is `xmssadv(key, signature)`, and that is what to save. The key is an R value, so `cpssign()` cannot advance the caller's copy; instead it records, in the session and in the index file described under `xmssadv()`, the highest index each key has signed with, and refuses that index or an earlier one. Passing an exhausted key is an error, not a silent wrap-around.

Security

The implementation is hand-written. Its standardised schemes are checked byte for byte against OpenSSL 3.5 and NIST known-answer vectors in the test suite. Side channels are checked, not assumed: `inst/ctcheck` runs every operation that touches a secret under valgrind memcheck with the secret marked undefined (the ctgrind method), so a branch or a memory address that depends on it is a reported error, and afterwards scans the dead stack for copies of the secret. Both checks run in CI on every change, with GCC and with Clang. Timing is also measured on x86-64 and arm64

hardware (inst/dudect), power leakage is assessed in simulation under the value and the transition models (inst/tvla), and ML-KEM decapsulation and ML-DSA signing are first-order masked by default. These are checks of this code on those compilers and under those leakage models; the README's security section lists what each check does and does not cover.

See Also

cpsvrf(), chmac()

Examples

```
# keep this example's
op <- options(rmoriebricklayer.xmssidxfile = tempfile())
# one-time index record out of your configuration directory; leave the
# default in use Symmetric: one shared secret.
sig <- cpssign("sha256:abc123", key = "shared-secret", scheme = "hmac")
cpsvrf("sha256:abc123", sig, "shared-secret")
cpsvrf("sha256:TAMPERED", sig, "shared-secret")

# Post-quantum: the verifier needs only the public root.
key <- pqckg(height = 2)
s1 <- cpssign("manifest-1", key)
cpsvrf("manifest-1", s1, signpubkey(key))

# Advance the key past the signature before signing again.
key <- xmssadv(key, s1)
key$next_index
s2 <- cpssign("manifest-2", key)
cpsvrf("manifest-2", s2, signpubkey(key))

# A signature does not transfer to another message.
cpsvrf("manifest-1", s2, signpubkey(key))

# A standardised key: no state, any number of signatures.
fk <- fipskg("ML-DSA-44")
s3 <- cpssign("manifest-3", fk, ctx = "release", determ = TRUE)
cpsvrf("manifest-3", s3, fipspubkey(fk), ctx = "release")

# An XMSS signature needs an XMSS key.
try(cpssign("manifest-4", "shared-secret", scheme = "xmss"))
options(op)
```

---

cpsspec *Do Reasonable Alternative Analyses Agree?*

Description

Runs several specifications of the same estimate (other covariates, another model, another handling of outliers or missing values) and reports how far they spread and whether they agree in sign. A finding that holds under one specification only is fragile; one that survives the reasonable alternatives is robust.

Usage

```
cpsspec(data, specs)
```

Arguments

`data`	A data frame.
`specs`	A named list of functions, each taking the data frame and returning one number: the estimate under that specification. The first is the one the paper reports.

Value

A list of class `rmbl_spec`: `ok` (every estimate has the sign of the reported one), `table` (`spec`, `estimate`, `same_sign`), `reported`, `range` and `share` (the share agreeing in sign). It prints as a box and goes into a manifest with `recevid()`.

See Also

`cpsplant()`, `recevid()`.

Examples

```
set.seed(2)
d <- data.frame(x = rnorm(300), z = rnorm(300))
d$y <- 0.5 * d$x + 0.3 * d$z + rnorm(300)
cpsspec(d, list(
  adjusted = function(df) coef(lm(y ~ x + z, df))["x"],
  simple = function(df) coef(lm(y ~ x, df))["x"],
  trimmed = function(df) {
    keep <- abs(df$y - mean(df$y)) < 3 * sd(df$y)
    coef(lm(y ~ x + z, df[keep, ]))["x"]
  }
))
```

cpsvrf *Verify a capsule manifest signature*

Description

Checks `sig` against `msg`. For `"hmac"` the comparison is constant-time (verified under valgrind memcheck in CI: `inst/ctcheck`, cases `digest-equal` and `hmac-sha256`). For XMSS the Winternitz chains are walked to their ends and the authentication path replayed to the Merkle root; the digest is bound to both the leaf index and the root, so a signature cannot be replayed at another index or under another key.

Usage

```
cpsvrf(msg, sig, key, ctx = NULL, prehash = NULL)
```

Arguments

`msg`	The message the signature is claimed to cover.
`sig`	A `bricklayer_signature` from `cpssign()`.
`key`	The shared secret for `"hmac"`, a public key (or full signing key) for XMSS, or a `fipskg()` key or its `fipspubkey()` for a standardised scheme. A signature is not verified against a key of a different scheme.
`ctx`	The context string the signature was made under, for the standardised schemes. A signature made under a different context, or under none, does not verify.
`prehash`	The pre-hash the signature was made with. Taken from the signature when not given, since `cpssign()` records it; supply it to check a signature that arrived without that field.

Details

Returns `FALSE` rather than erroring on a malformed or truncated signature: a verifier must treat unparseable input as "not verified", never as an exception to be caught and ignored.

`TRUE` means the signature is valid for `msg` under `key` (and, for the standardised schemes, `ctx` and the pre-hash). Verification needs only public material, except for HMAC, where the verifier holds the same secret as the signer and could therefore also sign. HMAC tags are compared in constant time.

Only two inputs are errors rather than `FALSE`: a `sig` that is not a `bricklayer_signature` at all, and, for an XMSS signature, a key that is not a key object (a string, say) or a `msg` that is not one string or a raw vector. A key or signature whose class, scheme or fields were edited returns `FALSE`. An XMSS signature object written by 0.5.9 to 0.5.12 (one that carries a `key_state` field) still verifies, with a warning that it exposed the signer's private key.

Value

`TRUE` or `FALSE` (a length-1 logical, never `NA`).

See Also

cpssign(), signpubkey(), fipspubkey().

Examples

```
# keep this example's
op <- options(rmoriebricklayer.xmssidxfile = tempfile())
# one-time index record out of your configuration directory; leave the
# default in use
key <- pqckg(height = 2)
sig <- cpssign("pinned-manifest", key)
pub <- signpubkey(key)

cpsvrf("pinned-manifest", sig, pub)

# Every way of being wrong returns FALSE.
cpsvrf("edited-manifest", sig, pub) # message changed
bad <- sig
bad$signature <- paste0("ff", substring(bad$signature, 3))
cpsvrf("pinned-manifest", bad, pub) # signature edited
cpsvrf("pinned-manifest", sig, signpubkey(pqckg(height = 2))) # foreign key

# A truncated signature is not verified, and does not error.
trunc <- sig
trunc$signature <- substring(sig$signature, 1, 64)
cpsvrf("pinned-manifest", trunc, pub)

# HMAC: the verifier holds the same secret.
tag <- cpssign("pinned-manifest", "shared-secret", scheme = "hmac")
cpsvrf("pinned-manifest", tag, "shared-secret")

# Something that is not a signature at all is an error, not FALSE.
try(cpsvrf("pinned-manifest", list(signature = "00"), pub))
options(op)
```

---

cpsxvrf *Verify a Signature Here and Again With OpenSSL*

Description

Checks an ML-DSA or SLH-DSA signature twice: with this package's own implementation (cpsvrf()) and with the `openssl` program (version 3.5 or later), which implements FIPS 204 and 205 independently. The signature counts as cross-verified only when both accept it.

Usage

```
cpsxvrf(msg, sig, key, ctx = NULL, openssl = Sys.which("openssl"))
```

Arguments

msg	The signed message: a string or a raw vector, as given to cpssign().
sig	A signature from cpssign().
key	The signer's key from fipskg() or fipspubkey(); only its public half is used.
ctx	The context string the signature was made with, or NULL.
openssl	Path to the openssl program. The default is the option rmbl.fips.openssl, else the openssl on the PATH.

Details

OpenSSL is never required. When it is missing, older than 3.5, or the signature is one OpenSSL cannot check (XMSS, an HMAC tag, or a pre-hashed signature), its verdict is NA with the reason, and the result says the signature was verified here only.

The public key is written as a standard SubjectPublicKeyInfo (the NIST object identifiers for each parameter set), the message and the signature as raw bytes, and openssl pkeyutl -verify -rawin checks them, with the context string passed as hexcontext-string. Nothing is sent anywhere: the files live in a temporary folder that is removed afterwards.

Value

A list of class rmbl_xvrf: ok (TRUE only when both implementations accept the signature), ours and openssl (each verdict; openssl is NA when it could not be asked), agree, version (OpenSSL's, or NA), reason (why OpenSSL gave no verdict, or NA) and scheme. It prints as a box that opens with the verdict.

See Also

cpsvrf() for this package's verdict alone, fipskg().

Examples

```
key <- fipskg("ML-DSA-44")
sig <- cpssign("a manifest digest", key, ctx = "release")

# Both implementations accept it (or OpenSSL is reported as absent).
cpsxvrf("a manifest digest", sig, fipspubkey(key), ctx = "release")

# A changed message fails in both.
cpsxvrf("a manifest digest!", sig, key, ctx = "release")$ok
```

`cramerv` *Association in a contingency table*

Description

Cramer's V, with the bias correction of Bergsma (2013) available, and the small-expected-count condition reported rather than assumed away. Use it to say how strongly two categorical variables are associated (0 is none, 1 is complete), for example region by outcome in a cross-tabulation, on a scale that does not depend on the number of rows and columns.

Usage

```
cramerv(tbl, biascor = TRUE, minexp = 5, seed = 1L)
```

Arguments

`tbl`	A table or matrix of non-negative counts, at least 2 by 2 after empty rows and columns are dropped.
`biascor`	Logical scalar, default `TRUE`: whether to apply Bergsma's correction, which removes most of V's upward bias in a sparse table. Worth having: uncorrected V on a sparse table reports association that is an artefact of the table's size.
`minexp`	Numeric scalar, default `5`: the expected count below which the chi-square approximation is treated as unreliable. Reported, and it switches the p-value to the Monte Carlo one; it does not stop the calculation.
`seed`	Integer scalar, default `1`: the seed for the Monte Carlo branch; the caller's random number stream is restored afterwards.

Details

With χ^2 the Pearson statistic (no continuity correction), n the total, r rows and k columns, $\phi^2 = \chi^2/n$ and the uncorrected $V = \sqrt{\phi^2/\min(r-1, k-1)}$. Bergsma's correction replaces ϕ^2 with $\max(0, \phi^2 - (r-1)(k-1)/(n-1))$ and r, k with $r - (r-1)^2/(n-1)$ and $k - (k-1)^2/(n-1)$.

Empty rows and columns (an unused factor level) are dropped first, and `method` says so. When fewer than two rows or columns remain, every statistic is `NA`. When any expected count is below `minexp`, the p-value comes from `stats::chisq.test(simulate.p.value = TRUE, B = 10000)` under `seed`, and the caller's random number stream is restored afterwards.

Errors: `tbl` `NULL`, not numeric, or with a negative or non-finite count; `biascor` not `TRUE` or `FALSE`; `minexp` not a single non-negative number.

Value

A list with `v` (numeric, Cramer's V), `chisq` (the Pearson statistic), `df` (integer), `p_value`, `n` (the total count), `minexp` (the smallest expected count in the table), `cellsbelow` (integer), the number of cells whose expected count falls under the threshold. When any does, `p_value` comes from a Monte Carlo permutation instead of the chi-square approximation, and `method` says which was used.

References

Bergsma, W. (2013). A bias-correction for Cramer's V and Tschuprow's T. *Journal of the Korean Statistical Society* 42(3), 323-328. (Not in the local corpus; cited from the published paper.)

Examples

```
tbl <- rbind(c(120, 80), c(40, 160))
cramerv(tbl)

# No association gives a V near zero.
cramerv(rbind(c(100, 100), c(100, 100)))$v

# A sparse table's uncorrected V overstates the association.
sparse <- rbind(c(3, 1), c(1, 3))
c(raw = cramerv(sparse, biascor = FALSE)$v,
  corrected = cramerv(sparse)$v)

# small expected counts: a Monte Carlo p-value, and the method says so
cramerv(sparse)$method

# an unused level is dropped
cramerv(rbind(c(30, 10), c(0, 0), c(12, 28)))$method

# negative counts are refused
try(cramerv(rbind(c(-1, 2), c(1, 2))))
```

`crank` *Rank correlation and midranks (C backend)*

Description

`cspr()` is Spearman's rho: the Pearson correlation of the ranks, so it measures monotone association rather than linear association and is unaffected by any order-preserving transformation of either variable. `cmidrk()` exposes the ranks themselves; tied values share the average of the ranks they span, which is what makes the result agree with `stats::cor()` on tied data. Use Spearman's rho when two variables rise and fall together but not along a straight line (counts against rates, incomes against ranks), or when a few extreme values would dominate a Pearson correlation.

Usage

```
cspr(x, y)

cmidrk(x)
```

Arguments

`x, y`	Numeric (or logical) vectors of the same length.

Details

With r_i and s_i the midranks of x_i and y_i,

$$\rho = \frac{\sum_i (r_i - \bar{r})(s_i - \bar{s})}{\sqrt{\sum_i (r_i - \bar{r})^2 \sum_i (s_i - \bar{s})^2}},$$

the Pearson correlation of the ranks (Spearman, 1904). Because ties get midranks, the shortcut $1 - 6\sum d_i^2/(n(n^2-1))$ is NOT what is computed; the full formula above is, and it stays correct with ties.

`cspr()` does not remove missing values: an `NA` in either vector gives `NaN`, and so does a constant vector (its ranks have no variance). `cmidrk()` leaves an `NA` in place and ranks the rest, as `rank(x, na.last = "keep")` does. Refusals: vectors of different lengths (`"must have the same length"`) and a character, factor or list argument (`"must be numeric"`). Ranking sorts, so the cost is $O(n \log n)$.

Value

`cspr()`: a length-1 numeric in $[-1, 1]$ (`NaN` as described above). `cmidrk()`: a numeric vector the length of `x`, the rank of each value (1 for the smallest; tied values share the mean of their ranks; `NA` kept as `NA`).

References

Spearman C (1904). The proof and measurement of association between two things. *The American Journal of Psychology* 15(1), 72–101. DOI 10.2307/1412159

See Also

`ccorr()` for the Pearson correlation.

Examples

```
x <- c(1, 2, 3, 4, 5)
y <- c(2, 4, 9, 16, 25)

# Perfectly monotone but not linear: rho is 1 where Pearson is not.
cspr(x, y)
ccorr(x, y)

# Agrees with stats::cor(), ties included.
xt <- c(1, 2, 2, 2, 5, 5, 7)
yt <- c(3, 1, 1, 4, 4, 9, 2)
all.equal(cspr(xt, yt), stats::cor(xt, yt, method = "spearman"))

# Tied values share the average of the ranks they cover.
cmidrk(xt)
all.equal(cmidrk(xt), rank(xt))

# Invariant to any monotone rescaling.
all.equal(cspr(x, y), cspr(exp(x), log(y)))
```

```
# Missing values are kept in place by cmidrk().
cmidrk(c(3, NA, 1, 3))
try(cspr(1:3, 1:4))
```

`crfpm25` *Concentration-response functions for PM2.5 and NO2*

Description

A concentration-response function (CRF) turns an air-pollution concentration into the relative risk (RR) of a health outcome compared with a counterfactual "clean air" level: an RR of 1.05 means 5% more deaths (or cases) than at the reference concentration. These are the first step of any burden estimate; `polbrdn` and `vrfpol` call them for you.

Usage

```
crfpm25(expo, y = "allcause", refconc = 5.8)

crfno2(expo, y = "allcause", refconc = 10, betaper10 = NULL)
```

Arguments

`expo`	Ambient concentration in micrograms per cubic metre, a single non-negative number or a vector of them (one per area, day or person).
`y`	PM2.5: `"allcause"`, `"ihd"` or `"stroke"` (deaths). NO2: `"allcause"` and `"respiratory"` (deaths; Huangfu and Atkinson 2020, RR 1.02 and 1.03 per 10 micrograms per cubic metre) or `"chiast"` (incident cases, not deaths; Khreis et al. 2017, OR 1.05 per 4 micrograms per cubic metre, scaled to per 10).
`refconc`	Counterfactual concentration below which no excess risk is assumed. PM2.5 default 5.8, the lower bound of the GBD 2019 theoretical minimum risk exposure level (the WHO 2021 guideline is 5); NO2 default 10, the WHO 2021 guideline.
`betaper10`	Optional log relative risk per 10 micrograms per cubic metre (a single finite number, e.g. `log(1.04)`) overriding the NO2 outcome table; `NULL` (the default) uses the table.

Details

`crfpm25()` is log-linear for all-cause mortality with the pooled cohort estimate of the WHO 2021 guideline review (Chen and Hoek 2020: RR 1.08, 95% CI 1.06-1.09, per 10 micrograms per cubic metre),

$$RR(z) = \exp(\ln(1.08)(z - z_{cf})/10)$$

for $z > z_{cf}$ and 1 otherwise; for the cause-specific outcomes (ischaemic heart disease, stroke) it is the Integrated Exposure-Response curve of Burnett et al. (2014, equation 1),

$$RR(z) = 1 + \alpha(1 - e^{-\gamma(z - z_{cf})^{\delta}}),$$

with the GBD 2013 parameter triples; the IER was fitted per cause and has no all-cause form. `crfno2()` is the log-linear function of the WHO 2021 review (Huangfu and Atkinson 2020: RR 1.02, 95% CI 1.01-1.04, per 10 micrograms per cubic metre for all-cause mortality),

$$RR(z) = \exp(\beta(z - z_{cf})/10).$$

A vector of exposures gives the mean RR and log-RR. The per-unit relative risks are then in `extra$rrperunit`.

Every function has zero excess risk at and below the reference concentration ($z \leq z_{cf}$ gives RR 1). The published 95% limits of the log-linear coefficients are carried through the same curve into `rrlo` and `rrup`; the IER parameters come without an interval, so those are `NA` for `"ihd"` and `"stroke"`, and also when `betaper10` is supplied.

Refusals: an unknown `y` (the message lists the available ones); an exposure that is missing, infinite, negative, logical or empty; and a `refconc` (or `betaper10`) that is not a single finite number, or a negative `refconc`. A supplied `betaper10` may be negative (a protective association) and replaces the table, in which case `y` is only a label. The functions are pure computations: no file, network or option is used.

Value

A list of class `rmbl_crf` (`plot()` draws the curve) with: `rr` and `logrr`, the relative risk and its logarithm at the exposure (their means over a vector of exposures); `rrlo` and `rrup`, the RR at the published 95% limits (`NA` where there are none); `refconc`; `expconc` (the exposure, or the mean of a vector); `pollutant` (`"PM2.5"` or `"NO2"`); `citation` (the source as a string); and `extra`, a list with the coefficient (`betaper10` and `betaper10ci` for the log-linear forms, `alpha`, `gamma`, `delta` for the IER), `form` (PM2.5 only), `outcome`, and for a vector of exposures `rrperunit`, the RR at each one.

References

Chen, J. and Hoek, G. (2020). Long-term exposure to PM and all-cause and cause-specific mortality: a systematic review and meta-analysis. Environment International 143, 105974. Huangfu, P. and Atkinson, R. (2020). Long-term exposure to NO2 and O3 and all-cause and respiratory mortality: a systematic review and meta-analysis. Environment International 144, 105998. Khreis, H. et al. (2017). Exposure to traffic-related air pollution and risk of development of childhood asthma: a systematic review and meta-analysis. Environment International 100, 1-31. GBD 2019 Risk Factors Collaborators (2020). The Lancet 396(10258), 1223-1249. Burnett, R. T. et al. (2014). An integrated risk function for estimating the global burden of disease attributable to ambient fine particulate matter exposure. Environmental Health Perspectives 122(4), 397-403. WHO (2021). Global Air Quality Guidelines.

Examples

```
# All-cause mortality at an annual PM2.5 of 12 micrograms per cubic metre.
r <- crfpm25(12)
c(rr = r$rr, lower = r$rrlo, upper = r$rrup)

# The cause-specific IER curve for ischaemic heart disease.
crfpm25(12, y = "ihd")$rr

# NO2 and respiratory deaths; at or below the reference there is no
```

```
# excess risk.
crfno2(25, y = "respiratory")$rr
crfno2(8)$rr

# A vector of exposures (one per area): the mean RR and each area's RR.
crfpm25(c(5, 12, 20))$extra$rrperunit

# Your own coefficient: RR 1.04 per 10 micrograms per cubic metre.
crfno2(30, betaper10 = log(1.04))$rr

# An outcome without a published function is refused.
try(crfpm25(12, y = "asthma"))
```

`crmdens` *Where incidents cluster: a kernel density surface*

Description

Estimates the density of incident locations – police-reported occurrences, calls for service, addresses of people in a program – as a smooth surface in incidents per square kilometre, and the smallest areas holding a given share of them (the highest-density regions). The plot draws the surface as a heat map with those regions outlined.

Usage

```
crmdens(
  lon,
  lat,
  w = NULL,
  bw = NULL,
  ngrid = 200L,
  cover = c(0.5, 0.9),
  xy = c("lonlat", "planar")
)
```

Arguments

`lon`, `lat`	Coordinates of the incidents: longitude and latitude in degrees (WGS84) by default, or planar coordinates in metres with `coords = "planar"`. Points with a missing coordinate are dropped with a message; in degrees, points at exactly (0, 0), which is how several open-data portals publish an incident with no location, are dropped too.
`w`	Optional non-negative weight per incident. Default 1.
`bw`	Kernel standard deviation in kilometres: one value for both directions or two (east-west, north-south). Default: Scott's normal-reference rule, $\sigma n^{-1/6}$ for each direction.

`ngrid`	Grid nodes per direction. Default 200.
`cover`	Shares of the incidents whose highest-density regions are reported and outlined. Default `c(0.5, 0.9)`.
`xy`	`"lonlat"` (the default) or `"planar"`.

Details

The estimate is the Gaussian product-kernel density, computed on the grid by linear binning followed by an exact discrete convolution, which agrees with the direct sum to within the binning error (of order the squared grid spacing over the squared bandwidth). The grid extends three bandwidths past the outermost incident so the surface integrates to the total weight.

A highest-density region for share p is the set of grid cells with the largest densities whose estimated incidents add up to p of the total. Its area is the spatial counterpart of a concentration index: when half the incidents fall in a few per cent of the mapped area, they are concentrated. The bandwidth decides how sharp the surface is, so it should be reported with any map; the default suits a first look, not a definitive one.

Errors: `lon` and `lat` of different lengths, or (with `coords = "lonlat"`) a coordinate that is not a longitude and latitude in degrees; fewer than two located incidents with positive weight; `ngrid` not a whole number from 10 to 2000; `cover` not shares strictly between 0 and 1; incidents all on one line with no `bw` given; a `bw` that is not one or two positive numbers, or longer than half the earth's circumference. Incidents with a missing coordinate are dropped.

Value

A list of class `rmbl_density`: `field` (incidents per square kilometre on the grid), `x` and `y` (grid coordinates in kilometres from the centre), `lon` and `lat` (the same grid in degrees, for `"lonlat"`), `bandwidth`, `n`, `dropped`, `points` (the projected incidents), and `regions`: one row per coverage share with the density `level` that bounds it, its `areakm2`, and its share of the mapped area.

References

Scott, D. W. (2015). *Multivariate Density Estimation: Theory, Practice, and Visualization*, 2nd ed. Wiley, Hoboken, NJ.

Wand, M. P. (1994). Fast computation of multivariate kernel estimators. *Journal of Computational and Graphical Statistics*, 3(4), 433-445.

Hyndman, R. J. (1996). Computing and graphing highest density regions. *The American Statistician*, 50(2), 120-126.

Chainey, S., Tompson, L. and Uhlig, S. (2008). The utility of hotspot mapping for predicting spatial patterns of crime. *Security Journal*, 21(1-2), 4-28.

See Also

`knoxtst`, `plmfld` (the same kind of surface, for an air pollutant).

Examples

```
b <- read.csv(system.file("extdata", "tps_break_enter_d14_2024.csv",
                          package = "rmoriebricklayer"))
kd <- crmdens(b$lon, b$lat, ngrid = 120)
kd
plot(kd)

# a fixed 300 m bandwidth on a coarser grid: a sharper surface
kd3 <- crmdens(b$lon, b$lat, bw = 0.3, ngrid = 60)
kd3$regions

# planar coordinates in metres
set.seed(2)
xy <- cbind(rnorm(200, sd = 500), rnorm(200, sd = 500))
crmdens(xy[, 1], xy[, 2], xy = "planar", ngrid = 40)$bandwidth

# longitudes and latitudes are checked
try(crmdens(c(-79.4, 200), c(43.7, 43.8)))
```

`crob` *Quantiles, median and robust spread (C backend)*

Description

`cqnt()` is the type-7 quantile, which is R's default, so it agrees with `stats::quantile(x, probs, type = 7)`. `cmed()` is the 50% point. `cmad()` is the median absolute deviation, scaled by `constant` so that it estimates the standard deviation of a normal sample. `ciqr()` is the interquartile range and `ctkyfnc()` the outlier fences drawn at `k` IQRs beyond the quartiles.

Usage

```
cqnt(x, probs = c(0, 0.25, 0.5, 0.75, 1))

cmed(x)

cmad(x, const = 1.4826)

ciqr(x)

ctkyfnc(x, k = 1.5)
```

Arguments

`x`	Numeric (or logical) vector; coerced with `as.numeric()`.
`probs`	Numeric vector of probabilities, each in $[0, 1]$ (default the minimum, quartiles and maximum).

`const` Scale factor for `cmad()`. The default `1.4826` makes the MAD consistent for the standard deviation under normality, and is the same rounded value `stats::mad()` uses, so the two agree exactly. The unrounded consistency constant is `1 / qnorm(3/4)` = 1.4826022185...; pass it explicitly if you want the extra digits, or `1` for the unscaled median deviation.

`k` Fence width in IQRs, a single non-negative number (default 1.5, Tukey's convention; 3 is the usual "far out" cutoff).

Details

These are the robust counterparts of `cmom()`: a single corrupted row can move a mean or a variance arbitrarily far, but moves a median or a MAD hardly at all – which is what you want when deciding whether a freshly fetched column is still the column a capsule was pinned against, or when screening a column for values to inspect by hand.

The type-7 quantile at probability p interpolates linearly between the order statistics $x_{(j)}$ and $x_{(j+1)}$ with $h = (n-1)p+1$, $j = \lfloor h \rfloor$: $Q(p) = x_{(j)} + (h-j)(x_{(j+1)} - x_{(j)})$ (Hyndman and Fan, 1996). The MAD is $c \cdot \mathrm{median}_i |x_i - \mathrm{median}(x)|$ with c = `constant`; the IQR is $Q(0.75) - Q(0.25)$; and the fences are $Q(0.25) - k \cdot \mathrm{IQR}$ and $Q(0.75) + k \cdot \mathrm{IQR}$ (Tukey, 1977).

Missing values are not removed: an `NA` or `NaN` anywhere in `x` makes every result `NaN` (unlike `stats::quantile()`, which stops). An empty `x` gives `NaN`. `cqnt()` refuses an empty `probs` (`"probs is empty"`) and any probability outside $[0, 1]$ or missing (`"must lie in [0, 1]"`); every function refuses a character, factor or list `x` (`"must be numeric"`). The quantiles sort a copy of `x`, so the cost is $O(n \log n)$.

Value

`cqnt()`: a numeric vector the length of `probs`, in the units of `x`, named by percentage (`"25%"`, ...) as `stats::quantile()` names them. `cmed()`, `cmad()` and `ciqr()`: a length-1 numeric in the units of `x`. `ctkyfnc()`: a named length-2 numeric, `lower` and `upper`.

References

Hyndman RJ, Fan Y (1996). Sample quantiles in statistical packages. *The American Statistician* 50(4), 361–365. DOI 10.1080/00031305.1996.10473566

Tukey JW (1977). *Exploratory Data Analysis*. Addison-Wesley.

See Also

`cmom()` for the non-robust moments, `ctrmmean()` for a robust centre that still averages.

Examples

```
x <- c(2, 4, 4, 4, 5, 5, 7, 9)
cqnt(x, c(0.25, 0.5, 0.75))
all.equal(cqnt(x, c(0.1, 0.9)),
          as.numeric(stats::quantile(x, c(0.1, 0.9))))

cmed(x)
cmad(x)
```

```
all.equal(cmad(x), stats::mad(x))
cmad(x, const = 1)                  # unscaled median deviation
cmad(x, const = 1 / stats::qnorm(3/4))  # unrounded constant

ciqr(x)
ctkyfnc(x)

# Robustness: one wild value barely moves the median, but moves the
# mean a long way.
wild <- c(x, 1000)
c(mean = mean(wild), median = cmed(wild))

# Values outside the fences are the candidates to inspect.
f <- ctkyfnc(wild)
wild[wild < f[["lower"]] | wild > f[["upper"]]]

# A missing value is not dropped: the answer is NaN.
cmed(c(1, NA, 3))

# A probability outside [0, 1] is refused.
try(cqnt(x, 1.5))
```

---

csha256 *SHA-256 hex digest (C backend)*

Description

Computes the SHA-256 hash (FIPS 180-4) of text or raw bytes with the self-contained implementation in the package's compiled core, and returns it as a 64-character lowercase hexadecimal string. A hash is a short fingerprint: any change to the input changes it, so it is used throughout the package to record exactly which data, mapping or file a result came from. Sibling packages use the same routine through `LinkingTo: rmoriebricklayer`.

Usage

```
csha256(x)
```

Arguments

x	A character vector, or a raw vector. Other atomic vectors are converted with `as.character()`.

Details

For a raw vector the bytes are hashed as one message and one digest is returned. Any other input is converted with `as.character()` and each element is hashed separately, as the bytes of the string as stored in memory (no re-encoding). In a UTF-8 session those are the UTF-8 bytes; for a string marked latin1 they are the latin1 bytes, so use `enc2utf8()` first when the digest must be the same

on every platform. NA gives NA. Numbers are hashed through their printed form (1 hashes as "1"), and a factor through its labels.

For a digest of a whole R object, hash its serialization (see the examples) or use dgstobj; for BLAKE2b use cblake2b.

Value

A character vector of 64-character lowercase hex digests: one per element for character input, length 1 for raw input (character(0) for an empty character vector).

References

National Institute of Standards and Technology (2015). Secure Hash Standard (SHS). FIPS PUB 180-4. DOI 10.6028/NIST.FIPS.180-4

Examples

```
# Hash a character scalar (NIST test vector for "abc").
csha256("abc")
# ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

# Vectorised over character input: one digest per element.
csha256(c("abc", "def", NA))

# Raw input hashes the bytes directly; identical to the character form.
identical(csha256("abc"), csha256(charToRaw("abc")))

# The empty message.
csha256(raw(0))

# Fingerprint an arbitrary object via its serialization.
csha256(serialize(list(a = 1L, b = "x"), NULL))
```

---

csha512 *SHA-512 hex digest (C backend)*

Description

Hashes character or raw input with the self-contained SHA-512 (FIPS 180-4) in the compiled core and returns the digest as hex. A digest is a fingerprint: any change to the input gives a different value, so it pins a data file, a code string or a manifest. Use it over csha256() when a pin has to outlive the capsule by decades: the wider digest leaves more margin, including against a quantum adversary, for whom Grover's algorithm halves the effective preimage exponent.

Usage

```
csha512(x)
```

Arguments

x	A character vector (one digest per element) or a raw vector (one digest of all its bytes).

Details

Character input is hashed element by element, each string as its bytes (UTF-8 text gives the UTF-8 bytes; no newline is added). NA elements give NA. Other atomic input (numbers, logicals) is first converted with as.character(), so csha512(1) is the digest of the text "1". A raw vector is hashed as one byte string. For files use sha512file(). Pure computation; no side effects.

Value

A character vector of 128-character lowercase hex digests: one per element for character input (NA for NA), length 1 for raw input.

See Also

csha256(), ccrc32(), sha512file()

Examples

```
# FIPS 180-4 test vector for "abc".
csha512("abc")

# Vectorised over character input; NA stays NA.
csha512(c("abc", "def", NA))

# Raw input hashes the bytes directly, and agrees with the character
# form for the same bytes.
identical(csha512("abc"), csha512(charToRaw("abc")))

# Twice the digest width of SHA-256.
c(sha256 = nchar(csha256("abc")), sha512 = nchar(csha512("abc")))

# numbers are hashed as their text
identical(csha512(1), csha512("1"))
```

---

cspread *Standard deviation and Euclidean distance (C backend)*

Description

csdev() returns the standard deviation of a numeric vector and cdistm() the straight-line (Euclidean) distance between two vectors of the same length. Both run in the package's compiled core, the same code that rmorie and rmoriedata reach through LinkingTo: rmoriebricklayer, so a figure computed here is the figure those packages compute. Use them when you want the shared core's answer rather than base R's, for example when checking a capsule's recorded summary statistics against a fresh computation.

Usage

```
csdev(x, ddof = 1L)

cdistm(a, b)
```

Arguments

x, a, b	Numeric (or logical) vectors; coerced with as.numeric(). a and b must have the same length.
ddof	Delta degrees of freedom, a single number; as.integer() truncates it to an integer. The divisor is n - ddof. The default 1 gives the sample standard deviation, matching stats::sd(); 0 gives the population figure.

Details

csdev() computes

$$s = \sqrt{\frac{1}{n - \mathrm{ddof}} \sum_{i=1}^{n} (x_i - \bar{x})^2},$$

so ddof = 1 is the sample standard deviation of stats::sd() and ddof = 0 the population figure. cdistm() computes $\sqrt{\sum_i (a_i - b_i)^2}$.

Missing values propagate: there is no na.rm, and one NA makes the result NA. Remove them first (for example with stats::na.omit()) when that is what you want. A vector with no more than ddof values has no defined spread and gives NaN (one value with the default ddof = 1, or an empty vector). Two empty vectors are at distance 0.

Refusals: a character, factor, list or data-frame x, a or b ("must be numeric"), and for cdistm() vectors of different lengths ("must have the same length"). Logical input is accepted and counted as 0/1. Both run in one pass over the data and allocate nothing beyond the coerced input.

Value

csdev(): a length-1 numeric, the standard deviation in the units of x (NA when x holds a missing value, NaN when length(x) <= ddof). cdistm(): a length-1 non-negative numeric, the distance in the units of a and b.

See Also

cmom() for the mean, variance, skewness and kurtosis in a single pass; cmad() for a spread that one wild value cannot move.

Examples

```
# Sample standard deviation, matching stats::sd().
csdev(c(2, 4, 4, 4, 5, 5, 7, 9))
all.equal(csdev(1:10), stats::sd(1:10))

# ddof = 0 divides by n instead of n - 1.
csdev(1:10, ddof = 0)
```

```
all.equal(csdev(1:10, ddof = 0), sqrt(mean((1:10 - mean(1:10))^2)))

# Missing values propagate; remove them first if they should be ignored.
csdev(c(3, NA, 5))
csdev(stats::na.omit(c(3, NA, 5)))

# Euclidean distance between two points.
cdistm(c(0, 0), c(3, 4))     # 5
cdistm(1:5, 1:5)             # 0 -- a point is zero from itself

# Vectors of different lengths are refused rather than recycled.
try(cdistm(1:3, 1:2))
```

`cstat` *Fast summary statistics (C backend)*

Description

`cmn()`, `cvari()` and `ccorr()` compute the mean, the sample variance and the Pearson correlation in compiled code. They are thin R wrappers over the kernels in the package's C/C++ core, the same ones that other packages call through `LinkingTo: rmoriebricklayer`, so an R result and a compiled result agree exactly. Use them where you want those shared kernels; for everyday work `mean()`, `var()` and `cor()` give the same numbers.

Usage

```
cmn(x)

cvari(x)

ccorr(x, y)
```

Arguments

`x`, `y`	Numeric (or logical) vectors. `ccorr()` needs `x` and `y` of the same length.

Details

With n values,

$$\bar{x} = \frac{1}{n}\sum_i x_i, \qquad s^2 = \frac{1}{n-1}\sum_i (x_i - \bar{x})^2, \qquad r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}}.$$

The variance uses the $n - 1$ (sample) denominator, as `var` does. The mean is accumulated so that a sum of finite values that would overflow a double (for example two values of `1e308`) still gives the finite mean.

There is no `na.rm`: any `NA` or `NaN` gives `NA` or `NaN`; remove missing values first, for example with `na.omit`. Edge cases return `NaN` rather than an error: an empty vector for the mean, fewer than two values for the variance, and a constant vector for the correlation.

Errors: input that is a character vector, factor, list or data frame (logical input is accepted and read as 0/1); `x` and `y` of different lengths in `ccorr()`.

Value

`cmn()`, `cvari()` and `ccorr()` each return a length-1 numeric.

See Also

`cnormpdf`, `csha256`.

Examples

```
## cmn(): sample mean (NA/NaN propagate; no na.rm)
cmn(1:10)                 # 5.5
cmn(c(1, 2, NA))          # NA -- call stats::na.omit() first if needed
cmn(stats::na.omit(c(1, 2, NA)))
cmn(c(1e308, 1e308))      # no overflow

## cvari(): n-1 (sample) variance, matching stats::var()
cvari(c(2, 4, 4, 4, 5, 5, 7, 9))
all.equal(cvari(1:10), stats::var(1:10))   # agrees with base R

## ccorr(): Pearson correlation of two equal-length vectors
ccorr(1:10, (1:10)^2)        # strong positive, near 0.97
ccorr(1:10, 10:1)            # perfect negative: -1

## refusals
try(cmn(c("1", "2")))     # character input is not coerced
try(ccorr(1:3, 1:4))         # lengths differ
```

`ctrm` *Trimmed and winsorized means (C backend)*

Description

Two ways to stop a handful of extreme rows dominating a column's centre. `ctrmmean()` DISCARDS the `floor(n * trim)` largest and smallest values, as `mean(x, trim = )` does. `cwinsmean()` instead PULLS THEM IN to the most extreme surviving values, so every observation still contributes weight – usually the better choice when the extremes are real measurements rather than errors.

Usage

```
ctrmmean(x, trim = 0.1)

cwinsmean(x, trim = 0.1)
```

Arguments

x	Numeric (or logical) vector; coerced with as.numeric().
trim	Proportion trimmed (or winsorized) at *each* end, a single number in $[0, 0.5]$ (default 0.1, i.e. 10% at each end). At 0.5 both reduce to the median.

Details

With $g = \lfloor n \cdot \mathrm{trim} \rfloor$ and the sorted values $x_{(1)} \le \cdots \le x_{(n)}$, the trimmed mean is $\frac{1}{n-2g} \sum_{i=g+1}^{n-g} x_{(i)}$ and the winsorized mean replaces $x_{(1)}, \ldots, x_{(g)}$ by $x_{(g+1)}$ and $x_{(n-g+1)}, \ldots, x_{(n)}$ by $x_{(n-g)}$ before averaging all n values. At trim = 0.5 both return the median, as mean() does.

A trim that is not a single number in $[0, 0.5]$ is an error. An empty x, or any NA/NaN in x, gives NaN (missing values are not removed). A character, factor or list x is refused ("must be numeric"). Both sort a copy of x, so the cost is $O(n \log n)$.

Value

A length-1 numeric in the units of x.

See Also

cmed() and cmad() for the other robust summaries; cmom() for the ordinary mean.

Examples

```
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 100)

mean(x)                              # dragged up by the 100
ctrmmean(x, 0.1)             # the 100 and the 1 dropped
cwinsmean(x, 0.1)          # the 100 pulled back to 9

# Agrees with base R's own trimming.
all.equal(ctrmmean(x, 0.2), mean(x, trim = 0.2))

# trim = 0 is the plain mean; trim = 0.5 is the median.
all.equal(ctrmmean(x, 0), mean(x))
all.equal(ctrmmean(x, 0.5), cmed(x))

# A trim past one half is not a trim.
try(cwinsmean(x, 0.7))
try(ctrmmean(letters))
```

`cunif` *Reproducible uniforms shared with morie's Python arm (splitmix64)*

Description

Returns `n` pseudo-random numbers on [0, 1) from a 64-bit seed with the splitmix64 generator, the same numbers morie's Python computes for the same seed. Use it where the R and Python arms must draw identical `"random"` values so that their results agree: the within-day jitter of tied event dates in the TPS Hawkes fits, or a subsample.

Usage

```
cunif(n, seed)
```

Arguments

`n`	Number of uniforms, one non-negative whole number (0 gives `numeric(0)`).
`seed`	The seed, one whole number in $[0, 2^{53})$ (so that it is exact as an R double).

Details

splitmix64 (Steele, Lea & Flood 2014) advances a 64-bit state by the constant `0x9E3779B97F4A7C15` and mixes it with two multiply-xorshift rounds; the top 53 bits of each output divided by 2^{53} give a double in [0, 1). The stream depends only on `seed`: R's own generator and `set.seed()` are neither used nor changed. It is not a statistical replacement for R's generators.

Errors: `n` not one non-negative whole number; `seed` not one whole number in $[0, 2^{53})$.

Value

A numeric vector of length `n`, values in [0, 1).

References

Steele GL, Lea D, Flood CH (2014). Fast splittable pseudorandom number generators. *OOPSLA 2014*, 453–472. DOI 10.1145/2660193.2660195

Examples

```
cunif(3, 42)
identical(cunif(5, 1), cunif(5, 1))

# a stream is a prefix of a longer one with the same seed
all.equal(cunif(3, 7), cunif(10, 7)[1:3])

# R's own generator is left alone
set.seed(1); a <- runif(1)
set.seed(1); invisible(cunif(100, 9)); b <- runif(1)
identical(a, b)
```

```
# negative counts and seeds past 2^53 are refused
try(cunif(-1, 1))
try(cunif(2, 2^53))
```

---

cwbscr *The Community Well-Being index of a community*

Description

Indigenous Services Canada's Community Well-Being (CWB) index: income, education, housing and labour force activity, each scored from 0 to 100, and their mean, from census profile figures.

Usage

```
cwbscr(
  incpercap,
  highsch20p,
  uni25p,
  notcro,
  nomajrep,
  partic,
  empl,
  bounds = c(2650, 75000)
)
```

Arguments

`incpercap`	Income per capita, in the dollars of the series' bounds.
`highsch20p`, `uni25p`	Percentages for education.
`notcro`, `nomajrep`	Percentages for housing.
`partic`, `empl`	Percentages for labour force activity.
`bounds`	The income bounds (default the 2016 and 2021 series: 2650 and 75000).

Details

The components, as the index's methodology sets them out (Penney, O'Sullivan and Senecal 2012; O'Sullivan and McHardy 2013; Indigenous Services Canada's reports on trends, 1981 to 2016 and 2021):

income $100(\log y - \log a)/(\log b - \log a)$ for income per capita y (total income over total population) between the bounds a and b (2,650 and 75,000 dollars in the 2016 and 2021 series; 2,000 and 40,000 in the earlier ones), kept between 0 and 100 here.

education two thirds of the percentage aged 20 and over with at least a high school certificate, plus one third of the percentage aged 25 and over with a university degree at the bachelor's level or above.

housing half the percentage of people in dwellings with no more than one person per room, plus half the percentage in dwellings that need no major repairs.

labour half the participation rate, plus half the employment rate of those participating (ages 20 to 64).

The index is the mean of the four. Indigenous Services Canada computes it from unrounded census microdata, scores only communities of 65 people or more and suppresses the components of communities under 250 people or 40 households: scores from published, rounded profile figures come close to the published index but need not equal it, and a small community's should be treated as the Department would. The full current methodology is available from the Department on request. The index measures socio-economic conditions as the census counts them; it says nothing of culture, language or relationship to land, and is not a measure of wellness as First Nations and Inuit define it. The CWB is Indigenous Services Canada's index, with bounds in the Canadian dollars of its series: for another country, set `bounds` in its currency and read the result as a CWB-style score, not comparable with Canada's.

Value

A data frame, one row per community: `income`, `education`, `housing`, `labour`, `cwb`.

References

Penney C, O'Sullivan E, Senecal S (2012). The Community Well-Being Index (CWB): Examining Well-Being in Inuit Communities, 1981-2006. Aboriginal Affairs and Northern Development Canada.

O'Sullivan E, McHardy M (2013). The Community Well-Being (CWB) Index: Disparity in Well-Being Between First Nations and Other Canadian Communities Over Time. In White JP, Peters J, Beavon D, Spence N (eds), Aboriginal Well-Being: Canada's Continuing Challenge. Thompson Educational.

Indigenous Services Canada (2019). Report on trends in First Nations communities, 1981 to 2016 (Community Well-Being index).
sac-isc.gc.ca/eng/1345816651029/1557323327644

Examples

```
# made-up profile figures for two communities, to show the computation
cwbscr(incpercap = c(18000, 31000), highsch20p = c(58, 81),
          uni25p = c(6, 19), notcro = c(71, 95),
          nomajrep = c(62, 90),
          partic = c(55, 68), empl = c(78, 92))
```

`cwtd` *Weighted mean and variance (C backend)*

Description

The weighted mean and weighted variance of a numeric vector, for data where some observations count for more than others: an average over areas weighted by population, or measurements weighted by how precise each one is. Both come from the compiled core in one pass.

Usage

```
cwtd(x, w)
```

Arguments

`x`	Numeric vector of observations.
`w`	Numeric vector of non-negative weights, the same length as `x`. Only their relative sizes matter for the mean; the variance's correction uses them as given.

Details

The mean is $\bar{x}_w = \sum_i w_i x_i / \sum_i w_i$. The weights are treated as RELIABILITY weights (how precisely each observation is known), so the variance is

$$s_w^2 = \frac{\sum_i w_i (x_i - \bar{x}_w)^2}{\sum_i w_i - \sum_i w_i^2 / \sum_i w_i},$$

the unbiased estimator under that reading. With every weight equal to 1 it reduces exactly to `stats::var()`. Frequency weights (a weight of 3 meaning three identical rows) call for the denominator $\sum w - 1$ instead, and survey design weights call for a design-based variance, so do not use this variance for those.

Refusals: `x` and `w` of different lengths (`"must have the same length"`), any negative weight (`"must be non-negative"`), and a character, factor or list argument (`"must be numeric"`). All-zero weights, or an `NA` in either vector, give `NaN` for both entries.

Value

A named length-2 numeric vector: `mean` (units of `x`) and `variance` (squared units of `x`).

See Also

`cmom()` for the unweighted moments, `cipw()` for inverse-probability weights to feed in here.

Examples

```
x <- c(10, 20, 30, 40)
w <- c(1, 1, 2, 4)
cwtd(x, w)

# The mean agrees with base R.
all.equal(cwtd(x, w)[["mean"]], stats::weighted.mean(x, w))

# Equal weights recover the unweighted variance.
all.equal(cwtd(x, rep(1, 4))[["variance"]], stats::var(x))

# A single dominant weight pulls the mean onto that observation.
cwtd(x, c(1, 1, 1, 1000))[["mean"]]

# A negative weight is refused.
try(cwtd(x, c(1, -1, 1, 1)))
```

---

deccodes *Decode numeric category codes with an explicit code-to-label dictionary*

Description

Turns a vector of category codes (for example `1`, `2`, `3` from an SPSS, Stata or SAS file) into the category labels they stand for, using a code-to-label dictionary that you write out. Every code that occurs in the data must be in the dictionary; a code without a label is an error rather than a silent `NA` or a pass-through. Use it whenever a column arrives as numbers that are really categories, before any table or model, so that a group can never be renumbered by treating codes (or `as.numeric()` on a factor of them) as the categories.

Usage

```
deccodes(x, vlabs, keepna = TRUE)
```

Arguments

`x`	Vector of codes: numeric, integer, character or factor. A factor is read through its labels, not its integer codes; an ordered factor is treated the same as an unordered one (its level order is not used).
`vlabs`	Named character vector, code = label, for example `c("1" = "White", "2" = "Black")`. The names are the codes as they print with `as.character()`; the values are the labels. Codes in the dictionary that do not occur in `x` are allowed.
`keepna`	Logical, default `TRUE`: `NA` codes stay `NA` in the result. `FALSE` makes any `NA` code an error, for columns where missingness should not occur.

Details

x is converted to character (a factor by its labels, a number by `as.character()`, so `1` and `1.0` both become `"1"`) and every distinct non-missing code is looked up by name in `vlabs`. The lookup itself is done by grdrcd, so the result carries the same `recode_audit` attribute, including a SHA-256 of the dictionary.

Errors:

- `vlabs` not a character vector with non-empty names;
- a code in x that is not a name of `vlabs` (the message lists the unmapped codes);
- any NA in x when `keepna = FALSE`.

For a haven-style labelled vector, which carries its dictionary as a `labels` attribute, use declbld, which returns a factor with the levels in code order.

Value

A character vector of labels, the same length as x, with NA where x is NA. It carries a `recode_audit` attribute: a list with `mapping` (the dictionary), `kept` (`character(0)`) and `checksum` (64-character hexadecimal SHA-256 of the `code=label` pairs).

See Also

declbld, grdrcd, audcats.

Examples

```
# codes from a survey file with their code book
race <- c(1, 2, 2, 3, NA)
book <- c("1" = "White", "2" = "Black", "3" = "Indigenous")
deccodes(race, book)

# the dictionary that was applied is kept on the result
attr(deccodes(race, book), "recode_audit")$checksum

# a code with no label is refused, never passed through
try(deccodes(c(1, 2, 9), book))

# missing codes can be refused too
try(deccodes(race, book, keepna = FALSE))
```

---

declbld *Decode a labelled import by its value labels, by code*

Description

Converts a vector read from SPSS, Stata or SAS (for example by `haven::read_sav()`), which stores categories as numeric codes with a `labels` attribute, into a factor of the labels. Each code is looked up by its value, so neither the order of the codes nor the alphabetical order of the labels can affect which label a row gets. Codes with no label are refused. The factor's levels are in code order, which keeps the source program's reference category.

Usage

```
declbld(x, vlabs = NULL)
```

Arguments

`x`	A vector with a `labels` attribute (as produced by haven), or a plain vector of codes when `vlabs` is supplied. A factor, ordered or not, is read through its integer codes; the result is an unordered factor whose levels follow the codes.
`vlabs`	Optional named character vector, code = label, used only when `x` carries no `labels` attribute. Default `NULL`.

Details

When `x` has a `labels` attribute (a named vector, label = code) it is used and `vlabs` is ignored; otherwise `vlabs` (code = label) is required. Decoding is done by `deccodes`. The levels are the labels ordered by their numeric code (codes that are not numbers are ordered after the numeric ones, by string), and include labels of codes that do not occur.

Errors: a `labels` attribute without names; no attribute and no `vlabs`; a code with no label.

Value

A factor whose levels are the labels in code order (not alphabetical), with a `recode_audit` attribute as described in `grdrcd`.

See Also

`deccodes`, `relbl`, `xfrvrf`

Examples

```
x <- structure(c(1, 2, 2, 4),
               labels = c(White = 1, Black = 2, Other = 3, Unknown = 4))
declbld(x)

# plain codes with a code book; levels follow the codes 1, 2, 10
declbld(c(2, 1, 10), vlabs = c("10" = "Ten", "2" = "Two", "1" = "One"))

# codes alone are not categories
try(declbld(c(1, 2, 2)))
```

`dgstobj` *Digest an arbitrary R object*

Description

Fingerprints any R object by hashing its serialization, so a list, a data frame, a function or a fitted model all get a stable digest. The counterpart of `digest::digest()`, computed with this package's own hashes.

Serialization is pinned to version 2 with XDR byte order, so the digest is the same on a big-endian machine as on a little-endian one. Two objects that are `identical()` produce the same digest; two that merely print the same need not, because attributes are part of the serialization.

Usage

```
dgstobj(x, algo = c("sha256", "sha512", "blake2b", "crc32"), key = NULL)
```

Arguments

`x`	Any R object.
`algo`	`"sha256"` (default), `"sha512"`, `"blake2b"` or `"crc32"`.
`key`	Optional key (a single string or raw vector), default `NULL`. Only for `"sha256"` (HMAC-SHA-256) and `"blake2b"` (keyed BLAKE2b): gives a keyed fingerprint that only a key holder can reproduce.

Details

The object is serialized with `serialize(x, NULL, version = 2, xdr = TRUE)` and the bytes are hashed with `csha256()`, `csha512()`, `cblake2b()` (32 bytes) or `ccrc32()`. CRC-32 is a checksum for accidental corruption, not a cryptographic hash. With a `key` the digest is HMAC-SHA-256 for `"sha256"` and keyed BLAKE2b for `"blake2b"`; a key with `"sha512"` or `"crc32"` is an error rather than a silently unkeyed digest. Objects that contain environments (functions, many fitted models) serialize their enclosing environments too, so their digest can change between sessions even when the code does not.

Value

A length-1 character vector of lowercase hex (64 characters for `"sha256"` and `"blake2b"`, 128 for `"sha512"`), or a length-1 numeric for `"crc32"`.

See Also

`csha256()` for hashing text or bytes directly, `jsser()` for a readable form that keeps types and attributes.

Examples

```
dgstobj(list(a = 1L, b = "x"))

# Stable across calls, and sensitive to any change.
identical(dgstobj(1:10), dgstobj(1:10))
dgstobj(1:10) == dgstobj(1:11)

# Attributes are part of the object, so they are part of the digest.
dgstobj(matrix(1:6, nrow = 2)) == dgstobj(1:6)

# Any of the hashes, and a keyed fingerprint.
dgstobj(mtcars, algo = "sha512")
dgstobj(mtcars, algo = "crc32")
dgstobj(mtcars, algo = "blake2b", key = "secret")

# A data frame's digest pins the data, so it can go in a manifest.
dgstobj(data.frame(x = 1:3))

# a key with an algorithm that has no keyed form is refused
try(dgstobj(1:3, algo = "crc32", key = "secret"))
```

---

dispmix — *The mix of outcomes over time, with simultaneous intervals and trend tests*

Description

For a table of counts – periods in rows, outcomes in columns, such as cases withdrawn, pleaded guilty, or tried – gives each outcome's share of its period with Goodman's simultaneous confidence intervals, tests whether the mix is the same in every period, and tests each outcome for a linear trend in its share.

Usage

```
dispmix(cnts, prd = NULL, scores = NULL, conflvl = 0.95)
```

Arguments

`cnts`	A matrix or data frame of non-negative counts, one row per period and one column per outcome. Columns that are not numeric are refused; pass only the outcome columns.
`prd`	Optional period labels (default: the row names, or `1:nrow(counts)`).
`scores`	Numeric scores for the periods in the trend tests. Default: `1:nrow(counts)`, i.e. equally spaced.
`conflvl`	Confidence level. Default 0.95.

Details

Goodman's intervals hold for all k shares of a period at once: with A the upper α/k point of the chi-square distribution on one degree of freedom, the limits for a share n_i/N are

$$\frac{A + 2n_i \pm \sqrt{A(A + 4n_i(N - n_i)/N)}}{2(N + A)}.$$

The trend test for each outcome is the Cochran-Armitage test of its count against the period totals (`stats::prop.trend.test()`); the Holm adjustment covers the k tests. Counts of court cases are large, so these tests detect small changes; the shares and their intervals show whether a change is large enough to matter.

Errors: `cnts` not a numeric matrix or data frame, with a non-numeric column, fewer than one period or two outcomes, or a negative, missing or infinite count; a period with no cases; `scores` not one finite number per period, or all equal; `prd` of the wrong length or with repeats; `conflvl` outside (0, 1).

Value

A list of class `rmbl_disposition_mix`: `shares`, a long data frame (`period`, `outcome`, `count`, `total`, `share`, `lower`, `upper`); `homogeneity`, the Pearson chi-square test that every period has the same mix (with Cramer's V); `trend`, one row per outcome with the Cochran-Armitage trend statistic, its p-value, the Holm-adjusted p-value, and the change in share from the first to the last period.

References

Goodman, L. A. (1965). On simultaneous confidence intervals for multinomial proportions. *Technometrics*, 7(2), 247-254.

Armitage, P. (1955). Tests for linear trends in proportions and frequencies. *Biometrics*, 11(3), 375-386.

Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2), 65-70.

See Also

`shrs` for one period's shares with Wilson intervals.

Examples

```
d <- read.csv(system.file("extdata", "ocj_dispositions.csv",
                          package = "rmoriebricklayer"))
mix <- dispmix(d[, -(1:2)], prd = d$year)
mix
plot(mix)

# the parts of the result
head(mix$shares)
mix$trend
```

```
# years as trend scores, so the gap where no row exists is respected
dispmix(d[, -(1:2)], prd = d$year, scores = d$year)$trend[, 1:3]

# outcome columns only: a text column is refused
try(dispmix(data.frame(year = c("a", "b"), x = 1:2, y = 3:4)))
```

`dldata` *Download a File*

Description

Downloads one file to a path on disk through the package's own transport, `bldl()`, and returns the path. It is the plain download step of a data pipeline: resolve a URL (for example with `rslvckan()`), download it with this function, then check it with `vrfsha256()`. Use `fdl()` instead when a person should see advice about the failure and a Wayback Machine fallback.

Usage

```
dldata(url, tpath, mode = "wb", quiet = FALSE, allowfile = FALSE)
```

Arguments

`url`	The URL to download, a single string.
`tpath`	Destination path on disk, a single string. Its directory must exist.
`mode`	Kept for the signature of `utils::download.file()`; only `"wb"` (the default) or `"w"` is accepted. Every transfer is binary and written fresh through the package's own transport, so an append mode cannot be honoured and is an error rather than a silent `"wb"`.
`quiet`	Logical. `TRUE` suppresses the progress output; `FALSE` (the default) shows it.
`allowfile`	Logical. Accept a `file://` URL (default `FALSE`; offline tests and examples pass `TRUE`). Every URL must otherwise be `https` and public.

Details

Before anything is fetched the URL must be one public `https` URL: plain `http` needs the option `rmoriebricklayer.allowhttp` set to `TRUE`, `file://` needs `allowfile = TRUE`, and loopback, private, link-local and cloud-metadata addresses are refused. The transport then resolves the host, pins the connection to the checked addresses, re-checks every redirect hop, caps the body at 2 GiB and allows 3600 seconds for the transfer. The file at `tpath` is overwritten. Errors from the transfer (network, HTTP status, size cap) propagate as R errors; nothing partial is left behind.

Progress is drawn on stderr while `quiet = FALSE`, whatever the `morie.quiet` option says, because the value is passed to the transport explicitly.

Errors: `url` or `tpath` not a single non-empty string; a refused URL; `mode` other than `"wb"` or `"w"`.

Value

The tpath, returned invisibly.

Examples

```
# offline: a local file through a file:// URL
src <- tempfile(fileext = ".csv")
writeLines(c("id,value", "1,10", "2,20"), src)
dest <- dldata(paste0("file://", src), tempfile(fileext = ".csv"),
  quiet = TRUE, allowfile = TRUE)
read.csv(dest)

# file:// is refused unless asked for, and so is an append mode
try(dldata(paste0("file://", src), tempfile()))
try(dldata("https://example.org/x.csv", tempfile(), mode = "a"))


# needs the network
dest <- dldata("https://cloud.r-project.org/",
  tempfile(fileext = ".html"),
  quiet = TRUE
)
file.exists(dest)
```

---

dlogin *Sign in for the curated tables at data.rmorie.com*

Description

Stores your personal MORIE data key in $XDG_CONFIG_HOME/morie/credentials.json (owner-only; the file the morie, rmorie and rmoriedata packages read too, so one sign-in serves all of them). The key opens the curated tables (dload); it is not used for language models, which run on your own key or a local model (llmask). Keys are personal: the GitHub or emailed-code sign-in issues one, or ask at www.rmorie.com/access/. The sign-in addresses come from svcs. Three ways in:

- token: a key you already have, from the website or an email. It is checked with the service first and stored only when it is accepted; a refused key, or no answer, stores nothing.
- email: a one-time code is sent to the address (valid ten minutes, single use); you type it, or pass code.
- neither: the GitHub device flow; a code is shown to enter at github.com, and the key arrives once you approve.

Nothing is written except by this explicit call.

Usage

```
dlogin(
  token = NULL,
  email = NULL,
  code = NULL,
  openbro = interactive(),
  pollmaxsec = 600,
  accter = FALSE,
  name = NULL,
  affil = NULL,
  purp = NULL,
  restr = NULL,
  supnm = NULL,
  suprole = NULL,
  supemail = NULL
)
```

Arguments

token	Character scalar or NULL (the default): a key you already have, issued at www.rmorie.com/access/. Surrounding spaces are removed. Takes precedence over email.
email	Character scalar or NULL (the default): sign in with a one-time code sent to this address. Lower-cased before use.
code	Character scalar or NULL (the default): the emailed code when you already have it; NULL sends a new code and asks for it at the console. Used only with email.
openbro	Logical scalar, default interactive(): open the GitHub page for the device flow in the system browser (only a public https address is opened).
pollmaxsec	Numeric scalar, default 600: how many seconds the device flow waits for you to approve the sign-in.
accter	Logical scalar, default FALSE: TRUE accepts the data access terms (read them at data.rmorie.com/terms) without the console asking. A new key is issued only once they are accepted, here or at the prompt.
name	Character scalar or NULL: your name (asked for when NULL). Required for a new key.
affil	Character scalar or NULL: your university, employer or organisation (optional; asked for when NULL).
purp	Character scalar or NULL: why you need the data, a sentence or two (asked for when NULL). Required for a new key.
restr	Logical scalar or NULL: TRUE also accepts Schedule B of the terms, which opens the restricted tables (ICPSR, NCVS 38604, A2AJ case law); NULL asks at the console.
supnm, suprole, supemail	Character scalars or NULL: the supervisor, department chair, director or coordinator who knows of your use, their role and their email. Required with restricted = TRUE; asked for when NULL. The role is one of "supervisor", "chair", "director", "coordinator" or "other".

Details

The credentials file is edited as text: only the three members this package owns (`hosted_key`, `hosted_user` and `hosted_base_url`, the address the key was issued for) are set, and every other byte stays, so entries written by the other packages survive. The file is written through a fresh temporary name, created owner-only (mode 0600, folder 0700 on Unix) before the key is written. A file that is not a JSON object, is not valid JSON or names a member twice is refused rather than overwritten. `XDG_CONFIG_HOME` moves the file; when unset it is `~/.config/morie/credentials.json`.

The sign-in service address is the signed services document's `data.auth_url` (see `svcs`), or the environment variable `MORIE_HOSTED_AUTH_URL` when set (it must be a public `https` address). The device flow polls every `interval` seconds the service names (5 by default), prints a progress line about every 30 seconds, and stops with an error after `pollmaxsec`.

Errors: an empty `token`; a `token` the data service does not accept, or no answer from it (nothing is stored); an `email` that is empty or not of the form `name@domain.tld`; no code typed at the prompt; a refused code; a sign-in service that answers with an error or returns no key; a credentials file that cannot be edited safely.

Value

The key (character scalar), invisibly. A message names the file it was stored in.

See Also

`dlogout` to remove the key.

Examples

```
old <- Sys.getenv("XDG_CONFIG_HOME", unset = NA)
Sys.setenv(XDG_CONFIG_HOME = tempfile()) # never touch your real credentials

# input checks that need no network
try(dlogin(token = "   "))
try(dlogin(email = "not-an-email"))

if (is.na(old)) {
  Sys.unsetenv("XDG_CONFIG_HOME")
} else {
  Sys.setenv(XDG_CONFIG_HOME = old)
}


# these contact the sign-in service
dlogin(token = "<key issued at https://www.rmorie.com/access/>")
dlogin(email = "you@example.com")
dlogin(email = "you@example.com", code = "123456")
dlogin() # GitHub device flow; asks for the terms, name and reason
# everything given up front, for a script
dlogin(accter = TRUE, name = "Ada Lovelace",
                      purp = "Course project on court delays in Ontario.")
```

dlogout *Forget the MORIE data key*

Description

Removes the MORIE data key that dlogin stored, so this machine can no longer load the curated tables at data.rmorie.com until you sign in again. Use it on a shared machine, or to switch to another key. Nothing is contacted.

Usage

```
dlogout()
```

Details

The shared credentials file is credentials.json in the folder $XDG_CONFIG_HOME/morie, else in ~/.config/morie. It is edited as text. Three members are removed: hosted_key, hosted_user and hosted_base_url. Every other entry (written by morie, rmorie or rmoriedata) is kept byte for byte. When nothing else is left the file is deleted. A key set in the environment variable MORIE_HOSTED_KEY is not touched. A message says whether a key was removed.

Errors when the file is not a JSON object (or not valid JSON, or names a member twice), and when the key cannot be removed without damaging the file; in both cases the file is left as it was.

Value

Logical scalar, invisibly: TRUE when a key was removed, FALSE when none was stored.

See Also

dlogin.

Examples

```
old <- Sys.getenv("XDG_CONFIG_HOME", unset = NA)
Sys.setenv(XDG_CONFIG_HOME = tempfile()) # a scratch credentials folder

# nothing stored yet
dlogout()

# a file holding a key and another tool's entry: only the key goes
d <- file.path(Sys.getenv("XDG_CONFIG_HOME"), "morie")
dir.create(d, recursive = TRUE)
writeLines(
  '{"hosted_key": "k-123", "other_tool": 1}',
  file.path(d, "credentials.json")
)
dlogout()
readLines(file.path(d, "credentials.json"))
```

```
# a damaged file is refused, not overwritten
writeLines("not json", file.path(d, "credentials.json"))
try(dlogout())

if (is.na(old)) {
  Sys.unsetenv("XDG_CONFIG_HOME")
} else {
  Sys.setenv(XDG_CONFIG_HOME = old)
}
```

`dmnf` *Curated datasets at data.rmorie.com*

Description

The MORIE project keeps databases materialised from Google BigQuery public datasets (Chicago crime, EPA air quality, US census, FEC, FDA, NOAA, NHTSA, Hacker News, Ethereum, World Bank, ...) and serves their tables from the edge, under the data access terms at `www.rmorie.com/data-license/`. They open with the key `dlogin` stores, the same key rmorie and morie use; keys are personal: sign in with GitHub or an emailed code, or ask at `www.rmorie.com/access/`. The service address comes from `svcs`. `dmnf()` returns the gateway's manifest (every table with rows, columns, size, SHA-256 and its BigQuery source), kept for a day; `dtabs()` the same as a data frame; `dload("db/table")` one table as a data frame, cached under `tempdir()` (or the directory in `options(rmoriebricklayer.dcache = )`) so later calls are local. From the shell: `rmoriebricklayer data list` and `rmoriebricklayer data pull db/table [--out FILE.csv]`.

Usage

```
dmnf(refresh = FALSE)

dtabs(refresh = FALSE)

dload(key, refresh = FALSE, token = NULL)
```

Arguments

`refresh`	Logical scalar, default `FALSE`: fetch again even when a day-old copy (of the manifest, or for `dload()` of the table) is cached.
`key`	Character scalar, a `db/table` key from the manifest (the `key` column of `dtabs()`).
`token`	For `dload()`: the MORIE key to use for this call, in place of the one `dlogin` stored; `NULL` (the default) uses the stored key. It is never printed or written to disk.

Details

The manifest is cached as `manifest.json` in the cache folder and fetched again when it is more than a day old or `refresh = TRUE`. A reply that is not a manifest (not a JSON object with a list of datasets each carrying a `key`), or one that lists no tables while the cached copy lists some, is refused and the cached copy kept.

`dload()` downloads `db/table.csv.gz` once into the cache folder (as `db__table.csv.gz`) and checks its SHA-256 against the manifest before every use, cached or not: a file that does not match is deleted and refused (after one refresh of the manifest, since a table rebuilt upstream has a new digest). When the manifest cannot be fetched, the last copy on disk, however old, supplies the digest; with none, a fresh download is not kept and a cached file is kept but not used. The CSV is read in compiled code with the same result as `read.csv(gzfile(f), stringsAsFactors = FALSE)`, two malformed shapes apart, which are refused with the record they are in where `read.csv()` changes the data without a word: a record with more fields than the header (`read.csv()` turns the first column into row names) and a quote mark inside an unquoted field (`read.csv()` takes the rest of the file as one quoted run). Tables that inflate past 512 MiB are read by `read.csv()` itself, a piece at a time. A table whose SHA-256 matches the manifest is the file the service holds, so it is read whatever its size; one the manifest lists without a digest may decompress to at most the option `rmoriebricklayer.maxcsvbytes` (1 GiB by default). A large table needs memory for all of it: a 1 GB decompressed CSV takes a few GB of RAM as a data frame.

The key is sent only to the address of the signed services document. With `MORIE_DATA_URL` set to another address the key is not sent and the download fails with a message saying so. Some tables (ICPSR studies, the NCVS 38604 extract, the A2AJ case law) open only after their terms are accepted on the terms page, `data.rmorie.com/terms`. Downloads draw a progress bar when `options(morie.progress)` allows it.

Errors: `refresh` not `TRUE` or `FALSE`; no key stored (the message says how to sign in); the service unavailable, or the key rejected; a `key` that is not of the form `db/table`, or not listed in the manifest; a `token` that is not one non-empty string; a digest mismatch; a restricted table whose terms were not accepted.

Value

`dmnf()`: the manifest as a list (element `datasets`, one list per table with at least `key`, and as the service provides them `rows`, `columns`, sizes, the SHA-256 of the compressed file and the BigQuery `source`); `dtabs()`: a data frame with `key`, `name` (the description, else the source, else the key), integer `rows` and `source` (class `dtabs`, which prints one left-aligned line per table, cut to the console width); `dload()`: the table as a data frame.

See Also

dlogin, svcs.

Examples

```
# Offline, against a cache folder holding a manifest and one table made
# here (no key is stored and nothing is fetched).
oldenv <- Sys.getenv(c("XDG_CONFIG_HOME", "MORIE_HOSTED_KEY"), unset = NA)
Sys.setenv(XDG_CONFIG_HOME = tempfile())
Sys.unsetenv("MORIE_HOSTED_KEY")
```

```
cache <- file.path(tempdir(), "data-hub-example")
dir.create(cache)
old <- options(rmoriebricklayer.dcache = cache)
f <- file.path(cache, "demo__counts.csv.gz")
con <- gzfile(f, "w")
write.csv(
  data.frame(region = c("a", "b"), n = c(3L, 5L)),
  con,
  row.names = FALSE
)
close(con)
m <- list(
  datasets = list(list(
    key = "demo/counts",
    rows = 2,
    source = "an example",
    meta = list(description = "Counts by region"),
    sha256_gz = csha256(readBin(f, "raw", file.size(f)))
  ))
)
writeLines(tojs(m, auto_unbox = TRUE), file.path(cache, "manifest.json"))

dtabs()
dload("demo/counts")

# keys are checked against the manifest
try(dload("counts"))
try(dload("demo/other"))

# a cached file that no longer matches its digest is deleted and refused
writeBin(as.raw(1:10), f)
try(dload("demo/counts"))
file.exists(f)

options(old)
unlink(cache, recursive = TRUE)
for (v in names(oldenv)) {
  if (is.na(oldenv[[v]])) {
    Sys.unsetenv(v)
  } else {
    do.call(Sys.setenv, as.list(oldenv[v]))
  }
}


# the real service: needs a key and the network
dlogin(token = "sk-...")  # a key issued at www.rmorie.com/access/
head(dtabs())
df <- dload("chicago_crime/incidents")
```

drbggen *Draw bytes from a DRBG*

Description

Generates n bytes (SP 800-90A's Generate function) and advances the generator. Use it to draw reproducible random bytes from a generator made by drbgnew(): the same seed and the same sequence of calls give the same bytes on every platform, which is what known-answer tests and reproducible key generation need.

Usage

```
drbggen(drbg, n, addl = NULL)
```

Arguments

drbg	A generator from drbgnew().
n	Integer scalar, the number of bytes, 1 to 65536 (the standard's limit per request for AES: 2^19 bits).
addl	Optional additional input of up to 48 bytes, raw or hex, mixed into the state before and after the output. Default NULL (none).

Details

Each call encrypts the counter V under Key with AES-256 to produce the output, then updates (Key, V) with the additional input, so the next call gives different bytes and earlier outputs cannot be recovered from the new state. The generator is changed in place. When the generator was created or last reseeded in another process (a fork, or a copy restored from a file), it first reseeds itself from the operating system. After 2^{48} requests since seeding the standard requires a reseed, and the call errors until drbgrsd() is called.

Errors: drbg not made by drbgnew(); n not a single whole number from 1 to 65536; addl longer than 48 bytes or neither raw nor hex.

Value

A raw vector of n bytes.

See Also

drbgnew(), drbgrsd().

Examples

```
d <- drbgnew(as.raw(0:47))
x <- drbggen(d, 16)
y <- drbggen(d, 16)
identical(x, y)  # the state moved on

# additional input changes the output and the state
e <- drbgnew(as.raw(0:47))
identical(drbggen(e, 16, addl = "00ff"), x)

# random integers in 1..6 from the bytes (rejection keeps them uniform)
r <- as.integer(drbggen(drbgnew(as.raw(0:47)), 64))
head(r[r < 252] %% 6 + 1, 10)

# one request is at most 65536 bytes
try(drbggen(d, 70000))
```

---

drbgnew *A deterministic random bit generator: AES-256 CTR_DRBG (NIST SP 800-90A)*

Description

Creates the counter-mode DRBG of NIST SP 800-90A Rev. 1, section 10.2.1, with AES-256 and no derivation function, implemented in this package (no system library). From the same entropy, personalization and additional inputs it returns the same bytes, which is what reproducible key generation and known-answer tests need; for keys meant to stay secret, seed it from `rndbytes()` (the default) or use `rndbytes()` directly.

Usage

```
drbgnew(entropy = NULL, person = NULL)
```

Arguments

`entropy`	Entropy input: exactly 48 bytes, as a raw vector or one string of 96 hexadecimal digits. `NULL` (the default) draws them from the operating system's CSPRNG through `rndbytes()`.
`person`	Optional personalization string of up to 48 bytes, raw or hex; `NULL` (the default) is the empty string. Different personalization strings give independent streams from the same entropy.

Details

It reproduces all 720 AES-256 no-df vectors of NIST's DRBG validation suite (without reseeding, with reseeding, and with prediction resistance; the package's tests check a sample of them), and it is the `randombytes()` of NIST's `rng.c`, with which most post-quantum submissions wrote their

known-answer files: `drbgnew(as.raw(0:47))` followed by `drbggen(d, 48)` gives the seed of their vector 0. AES runs in constant time: the S-box is computed (an inversion in GF(2^8) and the affine map) rather than looked up, and on x86-64 processors with AES-NI the rounds use those instructions. Both paths are checked under valgrind memcheck with the entropy input marked secret (`inst/ctcheck`, case `drbg-portable` and `drbg-aesni`).

The generator is an object that changes as it is used: every `drbggen()` and `drbgrsd()` advances it in place.

The generator is an environment, so copies share one state. It is not meant to be saved: a generator restored from `saveRDS()` or inherited by a forked process reseeds itself from the operating system on its next use, so two processes never replay the same stream (and such a stream is then no longer reproducible). Errors when `entropy` is not exactly 48 bytes or `person` is longer than 48 bytes, or either is neither a raw vector nor one hexadecimal string.

Value

An object of class `bricklayer_drbg` (an environment holding the 32-byte Key, the 16-byte V, the request count and the process id). Its state (Key, V) is not printed.

Security

The implementation is hand-written. Its standardised schemes are checked byte for byte against OpenSSL 3.5 and NIST known-answer vectors in the test suite. Side channels are checked, not assumed: `inst/ctcheck` runs every operation that touches a secret under valgrind memcheck with the secret marked undefined (the ctgrind method), so a branch or a memory address that depends on it is a reported error, and afterwards scans the dead stack for copies of the secret. Both checks run in CI on every change, with GCC and with Clang. Timing is also measured on x86-64 and arm64 hardware (`inst/dudect`) and power leakage is assessed in simulation under the value and the transition models (`inst/tvla`) for the masked ML-KEM and ML-DSA kernels; this scheme is not masked. These are checks of this code on those compilers; the README's security section lists what each check does and does not cover.

References

NIST SP 800-90A Rev. 1 (2015). Recommendation for Random Number Generation Using Deterministic Random Bit Generators, section 10.2.1. DOI 10.6028/NIST.SP.800-90Ar1

NIST FIPS 197 (2001, updated 2023). Advanced Encryption Standard (AES).
DOI 10.6028/NIST.FIPS.197-upd1

See Also

`drbggen()`, `drbgrsd()`, `rndbytes()`.

Examples

```
# NIST's rng.c, as the post-quantum known-answer files use it: vector 0's
# seed
d <- drbgnew(as.raw(0:47))
drbggen(d, 48)
```

```
# the same inputs give the same bytes; a personalization string separates
# streams
a <- drbgnew(as.raw(1:48), person = charToRaw("session 1"))
b <- drbgnew(as.raw(1:48), person = charToRaw("session 2"))
identical(drbggen(a, 16), drbggen(b, 16))

# seeded from the operating system
d <- drbgnew()
length(drbggen(d, 32))

# the entropy must be exactly 48 bytes
try(drbgnew(as.raw(1:16)))
```

---

drbgrsd *Reseed a DRBG*

Description

Mixes fresh entropy into the generator (SP 800-90A's Reseed function) and resets its request count. Use it after many requests (the standard requires a reseed after 2^{48}), to add prediction resistance before a sensitive draw, or to replay a known-answer test that reseeds with given entropy.

Usage

```
drbgrsd(drbg, entropy = NULL, addl = NULL)
```

Arguments

drbg	A generator from drbgnew().
entropy	Entropy input: exactly 48 bytes, raw or hex; NULL (the default) draws them from the operating system's CSPRNG.
addl	Optional additional input of up to 48 bytes, raw or hex; default NULL (none).

Details

The new (Key, V) is derived from the old state, the 48 entropy bytes and the additional input, so the stream after the reseed depends on everything that went before. The request count is reset to 1 and the generator is marked as belonging to the current process. Errors: drbg not made by drbgnew(); entropy not exactly 48 bytes; addl longer than 48 bytes; either not raw or hex.

Value

drbg, invisibly, reseeded in place.

See Also

drbgnew(), drbggen().

Examples

```
d <- drbgnew(as.raw(0:47))
drbgrsd(d, as.raw(48:95))
drbggen(d, 8)

# a reseed with given entropy is reproducible
d2 <- drbgnew(as.raw(0:47))
drbgrsd(d2, as.raw(48:95))
identical(drbggen(d2, 8),
          drbggen(drbgrsd(drbgnew(as.raw(0:47)), as.raw(48:95)), 8))

# fresh entropy from the operating system, with additional input
drbgrsd(d, addl = charToRaw("before key generation"))

# the entropy must be exactly 48 bytes
try(drbgrsd(d, as.raw(1:10)))
```

`drftapp` *Drift report as a Shiny app*

Description

The report of `drfthtml()` as a Shiny app, for working through a drift result rather than handing it on. `drftapp(x)` opens on a `cpsdrft()` result; `drftapp()` with no result opens on two file inputs, the reference and the current CSV, and compares them when both are uploaded. It needs the shiny package, which is suggested, not required.

Usage

```
drftapp(
  x = NULL,
  ref = NULL,
  cur = NULL,
  hist = NULL,
  warn = NULL,
  fail = NULL
)
```

Arguments

`x`	A `cpsdrft()` result, or `NULL` (the default) to upload two CSV files in the app.
`ref`, `cur`	Optional: the two data frames `x` was computed from, for the rerun panel.
`hist`	Optional: a directory of earlier drift JSON files, for the history tab.
`warn`, `fail`	The PSI bands the page opens with; `NULL` (the default) takes the object's `bands`.

Details

The app has the column table (verdicts under the bands), the PSI per column, the fingerprint, and for the chosen column its summary, the distributions, the ECDFs, the quantile shift, the first digits and a sentence on the verdict, as `drfthtml()` has, and a history heatmap with `hist`; the warn and fail bands and alpha are sliders, with a reset to the object's bands. Beyond the page it has: "what would clear this", the PSI of the column with its `k` largest bins left out and the rest renormalized; with `ref` and `cur` (or uploaded CSV files), a rerun of `cpsdrft()` on chosen columns with another number of bins; with `hist`, the PSI of each column over the runs in that directory (`rmbl drift --json` outputs or JSON exports of the page); a manifest tab when `x$manifest` holds an `mnfdiff()`; and downloads of the HTML report, Markdown and JSON. Uploaded files are read with the package's own CSV reader (the one `dload()` uses). `rmbl drift PATH --app` opens the app on the first capsule whose source moved.

The function builds the app and returns it; printing it, or `shiny::runApp()`, runs it.

Errors: `x` neither `NULL` nor from `cpsdrft()`; shiny not installed; only one of `ref` and `cur`, or either not a data frame; `hist` not a directory or holding no drift JSON; `warn` above `fail`.

Value

A Shiny app object (class `shiny.appobj`).

See Also

`drfthtml()` for a file to hand on.

Examples

```
set.seed(5)
ref <- data.frame(value = stats::rnorm(400),
                  grade = sample(c("a", "b", "c"), 400, TRUE))
cur <- data.frame(value = stats::rnorm(400, 0.4),
                  grade = sample(c("a", "b", "c"), 400, TRUE))
d <- cpsdrft(ref, cur)


if (requireNamespace("shiny", quietly = TRUE)) {
  # drftapp() alone opens on two CSV files to upload
  print(drftapp(d, ref = ref, cur = cur))
}
```

`drftcal` *Calibrate the drift screens on data known not to have drifted*

Description

cpsdrft() flags a column when a Kolmogorov-Smirnov, chi-square or population-stability screen fires. How often that happens when nothing has changed is the false-alarm rate of the screen on this capsule, and it depends on the columns, their sizes and alpha. This estimates it by splitting the same data at random into two halves n times and running the screens on the halves, which is the null of "a re-fetch of identical data". A capsule whose screens fire on 30 percent of identical re-fetches needs a smaller alpha or fewer screened columns before a drift verdict means anything.

Usage

```
drftcal(data, n = 50L, alpha = 0.01, seed = 1L, ...)
```

Arguments

data	The capsule's data frame.
n	Whole number, default 50: the number of random half-splits.
alpha	Number in (0, 1), default 0.01: the significance level passed to cpsdrft().
seed	Integer or NULL, default 1: the seed for the splits.
...	Further arguments to cpsdrft(), such as psithr, psiminn, bins or idelvls.

Details

Each split draws floor(nrow(data) / 2) rows at random as the reference and uses the rest as the current table, then runs cpsdrft on the pair. A column's false-alarm rate is the share of splits on which it was flagged; anyflag is the share of splits on which any column was. Each half has half the rows of the real data, so the rates describe a comparison at that size: smaller samples make the p-value screens less sensitive, and the PSI screen only counts once both halves reach psiminn rows.

The splits are drawn under seed and the caller's random number stream is restored afterwards (seed = NULL uses and advances the current stream instead). Run time is about n times one cpsdrft() call; a progress bar is shown in an interactive session.

Errors: data not a data frame with at least four rows; n not a whole number of at least 1; any error from cpsdrft().

Value

A list of class rmbl_drift_calibration: columns (a data frame of column and falsealarmrate, the share of splits on which it was flagged), anyflag (share of splits with at least one flag), alpha, n, and alphafam (alpha divided by the number of columns: the Bonferroni per-screen level that would hold the family-wise false-alarm rate at alpha). The print method lists the columns that fired.

See Also

cpsdrft, scanadj.

Examples

```
set.seed(1)
d <- data.frame(a = rnorm(400), b = sample(letters[1:4], 400, TRUE))
drftcal(d, n = 10)

# a stricter level, and the per-column rates
cal <- drftcal(d, n = 10, alpha = 0.001)
cal$columns

try(drftcal(d[1:3, ]))
```

`drftchisq` *Chi-square test for a categorical column*

Description

Tests whether the category counts of a new sample agree with a reference distribution that is taken as known, such as proportions fixed by a specification or a published breakdown. It is Pearson's chi-square goodness-of-fit test. When the reference is itself a sample, use `drfthomog()`, which accounts for the reference's own sampling error.

Usage

```
drftchisq(obs, expd)
```

Arguments

`obs`	Named numeric vector of counts in the new sample, or a factor/character vector to be tabulated.
`expd`	Named numeric vector of reference counts or proportions, or a factor/character vector to be tabulated. Rescaled to the total of `obs`.

Details

Pearson's chi-square goodness-of-fit statistic comparing observed category counts with the proportions a capsule was pinned against. Use it where `drftks()` cannot apply, because the column is a factor or a set of codes rather than a number.

Categories present in one argument and not the other are aligned by name, so a vanished category registers as a shortfall against its expected count.

A category that OCCURS but which the reference gives probability zero contradicts the pinned distribution outright, and its chi-square term is unbounded; `statistic` is then `Inf` and `p_value` is `0`. If a new category is a legitimate possibility rather than a contradiction – which it usually is when the reference is itself a finite sample – use `drfthomog()` instead.

With O_j the observed counts and $E_j = Ne_j / \sum e$ the reference scaled to the observed total N,

$$X^2 = \sum_j \frac{(O_j - E_j)^2}{E_j}$$

over the categories with $E_j > 0$, on $k - 1$ degrees of freedom. The p-value is the upper tail of the chi-square distribution, computed directly (not as one minus the lower tail), so it does not round to zero early. With a single category there is nothing to test: `df` is 0 and both p-values are `NA`. The chi-square approximation is poor when expected counts are below about 5; `stats::chisq.test()` with `simulate.p.value = TRUE` is an alternative there.

Errors: a numeric vector without names; negative, infinite or missing counts; a total of zero on either side.

Value

A named length-4 numeric: `statistic`, `df`, `p_value` and `logpvalue`, the natural log of the upper tail, which keeps its information where `p_value` underflows to 0. It has class `c("rmbl_drftchisq", "rmbl_measure", "numeric")`, so it prints as a box with its inputs (kept in attribute `"lakner"`); arithmetic, `[`, `c()` and `as.numeric()` give plain numbers.

See Also

`drfthomog()`, `cpsdrft()`, `stats::chisq.test()` for the full test object.

Examples

```
ref <- c(a = 50, b = 30, c = 20)

# Counts matching the reference proportions: nothing to report.
drftchisq(c(a = 100, b = 60, c = 40), ref)

# A reallocated mix is detected.
drftchisq(c(a = 40, b = 60, c = 100), ref)

# Agrees with stats::chisq.test().
o <- c(a = 40, b = 60, c = 100)
all.equal(drftchisq(o, ref)[["statistic"]],
          as.numeric(stats::chisq.test(o, p = ref / sum(ref))$statistic))

# Raw vectors are tabulated for you.
drftchisq(c("a", "a", "b", "b"), c("a", "b"))

# A category the reference rules out, but which occurs, is a flat
# contradiction rather than a large finite statistic.
drftchisq(c("a", "a", "b", "d"), c("a", "a", "b", "b"))

# When a new category is legitimate, compare two samples instead.
drfthomog(c("a", "a", "b", "b"), c("a", "a", "b", "d"))
```

drfthomog *Chi-square test of homogeneity for two categorical samples*

Description

Tests whether two samples of a categorical variable (for example race, offence type or region in an old and a new data extract) were drawn from the same distribution, by Pearson's chi-square on the 2-by-k contingency table of their counts. Use it to ask whether the mix of categories has changed between two samples; it is the categorical test that `cpsdrft()` runs.

Usage

```
drfthomog(x, y, seed = 1L)
```

Arguments

x, y	Factor or character vectors (or named count vectors) – the reference and the new sample.
seed	Integer, default 1: seed for the Monte Carlo branch (small expected counts). The draw is made under this seed and the caller's RNG stream is restored, so the same inputs give the same p-value and nothing downstream moves.

Details

With pooled shares $\hat{\pi}_j = (x_j + y_j)/(n_x + n_y)$ and expected counts $n_x\hat{\pi}_j$ and $n_y\hat{\pi}_j$,

$$X^2 = \sum_j \frac{(x_j - n_x\hat{\pi}_j)^2}{n_x\hat{\pi}_j} + \frac{(y_j - n_y\hat{\pi}_j)^2}{n_y\hat{\pi}_j}$$

on $k - 1$ degrees of freedom. Three branches, named in the `method` attribute:

- `"chi-square approximation"`: every expected count is at least 5; the p-value is the chi-square upper tail.
- `"Monte Carlo permutation (small expected counts)"`: some expected count is below 5; the p-value then comes from `stats::chisq.test()` with `simulate.p.value = TRUE` and `B = 10000` tables with the same margins, drawn under `seed`.
- `"inapplicable: ..."`: a single category, or fewer than two observations per category (for example two disjoint sets of dates); no test has power there, so `p_value` is `NA` rather than 1.

Errors: a numeric vector without names; negative, infinite or missing counts; an empty sample.

Value

A named length-4 numeric: `statistic`, `df`, `p_value` and `logpvalue`, the natural log of the upper tail, which keeps its information where `p_value` underflows to 0. On the Monte Carlo branch (sparse tables) `p_value` is floored at 1/(B+1) and `logpvalue` is its log, so it carries no extra range there. The attribute `method` names the branch, or says why the test was inapplicable (one category,

no counts). It has class `c("rmbl_drfthomog", "rmbl_measure", "numeric")`, so it prints as a box with its inputs (kept in attribute `"lakner"`); arithmetic, `[`, `c()` and `as.numeric()` give plain numbers.

Why this and not `drftchisq()`

`drftchisq()` compares observed counts against a distribution taken as KNOWN – proportions fixed by a specification. When the reference is itself a finite sample, that treatment ignores the reference's own sampling error, understates the variance of the comparison, and so reports drift too readily. A homogeneity test estimates the shared distribution from the pooled margins and carries the uncertainty of both samples, which is the right test when comparing a pinned extract with a fresh fetch. `cpsdrft()` therefore uses this one.

Categories present in only one sample are aligned by name and given zero counts, so an appearing or vanishing level registers.

See Also

`drftchisq()` for a known reference distribution, `stats::chisq.test()` for the full test object.

Examples

```
set.seed(1)
a <- sample(c("x", "y", "z"), 300, TRUE)
b <- sample(c("x", "y", "z"), 300, TRUE)

# Two draws from the same distribution: no evidence of a difference.
drfthomog(a, b)

# A reallocated mix is detected.
drfthomog(a, sample(c("x", "y", "z"), 300, TRUE,
                            prob = c(0.7, 0.2, 0.1)))

# Agrees with stats::chisq.test() on the 2-by-k table.
tab <- rbind(table(a), table(b))
all.equal(drfthomog(a, b)[["statistic"]],
          as.numeric(stats::chisq.test(tab)$statistic))

# It is more conservative than treating the reference as known, which
# is exactly the point.
drfthomog(a, b)[["p_value"]] >= drftchisq(b, a)[["p_value"]]

# named count vectors, and a small table that takes the Monte Carlo branch
drfthomog(c(a = 30, b = 20), c(a = 25, b = 25))
attr(drfthomog(c(a = 3, b = 1, c = 2), c(a = 1, b = 4, c = 1)), "method")

try(drfthomog(c(1, 2), c(a = 3)))
```

drfthtml *Interactive drift report as one HTML file*

Description

Writes a cpsdrft() result to a single web page for a reader who will not run R. Everything is inside the file: the stylesheet, a short script, and the binned summaries of each column as JSON. Nothing is fetched, so the page opens from disk (file://) and looks the same later.

Usage

```
drfthtml(
  x,
  file,
  title = "Drift report",
  warn = NULL,
  fail = NULL,
  open = FALSE
)
```

Arguments

x	A cpsdrft() result. Attach a provenance record as x$provenance (for example from loadprov()) and an mnfdiff() as x$manifest to show them; rmbl drift --html does both.
file	The path to write.
title	The page title, a string.
warn, fail	The PSI bands the page opens with; NULL (the default) takes the object's bands.
open	Logical: open the page in the browser when written (default FALSE).

Details

The page shows:

- a header with the overall verdict, the count of columns per verdict, the row counts, when the page was written, the package version, and the capsule's provenance (name, source, pinned SHA-256, data date) when x$provenance holds one;
- a hero canvas: the chosen column's reference and current distributions as ribbons, the band between them shaded, drawn in on load and on each new column (still when the reader asks for reduced motion);
- Columns: the table (verdict, column, type, test: KS for numbers and dates, chi-square homogeneity for categories; statistic, p-value, PSI, the change in the missing share and both sample sizes) with a search box, filters by verdict and type and sorting on any header; and the column detail for the column chosen (click a row, or the up and down keys; / goes to the search box): both samples' summary statistics (dates in ISO 8601) or the ten largest categories, a sentence on the verdict, and charts each with a line on how to read it: the density overlay with the area

between shaded (numeric) or the category flow from reference to current shares (categorical), the ECDF overlay with the largest gap marked, the quantile shift against the identity line (numeric), the PSI by bin, drawn up where the current sample has more mass, and the first digits against Benford's law with its 95% band (numeric);

- Overview: the drift fingerprint, one spoke per column at its PSI with the warn and fail bands as rings, and a tile per column with its two distributions in small;
- Manifest, when `x$manifest` holds an `mnfdiff()`: the resources before and after as a tree, and the table of digests and sizes;
- History: pick earlier `rmbl drift --json` outputs or this page's JSON exports, and the PSI of each column over the runs is drawn as a heatmap in the bands, the first crossing of the fail band outlined, and as a line per column;
- Capsule chain: provenance, pinned bytes, manifest and attestation, each verified, moved, failed or not checked (from the capsule's checks when `rmbl drift` made the report);
- sliders for the warn and fail PSI bands and for alpha, which score every verdict and chart again as they move, and a button back to the object's bands;
- exports: Markdown, JSON, each chart as SVG, and the column table copied as a GitHub comment.

Every chart is drawn at its card's width, two to a row on a wide screen and one on a narrow one, in the colours of the theme in use. The verdicts: `fail` when the test rejects at alpha, or the PSI is above the fail band with both samples at least `psiminn`; `warn` when the PSI is above the warn band; `untested` when no test applies; ok otherwise. At the object's bands `fail` is `cpsdrft()`'s `drifted`. Each verdict is written in words beside its symbol, the tables have header cells, and each chart has a text label. A light or dark theme follows the system and a button switches it. Every value is escaped. The page holds no rows of the data, only the summaries in `x$summ`, so its size grows with the number of columns and categories, not of rows.

Errors: `x` not from `cpsdrft()`; `file` not one path; `title` not one string; `warn` above `fail`.

Value

The path `file`, invisibly.

See Also

`drftapp()` for the same report as a Shiny app, plot() for the same charts in base graphics.

Examples

```
set.seed(5)
ref <- data.frame(value = stats::rnorm(400),
                  grade = sample(c("a", "b", "c"), 400, TRUE))
cur <- data.frame(value = stats::rnorm(400, 0.4),
                  grade = sample(c("a", "b", "c"), 400, TRUE))
d <- cpsdrft(ref, cur)
f <- file.path(tempdir(), "drift.html")
drfthtml(d, f, title = "Pinned against fetched")
file.exists(f)
```

```
# narrower bands open the page with more columns to look at
drfthtml(d, f, warn = 0.02, fail = 0.05)
unlink(f)
try(drfthtml(list(), f))
```

`drftks` *Two-sample Kolmogorov-Smirnov test (C backend)*

Description

Tests whether two numeric samples come from the same distribution, for example a column in a pinned data extract and the same column in a fresh download. It needs no assumption about the shape of either distribution, and it reacts to any kind of change: a shift, a change in spread, or a change in shape. It is the numeric test that `cpsdrft()` runs on every shared numeric or date column.

Usage

```
drftks(x, y)
```

Arguments

`x`, `y`	Numeric vectors, the reference and the new sample. Logical vectors are read as 0/1.

Details

The statistic is the largest vertical gap between the two empirical distribution functions, with the asymptotic two-sided p-value from the Kolmogorov distribution. Distribution-free: it assumes nothing about the shape of either sample, which is what makes it the right first test on a column whose distribution was never specified.

The p-value is the ASYMPTOTIC one, $2\sum_k (-1)^{k-1} e^{-2k^2t^2}$ with $t = \sqrt{n_{\text{eff}}}D$. It is accurate for moderate samples and conservative for small ones; for an exact small-sample p-value use `stats::ks.test()`. Ties are handled by comparing the two EDFs at each distinct value, so tied data does not produce a warning the way `ks.test()` does.

Missing values are dropped. Errors: non-numeric input (character, factor, list), any infinite value, or no non-missing value left in either sample. One value per sample is enough to compute the statistic, but the p-value is only meaningful for samples of a few dozen or more.

Value

A named length-3 numeric: `statistic` (the KS D) , `p_value`, and `neff` (the harmonic-style effective size $n_x n_y/(n_x + n_y)$). It has class `c("rmbl_drftks", "rmbl_measure", "numeric")`, so it prints as a box with its inputs (kept in attribute `"lakner"`); arithmetic, `[`, `c()` and `as.numeric()` give plain numbers.

See Also

drftpsi(), drftchisq(), cpsdrft()

Examples

```
set.seed(1)
ref <- stats::rnorm(200)

# The same distribution: a small D and a large p-value.
drftks(ref, stats::rnorm(200))

# A shifted distribution is detected.
drftks(ref, stats::rnorm(200, mean = 0.8))

# So is a change in spread alone, which a mean comparison would miss.
drftks(ref, stats::rnorm(200, sd = 2.5))

# The statistic agrees with stats::ks.test().
a <- stats::rnorm(60); b <- stats::rnorm(45, 0.6)
all.equal(drftks(a, b)[["statistic"]],
          as.numeric(suppressWarnings(stats::ks.test(a, b))$statistic))

# Identical samples have nothing to report.
drftks(ref, ref)[["statistic"]]
```

`drftpsi` *Population stability index and Jensen-Shannon divergence (C back-end)*

Description

Two effect-size summaries of how far the distribution of a numeric column has moved from a reference sample. Both samples are binned on the reference's quantiles and the two sets of bin shares are compared. Unlike a p-value they do not grow with the sample size, so they say how large a shift is rather than whether it is detectable. cpsdrft() reports both for every numeric column.

Usage

```
drftpsi(x, y, bins = 10L, eps = 1e-06)
```

Arguments

x, y	Numeric vectors, the reference and the new sample. Binned on the quantiles of x, so the reference defines the bins.
bins	Whole number, default 10: the number of quantile bins of the reference. Fewer bins are used when the reference has tied quantiles.

`eps` Floor applied to empty bins in the PSI (default 1e-6), any value in (0, 1). An empty bin contributes $(p - eps)\log(p/eps)$ to the index, so the floor is part of the statistic: the default reports the raw index, and a larger floor (0.01, 0.05) smooths an empty bin toward that share, which dampens the PSI. Compare runs only at one `eps`.

Details

Binning: values are rounded to 12 significant digits (so a formatting difference between two releases does not move tied values across an edge), the `bins + 1` quantiles of `x` are computed, duplicates are removed, and the interior quantiles become right-closed cut points with open ends at $\pm\infty$. When the reference has fewer than three distinct quantiles (a constant or nearly constant column), every distinct quantile is a cut point instead. The bin shares p_i (reference) and q_i (new) are the proportions of each sample in each bin.

The population stability index is $\sum_i (p_i - q_i)\log(p_i/q_i)$ – the symmetrised Kullback-Leibler divergence of the two discrete distributions. The conventional reading, from credit-risk monitoring where it originates, is that below 0.1 is stable, 0.1 to 0.25 warrants a look, and above 0.25 is a material shift.

The Jensen-Shannon divergence is $\frac{1}{2}KL(p\|m) + \frac{1}{2}KL(q\|m)$ with m the mixture $(p+q)/2$. Unlike PSI it is bounded – by $\log 2$ in nats – so it is comparable across columns with different numbers of bins, and it is finite even when a category is absent from one side.

Empty bins are floored at `eps` for the PSI only, since $\log(0)$ would otherwise send it to infinity on a single missing category.

Missing values are dropped. Errors: non-numeric or infinite input, an empty sample, `bins` not a whole number from 2 to 1,000,000, `eps` not strictly between 0 and 1. The PSI bands quoted above are heuristics with no calibrated null distribution; on a few hundred rows binning noise alone can exceed 0.25 (see `psiminn` in `cpsdrft()`).

Value

A named length-2 numeric: `psi` (non-negative, unbounded) and `jsdiv` (in nats, between 0 and $\log 2 \approx 0.693$). It has class `c("rmbl_drftpsi", "rmbl_measure", "numeric")`, so it prints as a box with its inputs (kept in attribute `"lakner"`); arithmetic, `[`, `c()` and `as.numeric()` give plain numbers.

References

Wu D, Olson DL (2010). Enterprise risk management: coping with model risk in a large bank. *Journal of the Operational Research Society* 61(2), 179–190. DOI 10.1057/jors.2008.144

Lin J (1991). Divergence measures based on the Shannon entropy. *IEEE Transactions on Information Theory* 37(1), 145–151. DOI 10.1109/18.61115

See Also

`drftks()`, `cpsdrft()`.

Examples

```
set.seed(2)
ref <- stats::rnorm(500)

# Same distribution: both indices near zero.
drftpsi(ref, stats::rnorm(500))

# A shift both indices register.
drftpsi(ref, stats::rnorm(500, mean = 1))

# A sample compared with itself has moved nowhere at all.
drftpsi(ref, ref)

# The Jensen-Shannon divergence is bounded by log(2), whatever the
# shift, which is what makes it comparable across columns.
drftpsi(c(1, 1, 1), c(9, 9, 9))[["jsdiv"]] <= log(2)

# a coarser binning, and a larger floor that damps an empty bin
drftpsi(ref, stats::rnorm(500, mean = 1), bins = 4)
drftpsi(ref, stats::rnorm(500, mean = 1), eps = 0.01)

try(drftpsi(ref, ref, bins = 1))
```

---

drp *Drop empty or constant columns and rows*

Description

`drpempty()` removes rows or columns that are entirely missing. `drpconst()` removes columns that hold a single distinct value. They are the counterparts of the janitor functions `remove_empty()` and `remove_constant()`. Use them after reading a spreadsheet export, which often carries blank padding rows, empty columns and columns that repeat one value (a year, a jurisdiction) on every row.

Usage

```
drpempty(data, which = c("rows", "cols"), quiet = FALSE)

drpconst(data, naasval = FALSE, quiet = FALSE)
```

Arguments

`data`	A data frame whose empty rows or columns are dropped.
`which`	`"rows"`, `"cols"`, or both (the default, `c("rows", "cols")`).
`quiet`	Logical: `TRUE` skips the boxed summary of what was dropped (rows and columns before and after, the columns removed), which is otherwise sent through `message()`; default `FALSE`.

naasval — Logical: treat NA as a distinct value, so a column of NAs plus one real value counts as two (default FALSE).

Details

A constant column carries no information and breaks anything that scales by variance, so it is worth removing – but it is also a FINDING. A column that was informative in the pinned capsule and is constant in a fresh fetch means the source changed, so check cpsdrft() before deleting it and moving on.

drpempty() removes all-NA columns first, then all-NA rows among the remaining columns, and resets the row names. An empty string is a value, not missing. drpconst() counts the distinct non-missing values of each column (with naasval = TRUE, NA counts as a value), so an all-NA column is also dropped by default. Neither function modifies anything in place.

Errors: data not a data frame (or with missing or duplicated names); which other than "rows" and/or "cols".

Value

The data frame, with the offending rows or columns removed. The names of the columns that were dropped are attached as the "dropped" attribute (a character vector, empty when none); dropped rows are not listed.

See Also

profcols(), which reports n_distinct without removing anything.

Examples

```
df <- data.frame(
  keep = c(1, 2, NA),
  allna = c(NA, NA, NA),
  constant = c(7, 7, 7),
  stringsAsFactors = FALSE
)

drpempty(df)
attr(drpempty(df), "dropped")

drpconst(df)
attr(drpconst(df), "dropped")

# Rows only.
drpempty(data.frame(a = c(1, NA), b = c(2, NA)), which = "rows")

# A column of NAs plus one value is constant by default, and not when
# NA is treated as a value of its own.
x <- data.frame(v = c(NA, NA, 5))
ncol(drpconst(x))
ncol(drpconst(x, naasval = TRUE))

# only data frames
```

```
try(drpempty(c(1, NA)))
```

---

drvkey *Derive a key from a passphrase*

Description

Turns a passphrase that a person can remember into a fixed-length key (32 bytes by default) for signing or encryption, in a way that makes guessing the passphrase slow. The same passphrase, salt and iteration count always give the same key, so the key need not be stored: it can be derived again when needed.

Usage

```
drvkey(pass, salt, iter = 100000L, length = 32L)
```

Arguments

`pass`	Passphrase, as a length-1 character or raw vector.
`salt`	Unique, non-secret salt (length-1 character or raw). Use `rndbytes()` to make one, and store it beside the key.
`iter`	Whole number, default 100000 (minimum 1): the PBKDF2 iteration count. Verification must use the same value.
`length`	Whole number from 1 to 1024, default 32: the derived key length in bytes.

Details

PBKDF2-HMAC-SHA256 (RFC 8018): stretches a passphrase into a key of full width by iterating a keyed hash, so guessing the passphrase costs `iter` times more than a single hash would.

The `salt` must be UNIQUE per key and need not be secret. Its job is to make precomputation useless: without one, a single table of common passphrases attacks every key at once.

`iter` is the cost knob. The default 100,000 is a reasonable 2020s floor for an interactive use; raise it for anything valuable, and record the value you used, since verification must repeat it exactly.

PBKDF2 resists brute force by ITERATION only, not by memory. Where a memory-hard function is available (`argon2` in **sodium**, `bcrypt_pbkdf` in **openssl**) prefer it for passwords a human chose. PBKDF2 is here because it needs nothing beyond the bundled SHA-256, so it works wherever this package works.

The derivation runs in compiled code; at the default 100,000 iterations it takes a fraction of a second, and a busy indicator is shown in an interactive session. Errors: a passphrase or salt that is not a single non-missing string or a raw vector; `iter` below 1; `length` outside 1 to 1024.

Value

A length-1 character vector: the key as lowercase hex (`2 * length` characters).

References

Moriarty K, Kaliski B, Rusch A (2017). PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018. DOI 10.17487/RFC8018

See Also

rndbytes(), chmac(), cpssign()

Examples

```
# The published PBKDF2-HMAC-SHA256 vector: "password", "salt", 1 round.
drvkey("password", "salt", iter = 1)

# Deterministic, so verification can repeat it.
identical(drvkey("pw", "s", 1000), drvkey("pw", "s", 1000))

# The salt, the passphrase and the iteration count all change the key.
drvkey("pw", "salt-a", 1000) == drvkey("pw", "salt-b", 1000)
drvkey("pw", "s", 1000) == drvkey("pw", "s", 2000)

# Use it to sign a manifest from a passphrase rather than raw bytes.
salt <- paste(format(rndbytes(16)), collapse = "")
key <- drvkey("purple otter lantern", salt)
sig <- cpssign("manifest-digest", key, scheme = "hmac")
cpsvrf("manifest-digest", sig, key)

# A longer key is a prefix-consistent extension of a shorter one.
identical(
  substring(drvkey("pw", "s", 10, length = 64), 1, 64),
  drvkey("pw", "s", 10, length = 32)
)

# a vector of passphrases is refused
try(drvkey(c("a", "b"), "salt"))
```

---

dstfit *Fit and rank candidate distributions*

Description

Fits candidate distributions to one numeric variable by maximum likelihood, ranks them by AIC, and reports for each the BIC, a Kolmogorov-Smirnov distance, an Anderson-Darling statistic and percentile bootstrap intervals for its parameters. A Hill estimate of the tail index is added for positive data. Everything is base R.

Usage

```
dstfit(x, fams = NULL, nboot = 200L, conflvl = 0.95, seed = 1L, k = NULL)

## S3 method for class 'rmbl_dstfit'
format(x, digits = 4L, ...)

## S3 method for class 'rmbl_dstfit'
print(x, ...)
```

Arguments

x	Numeric vector, the variable.
fams	Character vector of families (see Details), or NULL for every family the data allow.
nboot	Whole number, default 200: bootstrap resamples (0 for none).
conflvl	Number in (0, 1), default 0.95: the bootstrap and Hill interval level.
seed	Whole number, default 1: the seed of the bootstrap.
k	Whole number or NULL: the upper order statistics of the Hill estimate.
digits	Significant digits of the fit table's numbers.
...	For print() and plot(): see plot.rmbl_dstfit().

Details

The families and their parameters: normal (mean, sd), lognormal (meanlog, sdlog), exponential (rate), gamma (shape, rate), weibull (shape, scale), t (location, scale, df: a location-scale Student t), pareto (xmin, alpha, with xmin the smallest value, its maximum likelihood estimate), beta (shape1, shape2) for proportions, and for counts poisson (lambda), negbin (size, mu), zip (pi, lambda) and zinb (pi, size, mu), the zero-inflated forms with pi the share of structural zeros. The normal, lognormal, exponential, Pareto and Poisson estimates are closed form; the negative binomial searches its size alone (its mu estimate is the mean); the rest are maximised by optim() on a log or logit scale, Nelder-Mead then BFGS.

With fams = NULL every family the data allow is fitted: for whole numbers of 0 or more the count families; otherwise the positive families for values above 0 (exponential for values of 0 or more) and beta for values strictly between 0 and 1; and normal and t always. A family named in fams that the data do not allow is refused. A fit that fails (the optimiser stops at a non-finite likelihood) is a row with converged = FALSE and no statistics.

AIC is $2k - 2\ell$ and BIC $k \log n - 2\ell$. The Kolmogorov-Smirnov and Anderson-Darling statistics use the fitted parameters, so their usual null distributions do not apply; they rank fits, they are not tests. The bootstrap resamples the data nboot times and refits; the interval is the percentile interval at conflvl. The Hill estimator uses the k largest values (default $\lfloor\sqrt{n}\rfloor$): $\hat{\alpha} = k/\sum_{i \le k} \log(x_{(n-i+1)}/x_{(n-k)})$, with the asymptotic interval $\hat{\alpha}(1 \pm z/\sqrt{k})$.

Missing values are dropped and counted (nmiss). Errors: x not numeric, an infinite value, fewer than five values, all values equal, an unknown family or one the data do not allow. The bootstrap is seeded with seed for this call only; the session's random stream is left as it was.

Value

A list of class `c("rmbl_dstfit", "list")`:

`table` data frame, one row per family ranked by AIC: `rank`, `family`, `npar`, `loglik`, `aic`, `bic`, `ks`, `ad`, `converged` and `params` (the estimates as text).

`fits` named list of the estimates (named numeric vectors) of the converged fits.

`boot` data frame: `family`, `param`, `estimate`, `lower`, `upper` (`NA` without a bootstrap).

`hill` list: `k`, `alpha`, `lower`, `upper`, or `NULL` for data that are not all positive or fewer than ten values.

`x`, `nmiss`, `discrete`, `conflvl` the data used, the count dropped, whether the data are counts, and the level.

References

Hill BM (1975). A simple general approach to inference about the tail of a distribution. *Annals of Statistics* 3, 1163–1174. DOI 10.1214/aos/1176343247

Anderson TW, Darling DA (1954). A test of goodness of fit. *JASA* 49, 765–769. DOI 10.1080/01621459.1954.10501232

See Also

`plot.rmbl_dstfit()` for the density, QQ, PP and tail plots; `mksyncol()` with `type = "fitted"` for a synthetic sample from a fit.

Examples

```
set.seed(1)
x <- stats::rgamma(300, shape = 2, rate = 0.5)
f <- dstfit(x, nboot = 20)
f
f$fits$gamma

# counts with extra zeros
n <- ifelse(stats::runif(300) < 0.3, 0, stats::rpois(300, 4))
dstfit(n, nboot = 0)$table[, c("family", "aic")]

# a family the data do not allow
try(dstfit(c(-1, 2, 3, 4, 5), fams = "gamma"))
```

---

dstn *Distinct-value count in fixed memory*

Description

HyperLogLog: estimates how many distinct values a column holds using a fixed 4-byte register per bucket – 64 KB at the default `p = 14` – regardless of the column's length or cardinality. Exact counting needs memory proportional to the number of distinct values, which is the thing you cannot afford on a capsule member of unknown size. `dstnskch()` builds or extends the registers, `dstncnt()` turns them into an estimate, and `skchmrg()` combines two sketches.

Usage

```
dstnskch(x, p = 14L, regstr = NULL)

dstncnt(regstr)

skchmrg(a, b)
```

Arguments

x	A vector; coerced to character, since distinctness is compared on the rendered value. NA is skipped.
p	Log2 of the register count, a whole number from 4 to 20 (default 14, so 16,384 registers). Larger p is more precise and uses more memory.
regstr	Registers from a previous call (with the same p), to fold another chunk into the same sketch; NULL (the default) starts a new one. For dstncnt(), the registers to estimate from.
a, b	Register sets to merge.

Details

Each value is hashed and the first p bits pick one of $m = 2^p$ registers; the register keeps the longest run of leading zeros (plus one) seen in the remaining bits. The estimate is the bias-corrected harmonic mean $\alpha_m m^2 / \sum_j 2^{-M_j}$ of Flajolet et al. (2007).

The estimate carries a relative standard error of about `1.04 / sqrt(2^p)`, so 0.8% at the default. It is an ESTIMATE: use `length(unique(x))` when the column fits in memory and an exact answer matters. Below roughly `2.5 * 2^p` distinct values the estimator switches to linear counting, which is near-exact in that range (the estimate of one value is about 1.00003, not 1).

Values are hashed with the package's SHA-256, so the sketch is identical on every platform and across sessions. Values are compared as text (`as.character()`), so `1`, `1.0` and `"1"` are one value. Long vectors are processed in blocks of a million with a progress bar on stderr unless `options(morie.quiet = TRUE)`; the registers are the same as in one pass. Since a register only keeps a maximum, folding chunks with `regstr` and merging separate sketches with `skchmrg()` give the same registers as one sketch of all the data.

Errors: p not a whole number from 4 to 20; `regstr` not of length 2^p; registers that are not whole numbers from 0 to 64; two sketches of different sizes.

Value

`dstnskch()` an integer vector of 2^p registers; `dstncnt()` a length-1 numeric estimate (0 for an empty or NULL sketch); `skchmrg()` the element-wise maximum of two register sets, an integer vector.

References

Flajolet P, Fusy E, Gandouet O, Meunier F (2007). HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Analysis of Algorithms 2007*, 137–156.

Examples

```
set.seed(1)
x <- sample(1:5000, 200000, replace = TRUE)

# Close to the true 5000 distinct values, in fixed memory.
dstncnt(dstnskch(x))
length(unique(x))

# Small cardinalities are near-exact, via linear counting.
dstncnt(dstnskch(c("a", "b", "c", "a", "b")))

# Chunks fold into one sketch, so a file can be counted block by
# block, and two independent sketches can be merged.
s <- dstnskch(x[1:100000])
s <- dstnskch(x[100001:200000], regstr = s)
dstncnt(s)

a <- dstnskch(x[1:100000])
b <- dstnskch(x[100001:200000])
dstncnt(skchmrg(a, b))

# An empty input has no distinct values.
dstncnt(dstnskch(character(0)))

# a smaller sketch: less memory, more error
dstncnt(dstnskch(x, p = 8))

# sketches of different sizes cannot be merged
try(skchmrg(dstnskch(1:10, p = 4), dstnskch(1:10, p = 5)))
```

`dstntst` *Do distinct groups differ? A heterogeneity test before pooling*

Description

Tests whether the estimates of one indicator for distinct groups (First Nations, Metis and Inuit; Aboriginal and Torres Strait Islander people; any groups with published intervals) differ more than their intervals allow, and says how much of their spread is real: the evidence for reporting each group distinctly rather than as one pooled figure.

Usage

```
dstntst(
  data,
  grp,
  est,
  lower,
  upper,
```

```
  qual = NULL,
  conflvl = 0.95,
  pooled = NULL
)
```

Arguments

`data`	A data frame, one row per group.
`grp`, `est`, `lower`, `upper`	Column names.
`qual`	Optional column of quality flags; suppressed rows are left out (see `qltyrcd`).
`conflvl`	Confidence level of the published intervals, and the level of the verdict printed (default 0.95).
`pooled`	Optional labels of rows that pool other groups.

Details

With estimates y_g, standard errors s_g from the published intervals and weights $w_g = 1/s_g^2$, Cochran's $Q = \sum w_g(y_g - \bar{y}_w)^2$ has a chi-squared distribution on $G-1$ degrees of freedom when the groups share one value, and $I^2 = \max(0, (Q-(G-1))/Q)$ is the share of the spread that is real rather than sampling error (Higgins and Thompson 2002). A pooled figure is a weighted mean of different populations whenever I^2 is large; the population-weighted total Statistics Canada publishes also depends on how many people are in each group, so it is mostly a First Nations figure where First Nations people are most of the Indigenous population. Rows that pool groups are left out (`pooled`; see `indgap`); every other row is included, so drop the reference group (non-Indigenous people) yourself, and never test overlapping groups (`"alone"` beside "alone or in combination") together: they are not independent. With two or three groups the test has little power and I^2 is imprecise; a large p is not evidence that the groups are the same.

Value

A list of class `rmbl_distinctions`: `groups` (the estimates used, with their standard errors), `pooled` (the inverse-variance mean) and `poose`, `q`, `df`, `p_value`, `i2`, `leftout` (the rows not used) and `conflvl`. `print()` and `plot()` show it.

References

Higgins JPT, Thompson SG (2002). Quantifying heterogeneity in a meta-analysis. Statistics in Medicine 21(11):1539-1558. DOI 10.1002/sim.1186

See Also

`indgap`.

Examples

```
# perceived mental health very good or excellent, 2019/2022, adults (table
# 13-10-0924-01)
h <- indiex("health")
```

```
mh <- h[
  h$period == "2019/2022" &
    h$age == "Total, 18 years and over" &
    h$group != "Non-Indigenous" &
    h$indicator == "Perceived mental health, very good or excellent",
]
dt <- dstntst(mh, "group", "estimate", "lower", "upper", qual = "quality")
dt
plot(dt)
# your own table: made-up numbers, with the pooled row named
own <- data.frame(
  group = c("Aboriginal", "Torres Strait Islander", "Both"),
  est = c(24.0, 31.0, 24.6),
  lo = c(22.1, 26.2, 22.8),
  hi = c(25.9, 35.8, 26.4)
)
dstntst(own, "group", "est", "lo", "hi", pooled = "Both")
```

`dtctloc` *Where is this computation running?*

Description

Finds a location code for `co2int` without any network access, in this order:

1. the `RMBL_LOCATION` environment variable (any code or name the table knows);
2. `CODECARBON_COUNTRY_ISO_CODE` (alpha-3), the codecarbon variable, so a machine already configured for codecarbon needs nothing more;
3. the system time zone, mapped to its country through the IANA zone tables (`America/Toronto` is Canada, `Europe/Berlin` Germany);
4. the territory in the locale (`en_CA.UTF-8` is Canada).

When none of these yields a location in the table the answer is `"WORLD"`, and the method says so, so a report never silently applies one country's grid to another's computation.

Usage

```
dtctloc(
  envloc = Sys.getenv("RMBL_LOCATION"),
  codloc = Sys.getenv("CODECARBON_COUNTRY_ISO_CODE"),
  tz = Sys.timezone(),
  locale = Sys.getlocale("LC_TIME")
)
```

Arguments

`envloc`, `codloc`, `tz`, `locale`
: The inputs, exposed so a caller or a test can supply them; the defaults read the running session.

Details

The zone table (`inst/extdata/timezone_countries.csv`, built by `tools/build_timezones.R` from the IANA database) holds every zone of `zone.tab`, which names exactly one country per zone (`Europe/Stockholm` is Sweden even though the database now keeps its rules under `Europe/Berlin`), plus the aliases and old names of the `backward` file under their own country: the country of the zone the alias stood for before the database merged it (`Iceland` is `Atlantic/Reykjavik`, Iceland, not Abidjan's Ivory Coast), the link target's country when that target serves a single country (`Asia/Calcutta`, `US/Eastern`), and a reviewed list for the rest (`Atlantic/Jan_Mayen`, `America/Coral_Harbour`, `Canada/Eastern`). Every geographic zone `OlsonNames()` lists resolves to a country or territory; a territory the carbon table has no row of its own for (Monaco, Jersey, Vaduz, the Antarctic stations) uses the grid of the country it draws on or the world average, and the method says which. The `Etc/`, `UTC`, fixed-offset and generic zones (`CET`, `EST`, `WET`) name no country and fall through to the locale.

Value

A list with `location` (a code present in `co2inttab()`) and `method` (how it was found).

Examples

```
dtctloc()
dtctloc(envloc = "", codloc = "", tz = "Europe/Paris")
dtctloc(envloc = "", codloc = "", tz = "",
                locale = "en_IN.UTF-8")
# an explicit setting wins over everything else
dtctloc(envloc = "CA-QC", tz = "Europe/Paris")
# nothing usable: the world average, and the method says so
dtctloc(envloc = "", codloc = "", tz = "UTC", locale = "C")
```

---

dualcont *Wellness and distress as two continua*

Description

For each group, one wellness indicator against one distress indicator, each with its published interval, and the ratio of the two: the dual-continua view of mental health, in which wellness is not the absence of distress and is measured in its own right.

Usage

```
dualcont(x, well, distress, conflvl = 0.95)
```

Arguments

`x`	A wellness profile from `welprof`.
`well`, `distress`	One wellness and one distress indicator of `x`.
`conflvl`	Confidence level of the published intervals and of the ratio (default 0.95).

Details

Keyes (2002, 2005) showed that flourishing and mental illness are correlated but distinct axes: a person, or a population, can carry distress and also wellness. That is how Indigenous scholars of mental health describe it too, as wellness and as a process of healing rather than as illness alone (Stewart 2008). Plotting the groups on both axes keeps a strength visible beside the burden it sits next to. The `strratio` is $R = W/D$, wellness over distress, with the log-normal interval $\exp\{\log R \pm z\sqrt{\sigma_W^2 + \sigma_D^2}\}$ from the two published intervals: above 1, more people report the wellness than the distress. The two estimates come from the same respondents; their covariance is not published, so the interval treats them as independent and is too narrow when they are negatively correlated, as answers about wellness and distress usually are. Any two indicators on one scale can be set against each other.

Value

A data frame of class `rmbl_dual_continuum` with `group`, `wellness`, `wello`, `welup`, `distress`, `distrlo`, `distrup`, `strratio`, `ratiolo`, `ratioup` and `usewithcau`.

Acknowledgement

This estimator puts to quantitative use the framework of Professor Suzanne Stewart (Yellowknives Dene First Nation, University of Toronto).

See the acknowledgements of the `first-nations-metis-inuit-health` vignette.

References

Keyes CLM (2002). The mental health continuum: from languishing to flourishing in life. Journal of Health and Social Behavior 43(2):207-222. DOI 10.2307/3090197

Keyes CLM (2005). Mental illness and/or mental health? Investigating axioms of the complete state model of health. Journal of Consulting and Clinical Psychology 73(3):539-548. DOI 10.1037/0022-006X.73.3.539

Stewart SL (2008), see `holbal`.

See Also

`welprof`.

Examples

```
wb <- indiex("wellbeing")
w <- welprof(wb, "group", "indicator", "estimate", "lower", "upper",
                      well = "Always or often has a hopeful outlook",
                      distress = "Life satisfaction rating between 0 and 5",
                      qual = "quality")
dc <- dualcont(w, "Always or often has a hopeful outlook",
                     "Life satisfaction rating between 0 and 5")
dc
plot(dc)

# your own table: made-up percentages for two communities, with limits
```

```
own <- data.frame(
  community = rep(c("Lakeside", "Riverbend"), each = 4),
  item = rep(c("Hopeful", "Satisfied", "Rarely hopeful", "Low satisfaction"),
             2),
  pct = c(64, 71, 12, 18, 58, 66, 16, 22),
  lo = c(59, 66, 9, 14, 52, 60, 12, 17),
  hi = c(69, 76, 15, 22, 64, 72, 20, 27)
)
w <- welprof(own, "community", "item", "pct", "lo", "hi",
             well = "Hopeful", distress = "Rarely hopeful")
dualcont(w, "Hopeful", "Rarely hopeful")
```

`duprows` *Duplicated rows, with their groups*

Description

Returns the rows that share a combination of `cols` with at least one other row, grouped so the duplicates sit together. The counterpart of `janitor::get_dupes()`.

Usage

```
duprows(data, cols = NULL, quiet = FALSE)
```

Arguments

`data`	A data frame whose rows are checked for duplicates.
`cols`	Character vector of the columns defining a duplicate, or `NULL` (the default) for all of them.
`quiet`	Logical: `TRUE` skips the boxed count of duplicate rows and groups, which is otherwise sent through `message()`; default `FALSE`.

Details

`ruledstn()` tells you THAT there are duplicates, which is what a validation gate needs. This shows you WHICH, which is what fixing them needs – and a duplicate is usually a join that fanned out or a re-release appended rather than replaced, both of which are visible only once the offending rows are in front of you.

Two rows are duplicates when their values, converted to text, are equal in every column of `cols`; two `NA`s are equal. Columns are taken by position when `cols = NULL`, so a table with repeated column names can still be inspected. Errors when `data` is not a data frame, a named column does not exist, or a named column's name is shared by two columns.

Value

A data frame: the duplicated rows (every occurrence, including the first), ordered so each group sits together, with an integer `dupe_count` column giving each group's size and row names reset. Zero rows (with the same columns) when there are none.

See Also

ruledstn(), ruleuniq()

Examples

```
df <- data.frame(
  id = c(1, 2, 2, 3, 3, 3),
  value = c("a", "b", "b", "c", "d", "c"),
  stringsAsFactors = FALSE
)

# Duplicated on every column.
duprows(df)

# Duplicated on the identifier alone, which catches more.
duprows(df, "id")

# No duplicates gives zero rows, not an error.
duprows(data.frame(x = 1:3))

# a column that does not exist is an error
try(duprows(df, "code"))
```

---

ebrates *Empirical Bayes rates, shrunk toward the overall experience*

Description

A small area's rate is mostly noise, so ranking areas by their raw ratios puts the smallest areas at both ends of the table by construction. This function borrows strength across areas: each area's ratio of observed to expected is pulled toward the overall ratio by an amount that depends on how little information the area carries. Use the shrunk estimates for maps and rankings; use stdinc for each area's own evidence.

Usage

```
ebrates(obs, expd, area = NULL)
```

Arguments

obs	Numeric vector of observed counts (non-negative, no NA).
expd	Numeric vector of expected counts (positive), the same length, for example from expcnts.
area	Optional vector of area labels, added as the first column. Default NULL.

Details

The area relative risks θ_a are taken to come from a gamma prior with shape ν and rate α. Its mean is the overall ratio $m = \sum O_a / \sum E_a$, and its variance is estimated by the method of moments,

$$s^2 = \frac{\sum_a E_a(\theta_a - m)^2}{\sum_a E_a} - \frac{m}{\bar{E}},$$

with $\theta_a = O_a/E_a$ and $\bar{E}$ the mean expected count; the subtracted term is the spread that Poisson sampling alone would produce. Then $\alpha = m/s^2$, $\nu = m^2/s^2$, and the posterior mean $(O_a + \nu)/(E_a + \alpha)$ is reported, with shrinkage $\alpha/(E_a + \alpha)$. Where the expected count is large the data dominate and the estimate barely moves; where it is small the prior does, which is the intended behaviour and not a defect.

When the between-area variance estimate comes out at or below zero there is no evidence of any real variation between areas, and every estimate collapses to the overall rate. That is reported through `shrinkage` (all ones) and `NA` prior parameters rather than hidden.

Errors: lengths that differ; missing, negative or infinite counts; missing, infinite or non-positive expected counts; fewer than two areas.

Value

A data frame with one row per area: `area` (when given), `observed`, `expected`, the raw ratio `sir`, the shrunk estimate `eb`, the `shrinkage` applied (zero means untouched, one means replaced by the overall rate), and the fitted prior's `nu` (shape) and `alpha` (rate), the same on every row.

References

Clayton, D. and Kaldor, J. (1987). Empirical Bayes estimates of age-standardized relative risks for use in disease mapping. *Biometrics* 43(3), 671-681.

Lawson, A. B. *Using R for Bayesian Spatial and Spatio-Temporal Health Modeling*. Chapman and Hall/CRC, which cites Clayton and Kaldor as the empirical-Bayes approximation in the development of Bayesian disease mapping.

See Also

`stdinc`, `fnllim`

Examples

```
# Three areas, one of them tiny. The tiny area's raw ratio is
# extreme; its shrunk one is not.
ebrates(obs = c(30, 45, 2), expd = c(25, 50, 0.5),
         area = c("North", "South", "Tiny"))

# no variation beyond chance: every estimate is the overall ratio
ebrates(c(10, 11, 9), c(10, 10, 10))

# one area has nothing to borrow from
try(ebrates(5, 4))
```

envdiff *Compare two captured environments*

Description

Reports what changed between the environment captured by one run and another: the R version, the platform, and every package whose version differs, was added, or disappeared. Use it when a result does not reproduce, to see first whether the software around it changed.

Usage

```
envdiff(a, b)
```

Arguments

a, b	Environment records from capenv(), or whole manifests (such as mkmnf() returns) that hold an environment element.

Details

This is the question a failed reproduction actually raises. A manifest records the environment; comparing two manifests by eye across a few hundred packages does not scale, and the one line that matters – a dependency that moved a minor version – is exactly what gets missed.

Versions are compared as strings. A package recorded in b but not in a (or recorded with an NA version in a) is "added"; the reverse is "removed". The operating-system string, the capture time and the random number generator settings are not compared. Any other field present in only one record, or different in the two, is named in fields and makes them differ. Rows follow the order of a's packages, then b's new ones.

Error: a or b that is not a list.

Value

A list of class c("bricklayer_env_diff", "list"): identical (logical, TRUE when R version, platform and all package versions agree), r_version (a named length-2 character vector c(a = , b =) when they differ, else NULL), platform (likewise), and packages (a data frame with character columns package, a, b and change, where change is "added", "removed" or "changed"; no rows when nothing differs), and fields (the names of any other fields that differ, character(0) when none).

See Also

capenv(), mkmnf()

Examples

```
a <- capenv(c("stats", "utils"))
b <- a

# An environment matches itself.
envdiff(a, b)$identical

# Stage a moved dependency, an added one and a newer R.
b$packages[["stats"]] <- "0.0.0"
b$packages[["tools"]] <- "4.9.0"
b$r_version <- "9.9.9"
d <- envdiff(a, b)
d$packages
d$r_version
d

# Whole manifests are accepted, not just the environment block.
m <- mkmnf(list(run = "demo"))
envdiff(m, m)$identical

# anything other than a record is refused
try(envdiff("4.5.1", a))
```

---

evlrr *How strong would an unmeasured confounder have to be*

Description

The E-value of VanderWeele and Ding (2017): the minimum strength of association, on the risk-ratio scale, that an unmeasured confounder would need with BOTH the exposure and the outcome to explain away an observed risk ratio.

It answers the question a covariate list cannot: not whether the analysis adjusted for the right things, but how much unmeasured confounding it would take to move the result to nothing. An E-value of 1.2 says very little would be needed; an E-value of 5 says a confounder five times more common in the exposed group AND five times more associated with the outcome would have to have gone unnoticed.

The E-value for the confidence limit is the one to report alongside it: a large point-estimate E-value with a limit E-value of 1 means the interval already includes no effect, and no confounding is needed at all.

Usage

```
evlrr(rr, lo = NULL, hi = NULL, true = 1)
```

Arguments

`rr`	Observed risk ratio, greater than 0. Protective effects (below 1) are inverted first, as the measure is symmetric.
`lo`, `hi`	Optional confidence limits on the same scale. The limit nearer the null is the one used.
`true`	Number, default 1 (the null): the value to move the estimate to. A value other than 1 asks how much confounding would move the estimate to `true` instead of to no effect.

Details

For a risk ratio $RR \geq 1$ (after dividing by `true`, and inverting a ratio below 1),

$$E = RR + \sqrt{RR(RR - 1)}.$$

For the interval, the limit nearer the null is used: the lower limit when it is above 1, else the reciprocal of the upper limit when it is below 1; when the interval contains 1 the limit E-value is 1. The formula is for risk ratios; VanderWeele and Ding (2017) give approximate conversions for odds ratios with a common outcome and for hazard ratios, which this function does not apply. Only the first element of each argument is used. Errors: a ratio or limit that is not finite and positive.

Value

A named numeric vector: `evlpoint` and, when limits are given, `evllimit`.

References

VanderWeele, T. J., and Ding, P. (2017). Sensitivity Analysis in Observational Research: Introducing the E-Value. *Annals of Internal Medicine* 167(4), 268-274. DOI 10.7326/M16-2607

See Also

`cpsfals()` for controls that use the data, where this uses none.

Examples

```
# a risk ratio of 2 needs a confounder associated by 3.41 with both
evlrr(2)

# a protective effect is inverted, so 0.5 gives the same answer
evlrr(0.5)

# an interval that already includes the null needs nothing
evlrr(2, lo = 0.9, hi = 4.4)

# and a strong result with a limit well above the null is harder to
# explain away
evlrr(3, lo = 2.1, hi = 4.3)

# how much confounding would bring a risk ratio of 3 down to 1.5
```

```
evlrr(3, true = 1.5)

try(evlrr(-2))
```

---

evstudy *An event study: the effect period by period around adoption*

Description

Leads and lags of a policy's adoption, from a two-way fixed-effects regression on indicators of the time since adoption, with a joint test that the leads are zero (no effect before the policy, the testable part of parallel trends).

Usage

```
evstudy(
  data,
  unit,
  time,
  y,
  ftreat,
  leads = 4L,
  lags = 4L,
  ref = -1L,
  bin = TRUE,
  covs = NULL,
  cl = NULL,
  conflvl = 0.95,
  w = NULL
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`ftreat`	Column name of the period a unit is first treated (`NA`, 0 or `Inf` for a unit never treated). Treatment is taken as staying on.
`leads`, `lags`	The number of periods before and after adoption to estimate (default 4 and 4).
`ref`	The period left out (default -1).
`bin`	Bin the periods beyond the window into its end points (default `TRUE`).
`covs`	Optional column names of time-varying controls.

`cl` Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.

`conflvl` Confidence level of the intervals reported (default 0.95).

`w` Optional column name of non-negative weights, such as population or matching weights, for weighted least squares with weighted within-means. Rows of weight 0 take no part in the fit.

Details

With G_i the first treated period of unit i and $k = t - G_i$,

$$Y_{it} = \sum_{k=-L, k\neq r}^{M} \gamma_k \mathbf{1}(t - G_i = k) + X_{it}'\theta + \delta_i + \mu_t + \varepsilon_{it},$$

with the reference period r (default -1, the period before adoption) left out. Times beyond the window are binned into its end points (`bin = TRUE`), or dropped. Units never treated have every indicator 0. The variance and degrees of freedom are those of `plndid`. The pre-trend test is the Wald statistic $W = \hat{\gamma}_{pre}' \hat{V}_{pre}^{-1} \hat{\gamma}_{pre}$, reported as $F = W/q$ on q and $G - 1$ degrees of freedom.

With staggered adoption and effects that change over time, the lead and lag coefficients can mix in effects from other periods (Sun and Abraham 2021); `stgatt` gives the event-time effects free of that.

Value

A data frame of class `rmbl_event_study`: `evtm`, `estimate`, `se`, `lower`, `upper`, `t`, `p_value`, with the reference period as a row of zeros; attributes `pretrend` (a list: `F`, `df1`, `df2`, `p_value`), `vcov` (of the estimated event times), `n`, `clusters`, `conflvl`. `plot()` draws it.

References

Sun L, Abraham S (2021). Estimating dynamic treatment effects in event studies with heterogeneous treatment effects. Journal of Econometrics 225(2):175-199.
DOI 10.1016/j.jeconom.2020.09.006

Applied to the effect of donations on police stops, block by quarter: Zorro Medina A, Hackett D, Green D, Vargas R (2025). Evidence of donor bias in Chicago police stops. arXiv:2504.15140 (https://arxiv.org/abs/2504.15140).

See Also

`plndid`, `stgatt`.

Examples

```
p <- polex()
es <- evstudy(p, "district", "month", "crmrate", "refmonth", leads = 3,
                  lags = 4)
es
attr(es, "pretrend")$p_value
plot(es)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# three years either side of adoption, weighted by population
evstudy(own, "town", "year", "rate", "first", leads = 3, lags = 3,
        w = "pop")
```

---

expbands *Expand a banded frequency table into per-unit values*

Description

Turns a banded frequency table (labels and a count per band) into a numeric vector with one value per unit, using the representative value of each band from `bandvals`. The result can be passed to any function that expects individual values, such as `ginicoef`, `mean()` or `quantile()`. Bands that cannot be parsed but have units in them are an error by default, so a statistic never describes only part of the data without saying so.

Usage

```
expbands(bands, cnts, ..., drpunp = FALSE)
```

Arguments

bands	A data frame from `prsbands`, or labels to parse.
cnts	Numeric vector, one non-negative count per band.
...	Further arguments to `bandvals`: `rule`, `openupcap`, `openupfac` and `openloflr`.

drpunp — Logical, default FALSE. FALSE makes a band whose label could not be parsed, yet which has a count, an error; TRUE leaves those units out with a warning instead.

Details

Each band's value is repeated round(counts) times, in band order. Bands with an NA count are skipped. A band with an NA value (unparsed label) and a positive count is an error unless drpunp = TRUE, in which case its units are left out with a warning that gives their number and labels.

Errors: cnts of the wrong length, non-numeric or negative; an unplaceable band with units (by default); any error from bandvals.

Value

A numeric vector with one entry per unit (length sum(round(counts)) over the placed bands).

See Also

bandvals, bandsens

Examples

```
x <- expbands(c("1", "2 to 5", "Greater than 5"),
                  cnts = c(10, 4, 2), openupcap = 12)
table(x)
ginicoef(x)

# the lower-bound reading of the same table
expbands(c("1", "2 to 5", "Greater than 5"), cnts = c(10, 4, 2),
             rule = "lower")

# an unparseable band with units in it stops the call
try(expbands(c("1", "not stated"), cnts = c(3, 2)))
```

---

expcnts — *Expected counts under indirect standardisation*

Description

Computes, for each area (region, district, institution), the count it would have if it experienced the overall rate in every stratum, given its own population in each stratum. Comparing an area's observed count with this expected count, rather than comparing raw rates, removes the part of the difference that is explained by who lives in (or is held in) the area. The result feeds stdinc, ebrates and fnllim.

Usage

```
expcnts(cnts, pop, area, strata = NULL)
```

Arguments

`cnts`	Numeric vector of observed counts (non-negative), one per row of the area-by-stratum table.
`pop`	Numeric vector of the population at risk (positive), the same length as `cnts`.
`area`	Vector of area labels, one per row (converted to character).
`strata`	Optional vector of stratum labels, one per row: an age band, a gender, or their interaction. Default `NULL`: one stratum, so only size is adjusted for. With strata the standardisation is indirect in the usual sense: the overall stratum-specific rates are applied to each area's own stratum populations.

Details

With O_{as} and P_{as} the count and population of area a in stratum s, the reference rate of a stratum is pooled over all areas, $r_s = \sum_a O_{as} / \sum_a P_{as}$, and

$$E_a = \sum_s P_{as} r_s .$$

Without strata the expected count is just the area's population times the overall rate, which adjusts for size but not for composition. The two are worth distinguishing: a region holding disproportionately many young men will show an excess on the first and may show none on the second, and only the second is evidence about the region.

Rows may repeat an area-stratum pair; they are summed. Rows with a missing count, population, area or stratum are dropped. The output is sorted by area label. Because the reference rates come from the same data, $\sum_a E_a = \sum_a O_a$.

Errors: non-numeric, negative or infinite counts or populations; a population of zero; `cnts`, `pop`, `area` or `strata` of different lengths.

Value

A data frame with one row per area and columns `area` (character), `observed`, `population` and `expected` (numeric totals over the area's rows).

References

Lawson, A. B. *Using R for Bayesian Spatial and Spatio-Temporal Health Modeling*. Chapman and Hall/CRC. Chapter 1 sets out the convention used here: the expected counts come from applying the overall population rate to each area, and the standardised incidence ratio is the ratio of count to expected.

Hedderich, J. and Sachs, L. (2020). *Applied Statistics: Methods Using R*. Springer, on the distinction between indirect standardisation, which applies the reference's stratum-specific rates to the study population, and direct standardisation, which does the reverse.

See Also

`stdinc`, `ebrates`, `fnllim`

Examples

```
d <- data.frame(
  region = rep(c("North", "South", "East"), each = 2),
  age = rep(c("18 to 24", "25 to 49"), 3),
  n = c(30, 45, 12, 60, 8, 20),
  pop = c(1000, 6000, 900, 9000, 400, 3500)
)

# Adjusting for size only.
expcnts(d$n, d$pop, d$region)

# Adjusting for size AND age composition, which is the comparison
# that says something about the region.
e <- expcnts(d$n, d$pop, d$region, strata = d$age)
e

# the expected counts feed the standardised ratio
stdinc(e$observed, e$expected, area = e$area)

# a population of zero has no rate
try(expcnts(c(1, 2), c(10, 0), c("a", "b")))
```

---

expconcidx *Concentration index of exposure by income*

Description

Measures whether exposure to pollution is concentrated among lower-income or higher-income people. The index runs from -1 (all the exposure borne by the poorest) through 0 (exposure shared evenly across income) to +1 (all of it borne by the richest); environmental-justice analyses report it next to the burden.

Usage

```
expconcidx(data, expo, income)
```

Arguments

`data`	A data frame with one row per person or area.
`expo, income`	Column names (single strings): the exposure (a non-negative concentration) and the income measure (numeric).

Details

Wagstaff, Paci and van Doorslaer (1991):

$$CI = (2/\mu)\,\mathrm{cov}(h_i, R_i)$$

with $R_i = (rank_i - 0.5)/n$ the fractional income rank (tied incomes share their average rank, so the index does not depend on row order) and the population covariance. Negative values mean lower-income units bear more exposure. Both columns must be numeric: a factor or character income is refused rather than ranked by its level codes, and exposures must be non-negative.

Rows with a missing exposure or income are dropped. The interpretation calls an index below -0.01 a burden on the poor, above 0.01 a burden on the rich, and anything between an even distribution. Refusals: column names that are not single strings or not in `data`; fewer than 2 complete rows; a non-numeric or infinite value; a negative exposure; an income that takes one value only (no income gradient, so the index is undefined); and a mean exposure of zero (the index divides by it). Income only needs an order: an amount, a quintile number or a rank all give the same index.

Value

A list of class `rmbl_equity` (`plot()` draws the concentration curve) with `concidx` (a number in $[-1, 1]$), `interpretation` (a sentence), `expomean`, `citation` (a string) and `extra`, a list with `n` (rows used), `cov_h_R` (the covariance) and `curve`, a data frame of the concentration curve. Its column `population` is the cumulative population share, poorest first, and `exposure` the cumulative share of total exposure.

References

Wagstaff, A., Paci, P. and van Doorslaer, E. (1991). On the measurement of inequalities in health. Social Science and Medicine 33(5), 545-557.

Examples

```
d <- data.frame(exposure = c(30, 25, 20, 15, 10), income = 1:5)
ci <- expconcidx(d, "exposure", "income")
ci$concidx
ci$interpretation

# Even exposure across income gives 0; tied incomes share a rank.
expconcidx(data.frame(exposure = 20, income = c(1, 1, 2, 3)),
                              "exposure", "income")$concidx

# A text income cannot be ranked.
try(expconcidx(data.frame(exposure = 1:3, income = c("a", "b", "c")),
                                  "exposure", "income"))
```

`explerr` *Explain an error from rmoriebricklayer (or a package built on it)*

Description

Finds where an error or warning message comes from, without a language model and without the network: every message the packages raise is written in their code, so the code is searched. A message built from parts (a file name, a number) matches with those parts as wildcards; one that

was cut short or retyped matches by its words. For each match you get the function that raises it, the exported functions it is reached from (their help pages say what they expect), and the message as written, which usually says what to do.

Usage

```
explerr(msg = geterrmessage(), pkgs = NULL, n = 3L)
```

Arguments

msg	The message: the text of an error, as R printed it ("Error in f(x) : ..." is fine), a condition object, or a try() result. Default: the last error of this session (geterrmessage()). Or a data frame of check rows with the columns check, ok and detail, as rmbl verify --json writes them and vrfcps returns them in checks: each failed row is explained.
pkgs	Character vector of installed packages to search. The default, NULL, means this package, or the comma-separated list in RMBL_PACKAGES: an environment variable that a package built on this one sets for its own command line.
n	How many matches to return, best first: a whole number, default 3.

Details

The first call in a session walks the body of every function in the searched packages and collects each stop(), warning() and usage error whose literal text has at least 12 letters. A message built with sprintf(), gettextf(), paste() or paste0() becomes a template with its computed parts as wildcards. Each raising function is traced up to four calls to the exported functions that reach it (at most six are listed). The table is kept in memory for the session (about a second to build for this package), and, while a person is watching an interactive session and the answer cache is on, also saved as an .rds file in the answer-cache directory.

The answer-cache directory is the one tools::R_user_dir() gives for "rmoriebricklayer" and "cache", or the one the rmoriebricklayer.llmcachedir option names when it is set. The cache is off under options(morie.quiet = TRUE), MORIE_LLM_CACHE=off, tests and R CMD check.

The message is cleaned first: an "Error in f(x) :" or "Warning message:" prefix is dropped, curly quotes become straight ones and white space is collapsed. A template that matches the start of the message as a regular expression is an exact match and ranks first (longer templates before shorter ones). Otherwise a template scores $0.6h/|m| + 0.4h/|t|$, where h is the number of shared words (three letters or more, common words left out, at least two needed), $|m|$ the words of the message and $|t|$ those of the template; scores of 0.55 and above are kept.

Errors: msg that is not character text, a condition or a data frame with the columns check, ok and detail; n not a whole number of 1 or more; pkgs that are not installed.

Value

A data frame with at most n rows, best match first, and the character columns package, raised_by (the function whose code raises it), exported (the exported functions that reach it, separated by commas), kind ("error" or "warning"), message (the message as written, with ... where a value goes) and match ("exact" or the share of its words found, such as "80% of its words"). No rows when the message is not one of these packages' own: it came from another package or from R, and

llmask can help with it. For a data frame of check rows: one row per failed check, with the columns check, detail, explanation (what the failure means and what to do) and raised_by (pkg::fn when the detail is a message of these packages, else NA).

Examples

```
msg <- "`prompt` must be a single non-empty string"
explerr(msg, pkgs = "rmoriebricklayer")

# a condition object
e <- tryCatch(kemsz(3), error = identity)
explerr(e, n = 1)

# the failed rows of a capsule check
rows <- data.frame(
  check = c("data_present", "data_sha256"),
  ok = c(TRUE, FALSE),
  detail = c("d.csv", "expected 01..., got 9c...")
)
explerr(rows)$explanation

# n must be at least 1
try(explerr(msg, n = 0))


# an error caught with try(), and the message as R prints it, with the call
# in front
r <- try(llmcfg(no.such = 1), silent = TRUE)
explerr(r, pkgs = "rmoriebricklayer")
explerr(
  paste0(
    "Error in mrkprf(x, 9) : `index` must be between 1 and the number ",
    "of chunks"
  ),
  n = 1
)

# a message retyped from memory
explerr("index between number chunks", n = 1)

# R's own messages are not in the table: no rows
nrow(explerr("object 'x' not found"))
```

---

exprspplr *Exposure-response estimate with a bootstrap sensitivity interval*

Description

Fits the partially linear exposure-response model with plrcf (or any estimator with the same interface), then re-fits it on nboot non-parametric resamples of the rows for a percentile interval (Efron

and Tibshirani 1993, chapter 13). The bootstrap interval does not lean on the estimator's own standard error, so it is a check on it: when the two disagree, the analytic one should not be trusted alone.

Usage

```
exprspplr(data, y, expo, confs, nboot = 100L, rstate = 42L, estr = plrcf)
```

Arguments

`data`	A data frame with one row per unit.
`y`, `expo`, `confs`	Column names: the outcome and the exposure (single strings) and the confounders (one or more strings).
`nboot`	Number of resamples, a non-negative whole number (default 100); at least 10 must succeed.
`rstate`	Seed for the resampling (default 42).
`estr`	A function `(data, outcome, treatment, covariates)` returning a list with `ate` and `se`. The default is `plrcf`; rmorie's `morie_estimate_double_ml()` has the same interface.

Details

Each resample draws `nrow(data)` rows with replacement and calls `estr(resample, outcome, exposure, confounders)`; a resample on which the estimator fails is skipped, and the run stops if fewer than 10 succeed. The interval is the 2.5% and 97.5% type-7 quantiles of the replicate estimates, and `seboot` their standard deviation. Resampling uses `rstate` and restores R's own random stream afterwards. The cost is `nboot + 1` estimator calls; with `plrcf` at n = 20000 and 100 resamples that is a few seconds. A progress bar on stderr follows the resamples when a person is watching (`options(morie.progress = TRUE)` outside an interactive session; `options(morie.quiet = TRUE)` silences it).

Refusals: column names that are not strings, an `nboot` that is not a single non-negative whole number, an estimator that is not a function, and the estimator's own errors on the full sample.

Value

A list of class `c("rmbl_plr_boot", "list")` (its print method shows it as a box) with `ate` (the full-sample estimate), `seana` (the estimator's standard error), `seboot` (the standard deviation of the replicates), `cilobs` and `ciupbs` (the 95% percentile interval), `nboot` (the number of successful resamples) and `method` (the estimator's method with `" + percentile bootstrap"`).

References

Efron, B. and Tibshirani, R. J. (1993). An Introduction to the Bootstrap. Chapman and Hall.

See Also

`cbootmean` for a bootstrap of a mean.

Examples

```
set.seed(1)
d <- data.frame(x1 = rnorm(200), x2 = rnorm(200))
d$exposure <- 20 + 2 * d$x1 + rnorm(200)
d$asthma <- 5 + 0.3 * d$exposure + d$x2 + rnorm(200)
r <- exprspplr(d, y = "asthma", expo = "exposure",
                            confs = c("x1", "x2"), nboot = 20)
r$ate
c(r$cilobs, r$ciupbs)

# Any estimator with the same interface: here an ordinary regression.
ols <- function(data, outcome, treatment, covariates) {
  f <- stats::reformulate(c(treatment, covariates), outcome)
  cf <- summary(stats::lm(f, data = data))$coefficients
  list(ate = cf[treatment, 1], se = cf[treatment, 2], method = "OLS")
}
exprspplr(d, "asthma", "exposure", c("x1", "x2"),
                       nboot = 20, estr = ols)$method

# Fewer than 10 resamples cannot give an interval.
try(exprspplr(d, "asthma", "exposure", c("x1", "x2"), nboot = 5))
```

---

exptrnd *A dose-response across generations of exposure*

Description

For counts of an outcome (suicidal thoughts, distress) among people with 0, 1 or 2 generations of residential school attendance in their family (or any ordered exposure), the odds ratio of each level against the lowest level, with Woolf intervals, and the Cochran-Armitage test for a trend across the levels.

Usage

```
exptrnd(cases, n, dose = seq_along(cases) - 1L, conflvl = 0.95)
```

Arguments

`cases`	Number with the outcome at each exposure level.
`n`	Number of people at each level.
`dose`	The exposure levels, in order (default `0, 1, 2, ...`); the trend test scores them.
`conflvl`	Confidence level of the odds ratios' intervals (default 0.95).

Details

With a_k cases and $b_k = n_k - a_k$ non-cases at level k, the odds ratio against the lowest level is $(a_k/b_k)/(a_1/b_1)$, with Woolf's interval $\exp\{\log OR \pm z\sqrt{1/a_k + 1/b_k + 1/a_1 + 1/b_1}\}$. When any cell is empty, one half is added to every cell of every level (Haldane-Anscombe), so every odds ratio is finite. The trend test is the Cochran-Armitage test (`prop.trend.test`) with the levels as scores. Counts must be unweighted. For a complex survey, take the weighted analysis from the survey itself (`svydes`): this function is for published or aggregated counts.

Value

A data frame with `dose`, `cases`, `n`, `proportion`, `orratio`, `lower`, `upper`; the trend test is the attribute `trend` (an `htest`).

References

McQuaid RJ, et al. (2017), see `genexpo`.

Examples

```
# made-up counts by 0, 1 and 2 generations of attendance, to show the
# computation
et <- exptrnd(cases = c(60, 95, 70), n = c(500, 520, 300))
et
plot(et)
```

`falsfam` *Correct a family of falsification results for multiple testing*

Description

Takes the p-values from several `cpsfals()` runs – or a plain numeric vector – and reports which survive once the size of the family is accounted for.

Running the permutation control over twenty statistics and reporting the one that came in under 0.05 is not a finding: at that family size roughly one spurious result is what chance produces. Which correction to use depends on the claim. Holm controls the probability of ANY false positive, which is what a claim about a specific statistic needs. Benjamini-Hochberg controls the expected PROPORTION of false positives among those declared, which is what a screening exercise needs; it is less conservative and says something weaker.

A permutation p-value cannot fall below `1 / (n + 1)`, so with a small number of permutations a whole family can be uncorrectable – every p sits at the floor. That is reported rather than hidden, because the alternative is a table of adjusted values that look like evidence of nothing in particular.

Usage

```
falsfam(x, meth = c("holm", "bh", "bonferroni", "none"), alpha = 0.05)
```

Arguments

x	A named list of bricklayer_falsification objects, or a named numeric vector of p-values.
meth	"holm" (the default), "bh", "bonferroni" or "none".
alpha	Number, default 0.05: the threshold applied to the adjusted values.

Details

With the m p-values sorted, $p_{(1)} \le \cdots \le p_{(m)}$, the adjusted values are: Holm, $\max_{j \le i}(m - j + 1)p_{(j)}$; Benjamini-Hochberg, $\min_{j \ge i}(m/j)p_{(j)}$; Bonferroni, $mp_{(i)}$; all capped at 1. They agree with stats::p.adjust() with methods "holm", "BH" and "bonferroni". A member survives when its adjusted value is at most alpha.

For a list of cpsfals() results the p-value used is that of each result's permutation control, and the floor is $1/(n + 1)$ for its n permutations. For a numeric vector there is no floor to check and unnamed values are named p1, p2, ...

Errors: a list whose members are not all falsification results, or that is unnamed; a member without a permutation p-value; a missing p-value; a p-value outside $[0, 1]$.

Value

A list of class bricklayer_falsify_family with results (a data frame of name, p, adjusted and logical survives, sorted by p), method, famsz (integer), alpha, and atflr (the names whose p-value equals the smallest their permutation count could produce; empty for a numeric vector). The print method shows the table.

See Also

cpsfals(), prrgdecl().

Examples

```
# four statistics, one of which is real
set.seed(1)
d <- data.frame(x = rnorm(150))
d$y <- 0.6 * d$x + rnorm(150)
d$a <- rnorm(150)
d$b <- rnorm(150)
fam <- list(
  real = cpsfals(d, function(z) cor(z$x, z$y),
                         trt = "x", n = 199, seed = 1),
  noisea = cpsfals(d, function(z) cor(z$a, z$y),
                             trt = "a", n = 199, seed = 2),
  noiseb = cpsfals(d, function(z) cor(z$b, z$y),
                             trt = "b", n = 199, seed = 3))
falsfam(fam)

# the correction is what stops the smallest of several from being
# read as the finding
falsfam(c(a = 0.01, b = 0.04, c = 0.2, d = 0.5))$results
```

```
# the false-discovery rate is less strict than Holm
falsfam(c(a = 0.01, b = 0.04, c = 0.2, d = 0.5), meth = "bh")$results

try(falsfam(c(a = 0.01, b = 1.2)))
```

`fdl` *Download a File With Diagnostic Error Messages*

Description

Downloads one file like `dldata()`, but never stops on a network failure: it returns `TRUE` or `FALSE`, prints plain-language advice for the usual academic and corporate network problems, and then tries a Wayback Machine copy of the same address. Use it in scripts that other people run on networks you do not control (university VPNs, corporate proxies), where a bare "cannot open the connection" is not enough to act on.

Usage

```
fdl(url, tpath, attwb = NULL, allowfile = FALSE)
```

Arguments

`url`	URL to download, a single string.
`tpath`	Destination path on disk, a single string.
`attwb`	The Wayback Machine snapshot URL tried as a fallback if the primary download fails. `NULL` (the default) resolves a snapshot automatically via `wbsnpurl()` (one more request); a URL string overrides the lookup; `""` or `FALSE` disables the fallback entirely.
`allowfile`	Logical. Accept a `file://` URL for `url` (default `FALSE`; offline tests pass `TRUE`). It does not apply to `attwb`.

Details

The URL is checked first, as in `dldata()`; a refused URL is an error, not `FALSE`. The download is then attempted. On failure the messages name the URL and the error, then the likely causes picked out from the error text: rate limiting (HTTP 429, common behind shared VPN addresses), a TLS handshake failure (VPNs that inspect TLS, such as Cisco AnyConnect, GlobalProtect, Zscaler or Netskope), a DNS failure, a refused or reset connection, a timeout, an HTTP 403, or otherwise a moved file or a portal that is down. When base R reports the HTTP status only in a warning, that warning's text is used as the cause.

The fallback: with `attwb = NULL` a snapshot is looked up with `wbsnpurl()`; with a URL that URL is used; with `""` or `FALSE` the fallback is off. The fallback is fetched without `allowfile`, so it must be a public `https` URL even when `allowfile = TRUE`; a refused fallback URL counts as a failed fallback (`FALSE`), with a message.

All output goes to the message stream. The transport always draws its progress (it is called with `quiet = FALSE`). `tpath` is overwritten on success.

Value

TRUE if either the primary download or the Wayback fallback succeeded (the file is at tpath), otherwise FALSE.

Examples

```
src <- tempfile(fileext = ".csv")
writeLines(c("id,value", "1,10"), src)

# a successful download returns TRUE
fdl(paste0("file://", src), tempfile(fileext = ".csv"),
  attwb = "", allowfile = TRUE)

# a failed one returns FALSE with advice, here with the fallback off
fdl(paste0("file://", src, "-missing"), tempfile(),
  attwb = "", allowfile = TRUE)

# a refused URL is an error, not FALSE
try(fdl("http://example.org/data.csv", tempfile()))


# needs the network; the Wayback fallback is looked up on failure
ok <- fdl("https://cloud.r-project.org/",
  tempfile(fileext = ".html"))
ok
```

---

fetchvrf *Fetch a file from its source or a mirror, verified by its SHA-256*

Description

Downloads url to dest, trying the source first (with its Wayback Machine snapshot as the source's own fallback, as blfetch does) and then each mirror in turn, and keeps the first copy whose SHA-256 is sha256. A copy with another digest is never used: a changed or truncated file is a different file.

Usage

```
fetchvrf(
  url,
  dest,
  sha256 = NULL,
  mirrors = character(),
  tmout = 120L,
  arcs = "rmorie"
)
```

Arguments

url	The source URL (https).
dest	Path the verified file is written to (by way of a .part file).
sha256	The expected SHA-256, 64 hexadecimal characters, or NULL.
mirrors	Further URLs that serve the same file, tried in order after the source.
tmout	Seconds allowed for each download (default 120).
arcs	Archives tried after the source, its Wayback snapshot and the mirrors: "rmorie" (archive.rmorie.com, by arcget) and folders written by arccap (read from their WARC files), in order. character() tries none.

Details

Some servers refuse the Internet Archive's crawler while serving everyone else (Statistics Canada does), so a Wayback snapshot of their files may not exist. The remedy is a capture made from elsewhere, published as a mirror and pinned by its digest. It can be an Internet Archive item, at https://archive.org/download/ followed by <item>/<file>; the data service; or an institutional repository. Because the digest is checked, a mirror does not have to be trusted: a copy that matches is the same bytes the analysis used, wherever it came from.

Without sha256 the first copy that downloads is kept and its digest is returned, to be pinned for the next run.

Value

Invisibly, a list of class rmbl_fetch (it prints as a box): source ("live", "wayback", "mirror", "rmorie" or "local"), url (the URL that served the kept copy), sha256, bytes, and tried (a data frame of every attempt: url, outcome).

See Also

blfetch, sha256file, wbsave.

Examples

```
# a Statistics Canada table the Wayback Machine cannot capture, with the
# archive.org copy made from elsewhere as a mirror
item <- paste0(
  "https://archive.org/download/rmorie-statc",
  "an-indigenous-tables-2026-10"
)
fetchvrf(
  "https://www150.statcan.gc.ca/n1/tbl/csv/41100063-eng.zip",
  file.path(tempdir(), "41100063-eng.zip"),
  mirrors = paste0(item, "/41100063-eng.zip")
)
```

filedgst *SHA-512 and CRC-32 of a file*

Description

`sha512file()` returns the SHA-512 digest of a file's bytes and `crc32file()` its CRC-32 checksum. Use the first to pin a data file for the long term and the second as a fast check that a large file arrived intact (it is the checksum zip archives record). The SHA-256 counterpart is `sha256file()`.

Usage

```
sha512file(path, blkbytes = 1048576L)

crc32file(path, blkbytes = 1048576L)
```

Arguments

`path`	Path to an existing file, a single string.
`blkbytes`	Read size in bytes, a positive whole number (default 1048576). Affects speed only, never the result.

Details

The file is read in blocks of `blkbytes` with `readBin()`, the blocks are joined into one raw vector, and the digest of that vector is computed with `csha512()` or `ccrc32()`. The whole file is therefore held in memory once: memory use grows with the file size. `blkbytes` affects only the reading, never the result. An empty file gives the digest of no bytes (`cf83e135...` for SHA-512, 0 for CRC-32).

Errors: `path` that does not name an existing file (or names a directory); `blkbytes` that is not a positive whole number.

Value

`sha512file()`: a length-1 character vector, 128 lowercase hex characters. `crc32file()`: a length-1 numeric (double) checksum between 0 and `2^32 - 1`.

Examples

```
p <- tempfile()
writeLines("capsule payload", p)

sha512file(p)
crc32file(p)

# The block size is a speed knob and cannot change the digest.
identical(sha512file(p, 16), sha512file(p, 1048576))

# the same as hashing the bytes directly
```

```
identical(sha512file(p), csha512(readBin(p, "raw", file.size(p))))

# a missing file is an error
try(crc32file(file.path(tempdir(), "no-such-file")))
unlink(p)
```

`fipskey` *Assemble a standardised key from raw key material*

Description

Wraps key bytes that came from somewhere else – another implementation, a key store, a file written by an earlier session – in the object `cpssign()` and `cpsvrf()` expect. The lengths are checked against the parameter set, so material for the wrong scheme is refused here rather than producing a signature nothing can verify.

Usage

```
fipskey(scheme, public, secret = NULL)
```

Arguments

`scheme`	Scheme name, as in `fipskg()`.
`public`	Public key, hex or raw.
`secret`	Secret key, hex or raw. Omit for a verification-only key.

Details

The byte layouts are the standards' own, which is what makes this interoperable: an ML-DSA secret key is `rho || K || tr || s1 || s2 || t0` and an SLH-DSA one is `SK.seed || SK.prf || PK.seed || PK.root`, exactly as FIPS 204 and FIPS 205 encode them.

Hex input may be upper or lower case and is stored lower case; raw input is converted to hex. Only the lengths are checked (against `fipssz()`): a secret key that does not belong to the public key is not detected here, and gives signatures that do not verify under that public key. Refusals: an unknown scheme, and material that is not hex of an even length or not exactly the scheme's length (`"must be N bytes of key material"`).

Value

When `secret` is given, a `bricklayer_fips_key` with `public`, `secret` and `scheme`. Otherwise a key of class `bricklayer_fips_public_key` with `public` and `scheme`. Keys are held as lowercase hex, as `fipskg()` returns them.

See Also

`fipskg()`, `fipssz()`.

Examples

```
key <- fipskg("ML-DSA-44")
# the round trip through raw material changes nothing
again <- fipskey("ML-DSA-44", key$public, key$secret)
identical(again$secret, key$secret)

sig <- cpssign("m", again)
cpsvrf("m", sig, fipskey("ML-DSA-44", key$public))

# material of the wrong length is refused
try(fipskey("ML-DSA-65", key$public, key$secret))
```

`fipskg` *Generate a standardised post-quantum signing key*

Description

Generates a key for one of the NIST-standardised signature schemes: ML-DSA (FIPS 204) or SLH-DSA (FIPS 205). Both are implemented in this package, natively, with no system dependency; `pqcbknd()` lists the parameter sets. The key signs capsule manifests, attestations and bundles with `cpssign()`, and its public half, from `fipspubkey()`, is what you give to anyone who should be able to check those signatures with `cpsvrf()`.

Usage

```
fipskg(scheme = "SLH-DSA-SHA2-128s", seed = NULL)

fipspubkey(key)
```

Arguments

`scheme`	Scheme name, one of `pqcbknd()` other than `"xmss-sha256"`. The default, `"SLH-DSA-SHA2-128s"`, is the FIPS 205 small-signature set at the first security level, which rests on SHA-256 alone.
`seed`	Optional raw vector of key-generation seed bytes, of the length `fipssz()` reports for the scheme. Supplying it makes the key reproducible, which is what the standards' test vectors need; the default draws from the operating system's CSPRNG.
`key`	A key from `fipskg()`.

Details

Unlike `pqckg()` 's hash-based key, these are STATELESS: one key signs any number of messages, with no index to track and nothing to persist between signatures.

Which to pick. ML-DSA is small and fast and rests on a lattice assumption. SLH-DSA rests on nothing but the hash function, at the cost of a signature one to two orders of magnitude larger; its

s parameter sets have small signatures and slow signing, its f sets the reverse, and its SHAKE and SHA-2 families are equally strong: pick SHA-2 where a validated SHA-2 implementation is what an auditor will ask about. The default, "SLH-DSA-SHA2-128s", is the conservative choice for a record meant to be checked decades from now: its security rests on the hash function alone, and its signature (7,856 bytes) is the smallest SLH-DSA has. Signing takes about a second; verifying, milliseconds. Pick "ML-DSA-65" for many signatures a second or small signatures.

Key generation expands a random seed (the length fipssz() reports as seed: 32 bytes for ML-DSA, three times the security level in bytes for SLH-DSA) into the key pair exactly as FIPS 204 ML-DSA.KeyGen and FIPS 205 slh_keygen do, so the same seed gives the same key as any other conforming implementation. ML-DSA keys take milliseconds; SLH-DSA keys take from milliseconds (the f sets) to about a second (the s sets, which build a larger top tree), with a spinner on stderr after 0.75 s when a person is watching (see blprog()) and Ctrl-C honoured. Nothing is written to disk: saving the key, and keeping its secret private, is up to you. fipspubkey() drops the secret half, leaving what a verifier needs.

Refusals: an unknown scheme (the message lists the known ones, and points to pqckg() for XMSS), a seed that is not a raw vector of the right length, and for fipspubkey() a key that did not come from fipskg() or fipskey().

Value

fipskg(): a list of class bricklayer_fips_key (also bricklayer_oqs_key) with public and secret (the encoded keys as lowercase hex strings, in the standards' byte layouts) and scheme. Printing it never shows the secret. fipspubkey(): a list of class bricklayer_fips_public_key (also bricklayer_oqs_public_key) with public and scheme.

Security

The implementation is hand-written. Its standardised schemes are checked byte for byte against OpenSSL 3.5 and NIST known-answer vectors in the test suite. Side channels are checked, not assumed: inst/ctcheck runs every operation that touches a secret under valgrind memcheck with the secret marked undefined (the ctgrind method), so a branch or a memory address that depends on it is a reported error, and afterwards scans the dead stack for copies of the secret. Both checks run in CI on every change, with GCC and with Clang. Timing is also measured on x86-64 and arm64 hardware (inst/dudect), power leakage is assessed in simulation under the value and the transition models (inst/tvla), and ML-KEM decapsulation and ML-DSA signing are first-order masked by default. These are checks of this code on those compilers and under those leakage models; the README's security section lists what each check does and does not cover.

References

National Institute of Standards and Technology (2024). Module-Lattice-Based Digital Signature Standard. FIPS 204. DOI 10.6028/NIST.FIPS.204

National Institute of Standards and Technology (2024). Stateless Hash-Based Digital Signature Standard. FIPS 205. DOI 10.6028/NIST.FIPS.205

See Also

pqckg() for the stateful hash-based key, cpssign() which accepts either, fipssz() for the byte lengths.

Examples

```
key <- fipskg("ML-DSA-65")
key$scheme

sig <- cpssign("a manifest digest", key)
cpsvrf("a manifest digest", sig, fipspubkey(key))

# A context string binds the signature to its purpose: the same
# message signed for one context does not verify under another.
sig2 <- cpssign("a manifest digest", key, ctx = "release")
cpsvrf("a manifest digest", sig2, key, ctx = "release")
cpsvrf("a manifest digest", sig2, key, ctx = "staging")

# A fixed seed reproduces the key (for test vectors only: a guessable
# seed is a guessable key).
seed <- as.raw(1:32)
identical(fipskg("ML-DSA-44", seed)$public, fipskg("ML-DSA-44", seed)$public)

# The public half carries no secret.
names(fipspubkey(key))

# An unknown parameter set is refused.
try(fipskg("ML-DSA-99"))
```

`fipsmkg` *Make a Signing Key Inside a FIPS 140-3 Module*

Description

Generates an ML-DSA or SLH-DSA key pair inside the module that `fipsmod()` found. The private key stays in the module's own format (PEM) and never passes through this package's code; `cpssign()` hands it back to the module to sign. The public key is the standard one, so the signatures verify with `cpsvrf()` (this package's implementation), with `cpsxvrf()` (OpenSSL's), and anywhere else FIPS 204 and 205 are implemented.

Usage

```
fipsmkg(scheme = "SLH-DSA-SHA2-128s", mod = fipsmod())
```

Arguments

`scheme`	An ML-DSA or SLH-DSA parameter set (see `fipssz()`), such as `"ML-DSA-65"`; the default, `"SLH-DSA-SHA2-128s"`, is `fipskg()`'s.
`mod`	The module, from `fipsmod()`.

Details

The key prints without its private part. It is an ordinary R object: anything that saves it (`saveRDS()`, a workspace) saves the private key too, so keep it as carefully as any other private key. Signing checks that the module loaded now is the one that made the key and refuses otherwise. XMSS and HQC have no FIPS module and are refused.

Value

A key of class `rmbl_fipsmod_key` (also a `bricklayer_fips_public_key`, so every verifier accepts it): `public` (hex), `scheme`, `module` (name, build, status and certificate) and the private key in `pem`.

See Also

`fipsmod()`, `cpssign()`, `fipskg()` for keys made by this package's own code.

Examples

```
mod <- try(fipsmod(), silent = TRUE)
if (inherits(mod, "rmbl_fipsmod")) {
  key <- fipsmkg("SLH-DSA-SHA2-128s", mod)
  key
}
```

---

fipsmod *Sign Inside a FIPS 140-3 Module*

Description

Finds the OpenSSL FIPS provider (a FIPS 140-3 cryptographic module), loads it on its own, and reports which module it is and whether that module is on the package's list of validated ones. Keys from `fipsmkg()` are then made, and `cpssign()` signs with them, inside that module rather than in this package's own code.

Usage

```
fipsmod(
  openssl = getOption("rmbl.fips.openssl", Sys.which("openssl")),
  conf = getOption("rmbl.fips.conf", Sys.getenv("RMBL_FIPS_CONF"))
)
```

Arguments

`openssl`	Path to the `openssl` program. The default is the option `rmbl.fips.openssl`, else the `openssl` on the `PATH`.
`conf`	Path to an OpenSSL configuration that loads the FIPS provider, or `""` to let the package write one.

Details

The package's own ML-DSA and SLH-DSA are not a validated module, and FIPS mode does not change that. What it gives is a way to sign with a module that is: a module listed as validated, run as this function runs it, is used in its approved mode. Whether a given module is validated is NIST's to say; the package only recognises the exact module names and builds that a certificate names (the file `modules.csv` in the package's `fips` folder), and says "not on the list" for every other one, including a FIPS provider that may well be validated under a certificate the list does not have yet.

The `openssl` program must be 3.5 or later (the first with ML-DSA and SLH-DSA). Unless `conf` names a configuration, the package writes its own to a private temporary folder: it loads only the FIPS provider and the base provider, sets `default_properties = fips=yes`, and makes a configuration error fatal, so nothing can quietly fall back to the default provider. A distribution's own FIPS provider (Red Hat, Fedora, Chainguard) loads as it is; an upstream build is first installed with `openssl fipsinstall`, which runs the module's self-tests. Give `conf` (or set the option `rmbl.fips.conf` or the environment variable `RMBL_FIPS_CONF`) to use the configuration your organisation's FIPS deployment prescribes instead.

The status is the module's own report, matched against the list: it is as true as the `openssl` program run, which is why a key never chooses that program. `cpssign()` loads the module set up now (the options `rmbl.fips.openssl` and `rmbl.fips.conf`) and refuses a key made in another. Every run of the program is stopped after `rmbl.fips.timeout` seconds (default 300). The private key reaches the program through a file in a folder only you can read, removed when the operation ends; a process killed meanwhile leaves it in R's session folder.

Refusals: no `openssl` program, one older than 3.5, a FIPS provider that is missing or does not load, or a `conf` file that does not exist.

Value

A list of class `rmbl_fipsmod`: `name`, `version` and `build` (the module as it reports itself), `status` (`"validated"`, `"validated without PQC"` or `"not listed"`), `cmvp` (the certificate, such as `"#5523"`, or `NA`), `pqc` (whether that certificate covers ML-DSA and SLH-DSA), `openssl`, `openssl_version` and `conf`. It prints as a box. The result is remembered for the session.

See Also

`fipsmkg()`, `cpssign()`, `cpsxvrf()`.

Examples

```
# Only where an OpenSSL 3.5 FIPS provider is installed:
mod <- try(fipsmod(), silent = TRUE)
if (inherits(mod, "rmbl_fipsmod")) {
  print(mod)
  key <- fipsmkg("ML-DSA-44", mod)
  sig <- cpssign("a manifest digest", key)
  cpsvrf("a manifest digest", sig, key)
}
```

fipsmu *Sign an ML-DSA message digest computed elsewhere*

Description

The external-mu interface. `fipsmu()` reduces a message to the 64-byte value mu that is the only thing ML-DSA signing actually consumes; `fipssignmu()` signs that value and `fipsvrfmu()` checks it. Use it when the message and the signing key are in different places: the machine with the data computes mu from the public key, and only those 64 bytes travel to the signer.

Usage

```
fipsmu(key, msg, ctx = NULL, prehash = "none")

fipssignmu(
  key,
  mu,
  determ = FALSE,
  masked = getOption("rmoriebricklayer.masked", TRUE)
)

fipsvrfmu(key, mu, sig)
```

Arguments

key	A key from `fipskg()` or `fipskey()`. An ML-DSA scheme: SLH-DSA has no external-mu interface, because its digest depends on per-signature randomness that the signer chooses.
msg	Length-1 character vector or raw vector.
ctx	Optional context string, as in `cpssign()`.
prehash	Pre-hash function, as in `cpssign()`.
mu	The 64 raw bytes from `fipsmu()`.
determ	As in `cpssign()`.
masked	`TRUE` (the default) for the first-order masked ML-DSA signer, `FALSE` for the plain constant-time one; both give the same signature for the same randomness. The option `rmoriebricklayer.masked` sets the default for a session, so chain- and bundle-heavy code that has no physical attacker in its threat model can take the plain path everywhere with `options(rmoriebricklayer.masked = FALSE)`. SLH-DSA and XMSS signing are not masked and ignore it. Masking costs about 2.3 to 3.4 times the plain time on ML-KEM decapsulation (4 ms for ML-KEM-768 on this package's test machine) and 7 to 10 times on ML-DSA signing (26 ms for ML-DSA-65), depending on the CPU.
sig	A signature from `fipssignmu()` or `cpssign()`.

Details

The point is that the message need never reach the key. A large file can be reduced to mu on the machine that holds it and only mu handed to whatever holds the signing key – a smartcard, a remote signer, another process. mu is not a bare digest: it binds the public key (through `tr = H(pk)`) and the context string, so a mu computed under one key cannot be signed under another to any useful effect.

The resulting signature is an ordinary ML-DSA signature. `cpsvrf()` accepts it, given the same message and context.

mu is FIPS 204's $\mu = \mathrm{SHAKE256}(tr \| M', 64)$ with $tr = \mathrm{SHAKE256}(pk, 64)$ and M' the message prefixed by the domain byte, the context length and the context (or, with a `prehash`, the pre-hash's identifier and digest), as in FIPS 204 Algorithm 7. Signing from mu is Algorithm 7 from that point on, and `fipsvrfmu()` is Algorithm 8 from mu.

Refusals: a key that is not an ML-DSA key from `fipskg()` or `fipskey()` (SLH-DSA keys are refused with `"external mu is an ML-DSA interface"`), `fipssignmu()` with a public key only (`"signing needs a key with its secret half"`) or a `mu` that is not 64 raw bytes. `fipsvrfmu()` returns `FALSE`, not an error, for a mismatched scheme, a wrong-length mu or a malformed signature; it errors only when `signature` is not a `bricklayer_signature` or `key` is not a list.

Value

`fipsmu()`: a raw vector of 64 bytes. `fipssignmu()`: a list of class `bricklayer_signature` with `scheme` and `signature` (hex). `fipsvrfmu()`: `TRUE` or `FALSE`.

References

National Institute of Standards and Technology (2024). Module-Lattice-Based Digital Signature Standard. FIPS 204. DOI 10.6028/NIST.FIPS.204

See Also

`cpssign()`, `fipskg()`.

Examples

```
key <- fipskg("ML-DSA-65")
mu <- fipsmu(key, "a manifest digest", ctx = "release")
length(mu)

sig <- fipssignmu(key, mu)
fipsvrfmu(key, mu, sig)

# the same signature verifies the ordinary way, from the message
cpsvrf("a manifest digest", sig, key, ctx = "release")

# mu is computable from the PUBLIC key alone, which is what lets the
# message stay on the machine that has it
identical(fipsmu(fipspubkey(key), "a manifest digest",
                 ctx = "release"), mu)
```

```
# A different context gives a different mu.
identical(fipsmu(key, "a manifest digest", ctx = "staging"), mu)

# SLH-DSA has no external-mu interface.
try(fipsmu(fipskg("SLH-DSA-SHA2-128f"), "m"))
```

`fipssz` *Byte lengths of a standardised signature scheme*

Description

Reports the sizes fixed by a FIPS 204 or FIPS 205 parameter set, so a caller never has to hard-code them. Use it to check key material before `fipskey()`, to size storage for signatures, or to compare parameter sets: an SLH-DSA signature is 7,856 to 49,856 bytes where an ML-DSA one is 2,420 to 4,627.

Usage

```
fipssz(scheme)
```

Arguments

`scheme`	Scheme name, as in `fipskg()`.

Details

The sizes are the standards' tables (FIPS 204 Table 2, FIPS 205 Table 2) as the compiled code uses them. `seed` is the key-generation input of `fipskg()`; `rnd` is the per-signature randomness (32 bytes for ML-DSA's `rnd`, n bytes for SLH-DSA's `opt_rand`), which the deterministic variants replace with a fixed value. XMSS is not covered: its sizes follow from the tree height of `pqckg()`. An unknown scheme, or anything but one string, is refused.

Value

A named integer vector of five byte counts: `public_key`, `secret_key`, `signature`, `seed` and `rnd` (the per-signature randomness the scheme consumes).

See Also

`fipskg()`, `pqcbknd()`.

Examples

```
fipssz("ML-DSA-65")
fipssz("SLH-DSA-SHAKE-128s")

# An SLH-DSA signature is far larger than an ML-DSA one at the same
# security level, which is the price of dropping the lattice
# assumption.
fipssz("SLH-DSA-SHAKE-128s")[["signature"]] >
  fipssz("ML-DSA-44")[["signature"]]

# Every standardised set at once.
sapply(setdiff(pqcbknd(), "xmss-sha256"), fipssz)["signature", ]

# XMSS has no fixed sizes here.
try(fipssz("xmss-sha256"))
```

---

fnllim *Funnel-plot control limits*

Description

Gives the range within which an area's ratio of observed to expected would fall, for a given expected count, if the area were no different from the overall experience. Plotted against the expected count these limits form a funnel: wide for small areas, narrow for large ones. A funnel plot is the alternative to a league table: it shows directly that a small area's ratio can wander far from one without meaning anything.

Usage

```
fnllim(expd, target = 1, lvls = c(0.95, 0.998))
```

Arguments

`expd`	Numeric vector of expected counts (positive) at which to compute limits, for example a grid from the smallest to the largest area.
`target`	Number, default `1`: the ratio the limits are centred on. One is the overall experience.
`lvls`	Numeric vector of two-sided coverage levels in (0, 1), default `c(0.95, 0.998)`; one pair of limits per level.

Details

For expected count E, target ratio ρ and coverage $1 - a$, the limits are

$$\frac{Q(a/2;\ \rho E)}{E}, \quad \frac{Q(1 - a/2;\ \rho E)}{E}$$

with Q the Poisson quantile function. They are exact Poisson quantiles divided by the expected count, so they are the discrete counterpart of the usual normal funnel and stay correct at the small expected counts where the normal version goes below zero. Because the Poisson is discrete the limits move in steps of $1/E$. The usual pair of levels, 0.95 and 0.998, corresponds to about two and three standard deviations.

Errors: non-positive, infinite or missing expected counts (missing values are dropped); a non-positive `target`; a level outside (0, 1).

Value

A data frame with one row per expected count and level, sorted by level then expected count, with numeric columns `expected`, `level`, `lower` and `upper` (the limits on the ratio scale).

References

Spiegelhalter, D. J. (2005). Funnel plots for comparing institutional performance. *Statistics in Medicine* 24(8), 1185-1202. DOI 10.1002/sim.1970

Advanced Statistics in Criminology and Criminal Justice discusses the funnel plot as the display of the relationship between an estimate and the sample size behind it.

Lawson, A. B. *Using R for Bayesian Spatial and Spatio-Temporal Health Modeling*. Chapman and Hall/CRC, on the Poisson counts these limits are built from.

See Also

`stdinc`, `ebrates`

Examples

```
# The funnel narrows as the expected count grows, which is the whole
# point: a ratio of 2 means nothing at an expected count of 2 and a
# great deal at an expected count of 200.
fnllim(c(2, 20, 200))

# one 90 percent funnel around a target ratio of 1.2
fnllim(c(5, 50), target = 1.2, lvls = 0.90)

# a funnel drawn with base graphics
e <- c(1, 2, 5, 10, 20, 50, 100, 200)
f <- fnllim(e, lvls = 0.95)
plot(f$expected, f$upper, type = "s", log = "x", ylim = c(0, 4),
     xlab = "expected count", ylab = "ratio")
lines(f$expected, f$lower, type = "s")
abline(h = 1, lty = 2)

try(fnllim(10, lvls = 1))
```

fpeqv *Everyday equivalents of a quantity of CO2-equivalent*

Description

Converts grams of CO2-equivalent into two everyday comparisons: kilometres driven in an average European passenger car (175 gCO2/km) and tree-months of carbon sequestration (917 gCO2 per tree per month), the two comparisons the Green Algorithms calculator prints, with their sources. Use it to make the `co2eg` of `co2fp` easier to read in a report.

Usage

```
fpeqv(co2eg)
```

Arguments

`co2eg`	Numeric scalar, grams of CO2-equivalent; finite and non-negative (for example the `co2eg` element of a `co2fp` result).

Details

The conversions are plain divisions: $\mathrm{km} = g/175$ and $\mathrm{tree\ months} = g/917$. Both factors are averages and the comparisons are illustrations, not offsets: a tree-month is the carbon one mature tree takes up in a month, not a promise that planting one cancels the emission. Errors when `co2eg` is not a single finite, non-negative number.

Value

A list with `carkm` (numeric, kilometres), `treemon` (numeric) and `sources` (named character vector, `car` and `tree`, naming each factor's source).

References

Green Algorithms data v3.1, context.csv: car factor from Helmers et al. (2019) Atmospheric Environment and UK BEIS 2019 conversion factors; tree sequestration from Bernal et al. (2018) via the same file.

See Also

`co2fp`

Examples

```
fpeqv(1200)

# the footprint of a small computation, in everyday terms
fp <- co2fp(sum(sqrt(seq_len(2e5))), loc = "CA-ON",
                        cpupwrw = 45, memgb = 8)
```

```
unlist(fpeqv(fp$co2eg)[c("carkm", "treemon")])

# a vector is refused: convert one total at a time
try(fpeqv(c(10, 20)))
```

`freqtbl` *Frequency table for one column*

Description

Counts how often each value of a column occurs, with percentages of all rows and of the non-missing rows, and missing values counted as their own row. It is the first look at a categorical variable (a charge type, a region, a response category), and the counterpart of `janitor::tabyl()`.

Usage

```
freqtbl(data, col = NULL, maxlvl = 25L, sort = TRUE)
```

Arguments

`data`	A data frame, or a vector.
`col`	Column name (a string), when `data` is a data frame; ignored for a vector.
`maxlvl`	Maximum value rows to return, most frequent first (default 25). The remainder are folded into one `"(other)"` row so the percentages still sum to 100.
`sort`	Logical: sort by descending count (default `TRUE`); `FALSE` keeps the sorted order of the values.

Details

Two percentage columns, because both questions get asked and conflating them is how missingness gets hidden: `pct` is the share of ALL rows, `pctvalid` the share of rows where the value is present. When a column is 40% missing those two differ enormously, and only the second describes the values that are actually there.

Values are tabulated as text (`as.character()`), so numbers and factors are counted by their printed value, and factor levels that do not occur get no row. Rows are sorted by descending count (ties in the order `table()` gives) or kept in sorted value order with `sort = FALSE`. Beyond `maxlvl` distinct values, the rest are folded into one `"(other)"` row so the percentages still sum to 100. The missing row, when there is one, comes last, with `value` `NA` and `pctvalid` `NA`. An empty input gives a table with no rows. The print method adds a bar column.

Errors: a data frame without `col`; a `col` that is not in `data`; `maxlvl` not one positive whole number.

Value

A data frame of class `c("bricklayer_freq", "data.frame")` with columns `value` (character), `n` (integer count), `pct` (numeric, percent of all rows) and `pctvalid` (numeric, percent of non-missing rows; `NA` on the missing row).

See Also

profcols() for every column at once.

Examples

```
df <- data.frame(grade = c("a", "b", "b", "c", NA, "b"),
                 stringsAsFactors = FALSE)
freqtbl(df, "grade")

# The two percentage columns differ exactly by the missingness.
freqtbl(df, "grade")[, c("pct", "pctvalid")]

# A vector works directly.
freqtbl(c(1, 1, 2, 3, 3, 3))

# Sorted value order rather than frequency order.
freqtbl(c("c", "a", "b", "a"), sort = FALSE)

# Long tails are folded so the percentages still total 100.
set.seed(1)
ft <- freqtbl(sample(letters, 500, TRUE), maxlvl = 5)
ft
sum(ft$pct)

# a data frame needs a column name
try(freqtbl(df))
```

---

frmaud *How deficit-framed is a set of results?*

Description

Counts how many of the indicators in a report measure distress, deficit or disparity and how many measure wellness or strength, and names any group reported on deficits alone.

Usage

```
frmaud(x)
```

Arguments

x	A welprof result, whose rows are counted by side.

Details

Walter (2016) shows that statistics about Indigenous peoples cluster around five Ds: disparity, deprivation, disadvantage, dysfunction and difference. Results built only of those describe Indigenous peoples as their problems. This audit does not measure anything about people; it measures the report: the share of its indicators that are deficit-framed, overall and per group, so a report can be balanced before it is published. Pair each gap with a strength-framed level (`welprof`, `dualcont`).

Value

A list of class `rmbl_framing`: `bygrp` (a data frame of `group`, `wellness`, `distress`, `shrdef`), `shrdef` overall and `defonly` (the groups with no wellness indicator).

References

Walter M (2016). Data politics and Indigenous representation in Australian statistics. In Kukutai T, Taylor J (eds), Indigenous Data Sovereignty: Toward an Agenda, 79-97. ANU Press. DOI 10.22459/caepr38.11.2016.05

Walter M, Andersen C (2013). Indigenous Statistics: A Quantitative Research Methodology. Left Coast Press. DOI 10.4324/9781315426570

Examples

```
wb <- indiex("wellbeing")
w <- welprof(
  wb,
  "group",
  "indicator",
  "estimate",
  "lower",
  "upper",
  well = "Always or often has a hopeful outlook",
  distress = c(
    "Rarely or never has a hopeful outlook",
    "Life satisfaction rating between 0 and 5",
    "Fair or poor perceived health"
  )
)
frmaud(w)

# your own table: made-up percentages for two communities, with limits
own <- data.frame(
  community = rep(c("Lakeside", "Riverbend"), each = 4),
  item = rep(c("Hopeful", "Satisfied", "Rarely hopeful", "Low satisfaction"),
             2),
  pct = c(64, 71, 12, 18, 58, 66, 16, 22),
  lo = c(59, 66, 9, 14, 52, 60, 12, 17),
  hi = c(69, 76, 15, 22, 64, 72, 20, 27)
)
frmaud(welprof(own, "community", "item", "pct", "lo", "hi",
               well = c("Hopeful", "Satisfied"),
               distress = c("Rarely hopeful", "Low satisfaction")))
```

`fromjs` *Parse JSON into R objects (jsonlite's fromJSON, natively)*

Description

Reads JSON text, a JSON file or a JSON address on the web into R objects, with the same simplification rules as jsonlite's `fromJSON()`: an array of numbers becomes a numeric vector, an array of records a data frame, an array of equal-length arrays a matrix. The parser is this package's own, so it works without jsonlite, and it is stricter in two places that matter for provenance records: a repeated key is an error by default, and a whole number too large for a double is flagged.

Usage

```
fromjs(
  txt,
  simplifyVector = TRUE,
  simplifyDataFrame = simplifyVector,
  simplifyMatrix = simplifyVector,
  flatten = FALSE,
  bigint_as_char = FALSE,
  simplify = NULL,
  duplicate_keys = c("error", "keep"),
  bigint_warn = TRUE,
  ...
)
```

Arguments

`txt`	JSON text (a character vector, joined with newlines), a file path, an `http(s)` URL, or a connection. A URL goes through the same check and the same transport as every other fetch in the package (public https host, redirects re-checked, body capped).
`simplifyVector`	Logical scalar, default `TRUE`: turn arrays of scalars into atomic vectors.
`simplifyDataFrame`	Logical scalar, default `simplifyVector`: turn arrays of records into data frames.
`simplifyMatrix`	Logical scalar, default `simplifyVector`: turn arrays of equal-length arrays into matrices.
`flatten`	Logical scalar, default `FALSE`. `TRUE` expands nested data frame columns into columns named `outer.inner`.
`bigint_as_char`	Logical scalar, default `FALSE`: integers beyond 2^53 come back as strings.
`simplify`	Legacy, default `NULL`: `FALSE` turns every simplification off.
`duplicate_keys`	What to do with an object that repeats a key: `"error"` (default) refuses the document, `"keep"` returns both values under the repeated name, as jsonlite does.

bigint_warn	Logical scalar, default TRUE: warn (once per document) when a whole number beyond 2^53 is read into a double; the package's own readers of API payloads turn this off, since long numeric identifiers are routine there.
...	Ignored, for call compatibility with jsonlite.

Details

A single string shorter than 2,084 bytes that is not valid JSON is taken as a location: an http(s) address is fetched (a public https host only, redirects re-checked, 60 second timeout, body capped at 16 MiB), otherwise an existing file is read. Anything else is parsed as JSON text; a character vector is joined with newlines. A connection is read to its end (and closed when this call opened it).

Without simplification (simplifyVector = FALSE) objects become named lists, arrays unnamed lists, null NULL, and numbers integer or double. Whole numbers beyond 2^{53} cannot be held exactly in a double: they are read as doubles with a warning, or kept as text with bigint_as_char = TRUE.

Errors: txt not character or a connection; invalid JSON (the message gives the character offset); a repeated key under duplicate_keys = "error"; an address that is not a public https host, that cannot be reached or that answers with an HTTP error.

Value

An R object: a list, an atomic vector, a data frame, a matrix or NULL, depending on the JSON and the simplification options.

See Also

tojs() for the reverse.

Examples

```
fromjs('[{"a":1,"b":"x"},{"a":2,"b":"y"}]')
fromjs('[[1,2],[3,4]]')

# no simplification: nested lists exactly as in the document
str(fromjs('{"a": [1, 2], "b": {"c": null}}', simplifyVector = FALSE))

# from a file
f <- tempfile(fileext = ".json")
writeLines('[{"a":1},{"a":2,"b":"x"}]', f)
fromjs(f)

# a large identifier kept exactly, as text
fromjs('[12345678901234567890]', bigint_as_char = TRUE)

# a repeated key is refused unless asked for
try(fromjs('{"a": 1, "a": 2}'))
fromjs('{"a": 1, "a": 2}', duplicate_keys = "keep")
```

`fylbl` *Name a fiscal year by the years it spans*

Description

Turns fiscal years keyed by their END year into labels that name both calendar years. Ontario's inmate data (and many government series) key on the fiscal year's end year, so 2023 is the fiscal year running from April 2022 to March 2023; this renders that as `"2022/23"`, which a reader cannot mistake for the calendar year 2023. Pass it to `yoylbl()` to label a change table.

Usage

```
fylbl(endyear, sep = "/", short = TRUE)
```

Arguments

`endyear`	The fiscal year's end year, as a whole number or a string; a vector of them.
`sep`	Separator between the two years, a string; default `"/"`.
`short`	Logical. `TRUE` (the default) abbreviates the second year to two digits, as in `"2022/23"`; `FALSE` writes it in full, as in `"2022/2023"`.

Details

Each value is converted with `as.integer(as.character(endyear))`, so a string such as `"2023"` works; a fractional number is refused rather than truncated (2023.7 is an error). The label is `(y - 1)`, `sep`, then `y` itself or its last two digits. Values that cannot be read as a year give `NA`. The two-digit form is unambiguous within a century (2000 gives `"1999/00"`).

Errors: `endyear` that is neither numeric nor character (for example logical); a numeric year below 1; a numeric year that is not whole.

Value

A character vector of labels, the same length as `endyear`, with `NA` where a value is not a year.

Examples

```
fylbl(2019:2023)

fylbl(2023, short = FALSE)

fylbl(2023, sep = "-")

# strings work; something that is not a year gives NA
fylbl(c("2023", "n/a", NA))

# logical input is refused
try(fylbl(TRUE))
```

genexpo *Generations of residential school attendance in a family*

Description

Codes the intergenerational exposure used in the epidemiology of Indian Residential Schools: how many generations before the respondent attended, 0 (neither parents nor grandparents), 1 (parents or grandparents) or 2 (both a parent and a grandparent), as in McQuaid et al. (2017).

Usage

```
genexpo(parent, gparent, partial = c("na", "lowerbnd"))
```

Arguments

`parent`, `gparent`	Logical vectors: did a parent, did a grandparent attend.
`partial`	What a known attendance with the other generation unknown counts as: `"na"` (not known, the default) or `"lowerbnd"` (1).

Details

Each answer is `TRUE` (attended), `FALSE` (did not) or `NA` (not known). An unknown answer may hide an attendance, so the count is unknown (`NA`) whenever an unknown answer could change it: always when the known answer is `FALSE`, and, with `partial = "na"` (the default), also when the known answer is `TRUE` (the count is 1 or 2). `partial = "lowerbnd"` codes that case 1, a lower bound: it keeps the person but can classify them a level too low, which biases a trend toward no effect.

Value

An integer vector of 0, 1, 2 or `NA`.

References

McQuaid RJ, Bombay A, McInnis OA, Humeny C, Matheson K, Anisman H (2017). Suicide ideation and attempts among First Nations peoples living on-reserve in Canada: the intergenerational and cumulative effects of Indian Residential Schools. Canadian Journal of Psychiatry 62(6):422-430. DOI 10.1177/0706743717702075

Bombay A, Matheson K, Anisman H (2014). The intergenerational effects of Indian Residential Schools: implications for the concept of historical trauma. Transcultural Psychiatry 51(3):320-338. DOI 10.1177/1363461513503380

See Also

`exptrnd`.

Examples

```
genexpo(c(TRUE, FALSE, TRUE, NA, FALSE), c(TRUE, FALSE, FALSE, TRUE, NA))
genexpo(c(TRUE, FALSE, TRUE, NA, FALSE), c(TRUE, FALSE, FALSE, TRUE, NA),
                    partial = "lowerbnd")
```

---

ghann *GitHub workflow annotations, sticky comment and outputs of a capsule check*

Description

Plain text for a GitHub Actions step, from a cpschk result: nothing here talks to GitHub, the caller prints or posts it. `ghann()` gives one workflow command per failed check, an `::error` for an error row and a `::warning` for a warn or drift row (notes give none), naming the file and, where the row has one, the line, with the data and properties escaped as GitHub requires. `ghcmt()` gives the body of a sticky pull-request comment: the marker line that finds it again, the Markdown report and a link to the run, cut to fit GitHub's 65,536-character limit. `ghout()` gives the step outputs of the rmbl action as a named list (print each as `name=value` to `$GITHUB_OUTPUT`).

Usage

```
ghann(x)

ghcmt(
  x,
  runurl = NULL,
  failon = NULL,
  marker = "<!-- rmbl-action -->",
  limit = 65000L
)

ghout(x, report = "", json = "", comment = "off")
```

Arguments

`x`	An `rmbl_cpschk` result.
`runurl`	Character scalar or `NULL`: the run's address, linked at the end of the comment.
`failon`	Character scalar or `NULL`: the fail-on level, named in the report's last line.
`marker`	Character scalar, the comment's first line (an HTML comment, invisible on GitHub).
`limit`	Integer: the longest body; a longer one is cut and says so.
`report, json`	Character scalars: the paths of the report and the JSON written, passed through as outputs.
`comment`	Character scalar: what became of the comment (`off`, `dry-run`, `skipped`, `created`, `updated` or `failed`), passed through.

Value

ghann(): a character vector, one line per annotation. ghcmt(): one string. ghout(): a named list with status, report, json, changed (the capsules whose source bytes changed, comma-separated), verified (capsules without an error), failed, manifest-changed ("true" or "false") and comment. Error: x is not a cpschk() result.

See Also

cpschk.

Examples

```
root <- tempfile("capsules-")
d <- usecpstmpl(file.path(root, "one"), ex = TRUE)
invisible(cpsrfrsh(d))
p <- loadprov(file.path(d, "data_provenance.json"))
cat("1,2\n", file = file.path(d, p$resource$filename), append = TRUE)
x <- cpschk(root, fp = FALSE)
ghann(x)
substr(ghcmt(x, runurl = "https://github.com/o/r/actions/runs/1"), 1, 80)
str(ghout(x, report = "rmbl-report.md", json = "rmbl-results.json"))

# one output per line, as $GITHUB_OUTPUT takes them
o <- ghout(x)
cat(sprintf("%s=%s\n", names(o), unlist(o)), sep = "")

try(ghann(list()))
```

---

grdbin *Refuse a categorical column where a numeric 0/1 treatment is required*

Description

Checks that a column meant as a binary treatment or outcome indicator is numeric 0/1 and not a factor or character column. Coercing a factor with as.numeric() returns the level indices (1, 2, ...) rather than the data, and a mis-ordered level silently flips every observation; this guard stops before that happens. Call it at the top of any function that needs a 0/1 column.

Usage

```
grdbin(x, col)
```

Arguments

x	The column to check.
col	Character string: the column's name, used only in the error message.

Details

The checks are: `x` is an atomic vector; it is not a factor or character vector; its distinct non-missing values are all `0` or `1`. Logical vectors pass, because `TRUE` and `FALSE` compare equal to `1` and `0`. Missing values are ignored. Each failure is an error that names the column.

Value

`TRUE`, invisibly, when the column is a 0/1 indicator; otherwise an error.

See Also

`grdrcd` to build an explicit indicator.

Examples

```
grdbin(c(0, 1, 1, 0, NA), "treated")
grdbin(c(TRUE, FALSE), "arrested")

# a factor is refused rather than coerced
try(grdbin(factor(c("control", "treated")), "treated"))

# so is a numeric column with other values
try(grdbin(c(0, 1, 2), "treated"))
```

---

grdlvls *Build a factor with explicit, verified levels*

Description

Creates a factor whose levels you write out in full and in order, instead of letting `factor()` sort them alphabetically. The first level is the reference category of every regression fitted on the factor, so writing the levels out decides the baseline on purpose. Values outside the declared levels are an error, and you can assert which level is the reference so that a later edit to `lvls` cannot change it unnoticed.

Usage

```
grdlvls(x, lvls, ref = NULL)
```

Arguments

`x`	Character or factor vector. A factor is read through its labels.
`lvls`	Character vector of every allowed level, in the intended order. The first element is the reference category.
`ref`	Optional character string, default `NULL` (no check). When given it must equal `levels[1]`; use it to state the intended baseline next to the code that fits the model.

Details

The function checks that every non-missing value of `x` is in `lvls`, then returns `factor(x, levels = levels)`. Declared levels that do not occur are kept as empty levels (they still appear in tables with a count of zero).

Errors:

- `lvls` not a non-empty character vector, or containing `NA`;
- a value of `x` not in `lvls` (listed in the message);
- `ref` given and not identical to `levels[1]`.

Value

A factor with exactly the levels in `levels`, in that order.

See Also

`grdrcd`, `audcats`.

Examples

```
# reference category chosen on purpose, not by alphabet
g <- grdlvls(c("White", "Black", "White"), c("White", "Black"), "White")
levels(g)

# a declared level that does not occur is kept, with a zero count
table(grdlvls(c("a", "a"), c("a", "b")))

# a stray value, or a reference that is not first, is refused
try(grdlvls(c("White", "Blak"), c("White", "Black")))
try(grdlvls(c("White", "Black"), c("White", "Black"), ref = "Black"))
```

`grdrcd` *Recode a categorical column by explicit name-to-name mapping*

Description

Recodes a categorical variable with a mapping written out by name, `old_label = "new_label"`, so that the recode cannot depend on the order of the levels. Any value that the mapping does not cover is an error unless you list it in `keep`; nothing is silently set to `NA` or passed through. The result records the mapping and its SHA-256, so the recode that was applied can be shown later. Use it for collapsing or renaming categories (race, offence type, region) before analysis.

Usage

```
grdrcd(x, map, keep = character())
```

Arguments

x	A factor or character vector. Other atomic vectors are compared by matching the values that as.character() gives. Ordered and unordered factors are treated the same: both are read through their labels, and the result is character either way.
map	Named character vector, c(old_label = "new_label", ...). Every name must be non-empty.
keep	Character vector, default empty. Labels allowed to pass through unchanged.

Details

Let S be the set of distinct non-missing values of x. The function refuses unless $S \subseteq$ names(mapping) $\cup$ keep. Each value found in names(mapping) is replaced by its mapped label; values in keep are returned unchanged; NA stays NA. Several old labels may map to the same new label (a collapse). Matching is exact: case and whitespace count.

The checksum is csha256 of the string "old1=new1;old2=new2;..." built from the mapping in its given order, so the same mapping written in a different order has a different checksum.

Errors: map not a character vector with non-empty names; any observed value that is neither mapped nor kept (the message lists them).

relbl is the same check but returns a factor; vrfrcd proves the result against the declared mapping; rcdmnf records it.

Value

A character vector the same length as x, with a recode_audit attribute: a list with mapping (the mapping as given), kept (the keep vector) and checksum (hexadecimal SHA-256 of the mapping).

See Also

vrfrcd to prove the recode, grdlvls to fix the levels, rcdmnf to record the chain.

Examples

```
# rename three codes
x <- c("W", "B", "O", "W", NA)
grdrcd(x, c(W = "White", B = "Black", O = "Other"))

# collapse two categories and let one pass through unchanged
off <- c("assault", "robbery", "theft", "fraud")
grdrcd(off, c(assault = "violent", robbery = "violent",
                    theft = "property"), keep = "fraud")

# a value with no mapping is an error
try(grdrcd(x, c(W = "White", B = "Black")))
```

`healchg` *Healing as a process: change in published indicators between two periods*

Description

For each group and indicator, the change between two survey periods with its confidence interval, wellness indicators first: change over time read as Indigenous scholars describe mental health, as a process of healing, not as a fixed deficit.

Usage

```
healchg(
  data,
  grp,
  ind,
  prd,
  est,
  lower,
  upper,
  from,
  to,
  well = NULL,
  qual = NULL,
  conflvl = 0.95,
  distress = NULL
)
```

Arguments

`data`	A data frame, one row per group, indicator and period.
`grp`, `ind`, `prd`, `est`, `lower`, `upper`	Column names: the change is in the estimate's own units (percentage points for percentages).
`from`, `to`	The two periods.
`well`, `distress`	Optional indicators of wellness (listed first) and of distress; with `distress` given, any other indicator is marked `"other"`.
`qual`	Optional column of quality flags: `"E"` rows are marked use with caution, and suppressed ones (`"F"`, `"x"`, `".."`) get no estimate.
`conflvl`	Confidence level of each change's interval (default 0.95).

Details

Stewart (2008) describes Indigenous mental health as wellness and as a process of healing; a process is seen in change. The change $p_2 - p_1$ has the interval $p_2 - p_1 \pm z\sqrt{s_1^2 + s_2^2}$, with the standard errors from the published intervals and the two periods' samples independent (separate cycles of a

survey are). A change is only as comparable as the two periods' questions and designs: Statistics Canada flags breaks in a series, and a change across one is not a change in the people.

Value

A data frame of class `rmbl_healing_change`: `group`, `indicator`, `side`, `from`, `to`, `change`, `lower`, `upper`, `usewithcau`.

Acknowledgement

This estimator puts to quantitative use the framework of Professor Suzanne Stewart (Yellowknives Dene First Nation, University of Toronto).

See the acknowledgements of the `first-nations-metis-inuit-health` vignette.

References

Stewart SL (2008), see `holbal`.

See Also

`welprof`.

Examples

```
# 2015/2018 to 2019/2022, adults (table 13-10-0924-01)
h <- indiex("health")
belong <- paste0(
  "Sense of belonging to local community, ",
  "somewhat strong or very strong"
)
h <- h[
  h$age == "Total, 18 years and over" &
    h$indicator %in%
      c(
        "Perceived mental health, very good or excellent",
        belong,
        "Mood disorder",
        "Anxiety disorder"
      ),
]
hc <- healchg(
  h,
  "group",
  "indicator",
  "period",
  "estimate",
  "lower",
  "upper",
  from = "2015/2018",
  to = "2019/2022",
  qual = "quality",
  well = c("Perceived mental health, very good or excellent", belong)
```

```
)
hc
plot(hc)

# your own table: made-up percentages for one community in two surveys
own <- data.frame(
  community = "Lakeside",
  item = rep(c("Good mental health", "Anxiety"), 2),
  survey = rep(c("2018", "2023"), each = 2),
  pct = c(55, 14, 61, 11),
  lo = c(50, 11, 56, 8),
  hi = c(60, 17, 66, 14)
)
healchg(own, "community", "item", "survey", "pct", "lo", "hi",
        from = "2018", to = "2023", well = "Good mental health",
        distress = "Anxiety")
```

`hilltlidx` *Tail index of a heavy-tailed count*

Description

The Clauset-Shalizi-Newman maximum-likelihood estimator for a discrete power law above a threshold. An exponent near 2 or below means the mean is barely defined and the observed maximum is not informative about the next one, which is the substantive point when a few units dominate a total. Use it on counts per unit (placements per person, incidents per address) to judge whether the largest values are a heavy tail rather than a few outliers, and read `ks` and `reliable` before quoting the exponent.

Usage

```
hilltlidx(x, xmin = NULL, discrete = TRUE, approx = FALSE, mintl = 3L)
```

Arguments

`x`	Numeric vector of positive values, one per unit.
`xmin`	Numeric scalar or `NULL` (the default): the threshold at and above which the power law is fitted. A power law is a statement about the tail, so a threshold is required; `NULL` takes the 50th largest value (which leaves at least 50 observations in the tail), or the minimum if there are fewer than 50 values.
`discrete`	Logical scalar, default `TRUE`: whether the quantity is integer-valued. A count is, and then the likelihood maximised is the zeta distribution's, whose normalising constant is a Hurwitz zeta.
`approx`	Logical scalar, default `FALSE`. For discrete data, whether to use the closed-form continuity-corrected estimator instead of maximising the exact likelihood. It is much faster and much worse: the correction is an asymptotic approximation in `xmin`, and at `xmin` = 1 – where administrative counts start – it returns about 2.0 from data generated with an exponent of 2.5. Off by default for that reason.

`mintl` Integer scalar, default `3`: the fewest tail observations for which an estimate is reported at all (values below 2 are raised to 2). Below it there is nothing to estimate from and `alpha` is `NA`.

Details

For a power law $p(x) \propto x^{-\alpha}$ above x_{min}:

- discrete data, exact (the default): the log-likelihood of the zeta distribution truncated at x_{min}, $-n \log \zeta(\alpha, x_{min}) - \alpha \sum \log x_i$, is maximised over α in $(1.0001, 25)$ with `stats::optimize()`; ζ is `hurwzeta()`. The standard error comes from the observed information, by a numerical second derivative.
- continuous data (`discrete = FALSE`): the Hill estimator $\hat{\alpha} = 1 + n / \sum \log(x_i / x_{min})$, with standard error $(\hat{\alpha} - 1)/\sqrt{n}$.
- discrete data with `approx = TRUE`: the same closed form with $x_{min} - 1/2$ in place of x_{min}.

`ks` is the Kolmogorov-Smirnov distance between the fitted distribution and the tail, computed once per distinct value so that ties (most counts are ties) are handled correctly.

Only finite positive values of `x` are used; the rest are dropped silently. When the tail has fewer than `mintl` values, or every tail value is the same, `alpha` is `NA` and `method` says why. An estimate at the optimiser's bound is flagged in `method` and is not `reliable`. Errors: no positive finite value in `x`; `xmin` that is not a single positive number (logical, factor and text are refused); `discrete` or `approx` not `TRUE` or `FALSE`.

Value

A list with `alpha` (numeric, the exponent), `se` (its standard error), `xmin` (the threshold used), `ntl` (integer, the number of values at or above it), `ks`, `reliable` (logical) and `method` (character, the estimator used or why there is no estimate). `ks` is the Kolmogorov-Smirnov distance between the fitted tail and the data: a large value means the tail is not a power law, whatever `alpha` came out as. `reliable` is `FALSE` when fewer than 50 observations lie in the tail, which is the sample size Clauset, Shalizi and Newman give as the point below which the estimate should not be leaned on – it is reported rather than enforced, because the right response to a short tail is a wider interval, not a refusal.

References

Clauset, A., Shalizi, C. R. and Newman, M. E. J. (2009). Power-law distributions in empirical data. *SIAM Review* 51(4), 661-703. The estimator and its threshold guidance are theirs; the continuity correction they give in closed form is an asymptotic approximation in `xmin`, which is why the exact likelihood is maximised here instead. (Not in the local corpus; cited from the published paper.)

Examples

```
# A continuous Pareto tail with exponent 2.5.
set.seed(1)
x <- (1 - stats::runif(5000))^(-1 / 1.5)
round(hilltlidx(x, xmin = 1, discrete = FALSE)$alpha, 2)

# A discrete power law, where the exact likelihood is needed: the
```

```
# closed-form correction is badly biased at a threshold of one.
k <- 1:10000
p <- k^(-2.5) / sum(k^(-2.5))
set.seed(2)
z <- sample(k, 5000, replace = TRUE, prob = p)
c(exact = round(hilltlidx(z, xmin = 1)$alpha, 2),
  approx = round(hilltlidx(z, xmin = 1, approx = TRUE)$alpha, 2))

# A short tail still returns an estimate, marked as not to be leaned
# on, and with a standard error that says the same thing.
short <- hilltlidx(c(3, 4, 5, 9), xmin = 3)
c(alpha = round(short$alpha, 2), n = short$ntl,
  reliable = short$reliable)

# Below `mintl` there is nothing to estimate from.
hilltlidx(c(3, 4), xmin = 3)$alpha

# the threshold has to be positive
try(hilltlidx(z, xmin = 0))
```

`holbal` *Holistic balance: how evenly wellness is held across the domains of a whole person*

Description

For each group, from published percentages of wellness in each holistic domain (physical, emotional, mental, spiritual, or the domains a community names), the overall level of wellness and how balanced it is across the domains, with Monte Carlo intervals from the published ones.

Usage

```
holbal(
  data,
  grp,
  domain,
  est,
  lower,
  upper,
  w = NULL,
  qual = NULL,
  reps = 10000,
  seed = 1,
  conflvl = 0.95
)
```

Arguments

`data`	A data frame, one row per group and indicator (as `scests` gives).
`grp`, `domain`, `est`, `lower`, `upper`	Column names: the group, the domain of the indicator (any set of domains: Medicine Wheel, Te Whare Tapa Wha, a Nation's own), the percentage of wellness (0 to 100, strictly inside; multiply proportions by 100) and its published limits.
`w`	Optional named weights, one per domain (they are rescaled to sum to one).
`qual`	Optional column of quality flags for the estimates.
`reps`	Monte Carlo draws for the intervals (default 10,000).
`seed`	Seed for the draws, so the intervals are reproducible (default 1).
`conflvl`	Confidence level of the balance intervals (default 0.95).

Details

Indigenous understandings of mental health are holistic: wellness is a balance among the domains of a person, and illness a loss of that balance (Stewart 2008; Gabriel, Mirza and Stewart 2022). A single indicator, or an average of indicators, cannot show balance: a group with 80% in two domains and 20% in two others has the same average as one with 50% in all four. This method, which is ours and inspired by that work rather than taken from it, measures balance as the ratio of the weighted geometric to the weighted arithmetic mean of the domain levels p_k:

$$B = \exp\left(\sum_k w_k \log p_k\right) / \sum_k w_k p_k,$$

one less Atkinson's (1970) index with inequality aversion 1. $B = 1$ exactly when every domain with a positive weight is at the same level, and it falls toward 0 as one domain falls behind the others; it does not depend on the overall level, which is reported beside it (`level`, $\sum_k w_k p_k$) with the weakest domain.

The weights `w` say how much each domain counts. They are equal by default; a community that holds its own view of the domains sets its own (the CARE principle of authority to control, Carroll et al. 2020), and the result says which weights it used.

Several indicators in one domain are averaged, their standard errors combined as independent (they come from the same respondents, so that standard error is too small). The interval for B comes from `reps` draws of every domain level from a normal distribution on the logit scale with the standard error of its published interval, the domains independent. B is at most 1 and reaches 1 only at equal levels, so noise in the levels tends to pull the draws below $\hat{B}$ (B is not concave; the direction of the bias is seen in the draws, not guaranteed): the percentile interval is shifted back by the draws' mean bias ($\bar{B}^* - \hat{B}$), kept inside $(0, 1]$ and widened if need be to contain $\hat{B}$. The level's interval is normal. A group with a suppressed or missing domain, or one whose published interval misses its own estimate, has no balance: the domain is not imputed.

What the domains mean, and which indicators stand for each, is for the communities to say; the published tables have few measures of spiritual wellness, and a domain with no fitting indicator should be left out rather than filled with another.

Value

A data frame of class `rmbl_holistic_balance`: `group`, `level`, `lvllo`, `lvlup`, `balance`, `ballo`, `balup`, `weadom`, `usewithcau`; the attributes `domains` (the matrix of domain levels, groups by domains) and `weights`.

Acknowledgement

This estimator puts to quantitative use the framework of Professor Suzanne Stewart (Yellowknives Dene First Nation, University of Toronto).

See the acknowledgements of the `first-nations-metis-inuit-health` vignette.

References

Atkinson AB (1970). On the measurement of inequality. Journal of Economic Theory 2(3):244-263. DOI 10.1016/0022-0531(70)90039-6

Carroll SR, Garba I, Figueroa-Rodriguez OL, et al. (2020). The CARE Principles for Indigenous Data Governance. Data Science Journal 19:43. DOI 10.5334/dsj-2020-043

Gabriel MD, Mirza S, Stewart SL (2022). Exploring mental health and holistic healing through the life stories of Indigenous youth who have experienced homelessness. International Journal of Environmental Research and Public Health 19(20):13402. DOI 10.3390/ijerph192013402

Stewart SL (2008). Promoting Indigenous mental health: cultural perspectives on healing from Native counsellors in Canada. International Journal of Health Promotion and Education 46(2):49-56. DOI 10.1080/14635240.2008.10708129

See Also

`welprof`, `dualcont`.

Examples

```
# four domains from table 13-10-0924-01, 2019/2022, adults: which indicator
# stands for each is a choice for the people whose data these are; this is
# one reading of it
h <- indiex("health")
belong <- paste(
  "Sense of belonging to local community,",
  "somewhat strong or very strong"
)
dom <- c(
  "Perceived health, very good or excellent" = "physical",
  "Perceived mental health, very good or excellent" = "mental",
  "Life satisfaction, satisfied or very satisfied" = "emotional",
  stats::setNames("community", belong)
)
d <- h[
  h$period == "2019/2022" &
    h$age == "Total, 18 years and over" &
    h$indicator %in% names(dom),
]
```

```
d$domain <- dom[d$indicator]
hb <- holbal(
  d,
  "group",
  "domain",
  "estimate",
  "lower",
  "upper",
  qual = "quality"
)
hb
plot(hb)
# a community that counts belonging twice
holbal(
  d,
  "group",
  "domain",
  "estimate",
  "lower",
  "upper",
  qual = "quality",
  w = c(physical = 1, mental = 1, emotional = 1, community = 2)
)

# your own table: made-up percentages of wellness in four domains a Nation
# set for itself, for two communities
own <- data.frame(
  community = rep(c("Lakeside", "Riverbend"), each = 4),
  domain = rep(c("body", "mind", "spirit", "family"), 2),
  pct = c(62, 58, 70, 75, 55, 61, 66, 80),
  lo = c(57, 53, 65, 70, 49, 55, 60, 75),
  hi = c(67, 63, 75, 80, 61, 67, 72, 85)
)
holbal(own, "community", "domain", "pct", "lo", "hi", reps = 2000)
```

`hotpers` *Hot spots and how long they last*

Description

Counts incidents (break-ins, shootings, calls for service) in square grid cells for each period, measures how concentrated they are, picks out the hot cells of each period and says how much the hot set of one period overlaps the next. A hot spot that persists is a place problem; one that moves from period to period is not, and the two call for different responses.

Usage

```
hotpers(lon, lat, prd, cell = 0.25, top = 0.05, xy = c("lonlat", "planar"))
```

Arguments

lon, lat	Incident coordinates: longitude and latitude in degrees, or planar coordinates with coords = "planar".
prd	The period of each incident (a year, a month, a date, a label such as "Q1"); periods are taken in sorted order.
cell	Side of a grid cell: kilometres for longitude and latitude, map units otherwise; one positive number, default 0.25.
top	Share of the ever-active cells counted as hot in each period, in (0, 1]; default 0.05 (the busiest 5%).
xy	"lonlat" (the default, degrees) or "planar" (projected coordinates, such as metres in a UTM zone).

Details

Counts incidents in square cells of side cell (kilometres for longitude and latitude, map units otherwise) for each period and reports, per period: the incidents, the cells with any, the share of the ever-active cells that holds half the incidents (the law of crime concentration: a small share of places holds half the incidents), the hot cells (the top share of the ever-active cells with the most incidents, with the cells tied at the cut included unless the tie would more than double the hot set) and their share of the incidents, and the Jaccard overlap of the hot set with the previous period's (shared hot cells over cells hot in either). Per cell: the incidents in all, and in how many periods it was hot.

The ever-active cells (a cell with at least one incident in some period) are the base for the shares, so the empty land and water inside a city's bounding box do not dilute the concentration. The number of hot cells per period is $k = \max(1, \lceil top \times n_{cells} \rceil)$.

Longitude and latitude are projected onto a flat map through the median point (equirectangular, Earth radius 6371.0088 km), which is accurate for a city or region; a warning is given when the incidents span more than 2000 km. The grid starts at the smallest projected coordinate. Points at (0, 0), the raw exports' "no location", are dropped with a message, as are incidents with a non-finite coordinate or a missing period; longitudes across the antimeridian are taken on one side of it. When a tie empties a period's hot set, its overlap with the neighbouring periods is NA, not 0. The plot method draws the concentration, the overlap and a map of the cells.

Errors: lon, lat and prd of different lengths; cell not one positive number; top not in (0, 1]; coordinates outside the degree ranges with coords = "lonlat"; fewer than two usable incidents.

Value

An object of class rmbl_hotspot, a list:

periods data frame, one row per period: period (character), incidents, actcells, cellsforhalf (fewest cells holding half the incidents), shrforhalf (that number over the ever-active cells), hotcells, tiedin (the cells made hot only by a tie with the k-th busiest), tiedout (the tied cells left out when the tie would have made more than twice k cells hot, which printing says), hotshr (share of the incidents in hot cells) and jacpre (NA for the first period). Shares are proportions in $[0, 1]$.

cells data frame, one row per ever-active cell: x, y (its centre in the input's units), incidents and perhot.

`cell`, `top`, `coords` the settings.

`ncells`, `hotk` the number of ever-active cells and k.

Examples

```
b <- read.csv(system.file("extdata", "tps_break_enter_d14_2024.csv",
                          package = "rmoriebricklayer"))
b <- b[!is.na(b$lon), ]
q <- quarters(as.Date(b$occ_date))
h <- hotpers(b$lon, b$lat, q, cell = 0.3)
h$periods
plot(h)

# the cells hot in every quarter
subset(h$cells, perhot == 4)

# larger cells and a wider hot set
h2 <- hotpers(b$lon, b$lat, q, cell = 0.5, top = 0.1)
h2$periods[, c("period", "hotcells", "hotshr", "jacpre")]

# planar coordinates, here in kilometres
set.seed(3)
hotpers(stats::runif(200, 0, 10), stats::runif(200, 0, 10),
                    rep(2023:2024, 100), cell = 2, top = 0.2, xy = "planar")

# a share of zero hot cells is refused
try(hotpers(b$lon, b$lat, q, top = 0))
```

`hqccmpkey` *The compressed HQC decapsulation key*

Description

Shrinks an HQC key pair for storage by replacing its secret half with the seed it was generated from (32 bytes for v5, instead of several kilobytes). The full key is derived again from the seed whenever `hqcdecap()` needs it, so a compressed key works everywhere a full one does in this package.

Usage

```
hqccmpkey(key)
```

Arguments

`key`	A key from `hqckg()`.

Details

The HQC specification of 2025-08-22 allows the decapsulation key to be stored as its 32-byte `seed_KEM` alone (the compressed format `dk_KEM = (seed_KEM)`), from which the whole key pair is derived again. `hqccmpkey()` keeps only that seed; `hqcdecap()` takes such a key and expands it, and refuses one whose seed does not derive the key's public half.

The round-4 revision (64-byte shared secret) defines no compressed format, but its key pair is derived from the randomness its key generation draws, `skseed || sigma || pk_seed` (96, 104 or 112 bytes: the `seed` of `hqckg()` and `hqcsz()`), all of which its secret key `skseed || sigma || pk_seed || s` carries. For a round-4 key the compressed form is that seed: this package's convention, not a format of the submission, so software built on round 4 (liboqs, PQClean) expects the full secret key.

A key that is already compressed is returned unchanged. Errors: `key` not from `hqckg()`; a secret of neither the full nor the seed length. The public half is kept, and anyone with the seed can rebuild the whole secret key, so the compressed key must be protected exactly like the full one.

Value

The key (class `bricklayer_hqc_key`) with `secret` replaced by the hex of its seed: 32 bytes for v5, 96, 104 or 112 for round 4. The other elements are unchanged.

References

Gaborit, P. et al. (2025). Hamming Quasi-Cyclic (HQC), specification of 2025-08-22, section 3 (HQC-KEM key pair formats). pqc-hqc.org

See Also

`hqckg()`, `hqcdecap()`

Examples

```
key <- hqckg(1)
small <- hqccmpkey(key)
nchar(small$secret) / 2
sent <- hqcencap(hqcpubkey(key))
identical(hqcdecap(small, sent$ciphertext), sent$shared)

# compressing twice changes nothing
identical(hqccmpkey(small), small)

# a seed paired with someone else's public key is refused on use
spliced <- small
spliced$public <- hqckg(1)$public
try(hqcdecap(spliced, sent$ciphertext))
```

hqcdecap *Recover a shared secret from an HQC ciphertext*

Description

The recipient's half of an HQC key exchange. Given the key pair from `hqckg()` (with its secret half) and a ciphertext from `hqcencap()`, it returns the shared secret the ciphertext carries, as hex (32 bytes; 64 for a round-4 key). The sender and recipient then hold the same secret without it ever having been sent.

Usage

```
hqcdecap(key, ct)
```

Arguments

`key`	A key from `hqckg()`, with its secret half (full or compressed).
`ct`	Ciphertext from `hqcencap()`, as a hex string or a raw vector.

Details

A ciphertext that was not produced by a correct encapsulation under this key does not raise an error: it yields the key `J(H(ek) || sigma || c)` (round 4: `K(sigma || u || v)`), derived from a value only the decapsulation key holds (the Fujisaki-Okamoto transform's implicit rejection), so whoever sent it learns nothing from the outcome. A mismatch between the two sides' secrets is the signal that something was wrong. A decapsulation key whose stored seeds no longer derive from each other (corrupted or spliced; for round 4, whose public half is not the one its secret seed makes) IS refused.

A compressed key from `hqccmpkey()` is accepted: the full key pair is derived again from the seed, and the call errors if that does not give the key's own public half. The level and version are taken from the key, so the ciphertext must come from a key of the same level and version.

Errors: `key` not an HQC key; a public key (no secret half); a secret of the wrong length; a ciphertext of the wrong length or not valid hex.

Value

The shared secret as a lowercase hex string: 64 characters (32 bytes), or 128 for a round-4 key.

See Also

`hqcencap()`, `hqccmpkey()`.

Examples

```
key <- hqckg(1)
sent <- hqcencap(hqcpubkey(key))
identical(hqcdecap(key, sent$ciphertext), sent$shared)

# a tampered ciphertext returns a secret, and it is the wrong one
bad <- sent$ciphertext
substring(bad, 3L, 4L) <- if (substring(bad, 3L, 4L) == "ff") "fe" else "ff"
other <- hqcdecap(key, bad)
nchar(other) == 64L
identical(other, sent$shared)

# the public key alone cannot decapsulate, and a truncated ciphertext
# is refused
try(hqcdecap(hqcpubkey(key), sent$ciphertext))
try(hqcdecap(key, substring(sent$ciphertext, 1L, 100L)))
```

---

hqcencap *Encapsulate a shared secret under an HQC key*

Description

The sender's half of an HQC key exchange. Given the recipient's public key, it draws fresh randomness and produces a ciphertext together with the shared secret that the ciphertext carries. The sender keeps the shared secret (for example as the key of a symmetric cipher) and sends the ciphertext; the recipient recovers the same secret with `hqcdecap()`. Only the public key is needed, and a full key from `hqckg()` is accepted too (its public half is used).

Usage

```
hqcencap(key, m = NULL, salt = NULL)
```

Arguments

`key`	A key or public key from `hqckg()` / `hqcpubkey()`.
`m`, `salt`	Optional raw vectors of encapsulation randomness, default `NULL` (drawn fresh): `m` of `hqcsz()["message"]` bytes (16, 24 or 32) and `salt` of 16 bytes. Supplying them makes the operation reproducible, which is what the known-answer tests need. Reusing them for the same key repeats the shared secret, so supply them only deliberately.

Details

The level and version are taken from the key. The randomness is a message `m` and a 16-byte `salt`. The length of `m` in bytes is `hqcsz(level)[["message"]]`. They are both drawn with `rndbytes()` from the operating system's generator unless supplied. With the same key, `m` and `salt`, the ciphertext

and secret are the same; that is what the known-answer tests need, and it is why fixed values must never be used for real traffic.

Errors: `key` not from `hqckg()` or `hqcpubkey()`; a public key of the wrong length for its level; `m` or `salt` of the wrong type or length.

Value

A list of class `bricklayer_hqc_capsule` with `ciphertext` and `shared` (character, lowercase hex; see `hqcsz()` for the byte lengths: the shared secret is 32 bytes, 64 for a round-4 key), `level` (integer) and `version` (character). The print method withholds the shared secret.

See Also

`hqcdecap()`, `hqckg()`.

Examples

```
key <- hqckg(1)
a <- hqcencap(key)
nchar(a$ciphertext) / 2 == hqcsz(1)[["ciphertext"]]

# two encapsulations to one key give different secrets
b <- hqcencap(key)
identical(a$shared, b$shared)

# both decapsulate correctly
identical(hqcdecap(key, a$ciphertext), a$shared)
identical(hqcdecap(key, b$ciphertext), b$shared)

# fixed randomness repeats the result (for tests only)
r1 <- hqcencap(key, m = as.raw(1:16), salt = as.raw(1:16))
r2 <- hqcencap(key, m = as.raw(1:16), salt = as.raw(1:16))
identical(r1$ciphertext, r2$ciphertext)

# randomness of the wrong length is refused
try(hqcencap(key, m = as.raw(1:3)))
```

---

hqckg *Generate an HQC key pair*

Description

Generates a key for HQC (Hamming Quasi-Cyclic), the code-based key encapsulation mechanism NIST selected for standardisation in March 2025 (NIST IR 8545) as a second KEM next to ML-KEM. It is implemented in this package, with no system dependency. A key encapsulation mechanism (KEM) lets two parties agree on a secret key: the recipient publishes the public half of this key pair, a sender uses it with `hqcencap()` to make a ciphertext and a shared secret, and only the holder of the secret half can recover that secret from the ciphertext with `hqcdecap()`. The shared secret

is then used as the key of a symmetric cipher. hqcpubkey() strips the secret half so the public key can be handed out.

Usage

```
hqckg(lvl = 3L, seed = NULL, ver = c("v5", "round4"))

hqcpubkey(key)
```

Arguments

lvl	Security level, as a NIST category: 1, 3 (the default) or 5 (HQC-1, HQC-3, HQC-5; 128, 192 and 256 are accepted for the same three). Level 3 matches the category of ML-KEM-768, kemkg()'s default.
seed	Optional raw vector. For v5 it is the 32 bytes of seed_KEM. For round 4 it is the hqcsz()["seed"] bytes its key generation draws (skseed, sigma, pk_seed: 96, 104 or 112). Supplying it makes the key reproducible, which is what the known-answer tests need; the default draws from the operating system's CSPRNG.
ver	"v5" (the default: the specification of 2025-08-22, a 32-byte shared secret) or "round4" (the submission of 2023-04-30, a 64-byte shared secret). Keys remember it; encapsulation and decapsulation follow the key.
key	A key from hqckg() (hqcpubkey() only).

Details

The implementation follows the HQC specification of 2025-08-22 and is byte for byte the scheme of the authors' reference implementation v5.0.0: all 300 official known-answer vectors are reproduced (the package's tests check them). It runs in constant time with respect to secrets – no secret reaches a branch, a memory index or a variable shift, checked under valgrind memcheck in CI (inst/ctcheck: the hqc1, hqc3, hqc5 and hqc1-round4 cases) – wipes its secret intermediates (the same harness scans the dead stack for them), and gives the same bytes on little- and big-endian machines. On x86-64 processors with PCLMULQDQ and on ARMv8 with the crypto extension the polynomial products use the carry-less multiply instruction; elsewhere a portable constant-time product.

NIST will publish HQC-KEM as FIPS 207 but has not yet released it, or a draft of it (checked 2026-10-05). The specification of 2025-08-22 followed here already makes the changes NIST listed for FIPS 207 (the salted FO transform, SHA3-512 seed expansion, the 32-byte shared secret, the new sampler, the seed in the decapsulation key, the seed-only compressed key that hqccmpkey() writes); the published standard may still differ in detail. Keep keys and ciphertexts tagged with the scheme, version and level they belong to.

version = "round4" gives the earlier revision instead: the fourth-round submission of 2023-04-30 (HQC-128/192/256), the HQC of liboqs up to 0.12 and of PQClean, with a 64-byte shared secret. It uses the same codes and parameters under a different key encapsulation (40-byte seeds, SHAKE256 with domain bytes in place of SHA3, another sampler), so its keys and ciphertexts are not interchangeable with v5's. It reproduces that revision's 300 official known-answer vectors (the package's tests check them); use it to exchange keys with software built on it, and v5 otherwise.

Errors: a lvl other than 1, 3, 5, 128, 192 or 256 (a typo is never mapped to a default, because the level is not recorded in the ciphertext); a seed that is not a raw vector of exactly hqcsz(level,

version)[["seed"]] bytes; a ver other than "v5" or "round4". hqcpubkey() errors on anything that is not a bricklayer_hqc_key. Key generation takes a few tens of milliseconds even at level 5.

Value

hqckg() returns a list of class bricklayer_hqc_key with public and secret (character, lowercase hex of the encapsulation and decapsulation keys; see hqcsz() for their byte lengths), level (integer 1, 3 or 5) and version ("v5" or "round4"). Its print method shows the level and the start of the public key and withholds the secret. hqcpubkey() returns a list of class bricklayer_hqc_public_key with public, level and version only. The v5 secret key is the specification's ek || seed_dk || sigma || seed_KEM; the round-4 one is skseed || sigma || ek.

Security

The implementation is hand-written. Its standardised schemes are checked byte for byte against OpenSSL 3.5 and NIST known-answer vectors in the test suite. Side channels are checked, not assumed: inst/ctcheck runs every operation that touches a secret under valgrind memcheck with the secret marked undefined (the ctgrind method), so a branch or a memory address that depends on it is a reported error, and afterwards scans the dead stack for copies of the secret. Both checks run in CI on every change, with GCC and with Clang. Timing is also measured on x86-64 and arm64 hardware (inst/dudect) and power leakage is assessed in simulation under the value and the transition models (inst/tvla) for the masked ML-KEM and ML-DSA kernels; this scheme is not masked. These are checks of this code on those compilers; the README's security section lists what each check does and does not cover.

References

Gaborit, P., Aguilar-Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.-C., Persichetti, E., Zemor, G., et al. (2025). Hamming Quasi-Cyclic (HQC), specification of 2025-08-22; reference implementation v5.0.0. pqc-hqc.org

Alagic, G. et al. (2025). Status Report on the Fourth Round of the NIST Post-Quantum Cryptography Standardization Process. NIST IR 8545. DOI 10.6028/NIST.IR.8545

See Also

hqcencap(), hqcdecap(), hqcsz(), kemkg() for ML-KEM.

Examples

```
key <- hqckg(1)
pub <- hqcpubkey(key)

# the sender holds only the public key
sent <- hqcencap(pub)
# the recipient recovers the same secret from the ciphertext
got <- hqcdecap(key, sent$ciphertext)
identical(sent$shared, got)

# a corrupted ciphertext yields a DIFFERENT secret, not an error
bad <- sent$ciphertext
```

```
substring(bad, 1L, 2L) <- if (substring(bad, 1L, 2L) == "00") "01" else "00"
identical(hqcdecap(key, bad), got)

# a reproducible key from a fixed seed
identical(hqckg(1, seed = as.raw(1:32))$public,
          hqckg(1, seed = as.raw(1:32))$public)

# the round-4 revision: the same exchange, a 64-byte shared secret
old <- hqckg(1, ver = "round4")
cap <- hqcencap(hqcpubkey(old))
nchar(cap$shared) / 2
identical(hqcdecap(old, cap$ciphertext), cap$shared)
```

`hqcsz` *Byte lengths of an HQC parameter set*

Description

Reports the byte lengths that the HQC specification fixes for a security level: the encapsulation (public) key, the decapsulation (secret) key, the ciphertext, the key-generation seed, the encapsulation randomness and the shared secret. Use it to validate stored keys and ciphertexts, or to size storage, without hard-coding numbers.

Usage

```
hqcsz(lvl, ver = c("v5", "round4"))
```

Arguments

`lvl`	Security level: 1, 3 or 5 (or 128, 192, 256 for the same three). There is no default.
`ver`	`"v5"` (the default) or `"round4"`, as in `hqckg()`.

Details

The values come from the compiled implementation, so they are the sizes it actually uses. For `version = "v5"` (specification of 2025-08-22):

	HQC-1	HQC-3	HQC-5
encapsulation key	2241	4514	7237
decapsulation key	2321	4602	7333
ciphertext	4433	8978	14421
seed	32	32	32
message	16	24	32
salt	16	16	16
shared secret	32	32	32

For `version = "round4"` the codes are the same but the key encoding differs: the seed is 96, 104 or 112 bytes and the shared secret 64 bytes. A level other than 1, 3, 5 (or 128, 192, 256) is an error.

Value

A named integer vector of byte lengths: `public` (the encapsulation key), `secret` (the decapsulation key), `ciphertext`, `seed` (key generation), `message` and `salt` (encapsulation randomness) and `shared` (the shared secret), named as `hqckg()` and `hqcencap()` name their results.

See Also

`hqckg()`, `kemsz()` for ML-KEM.

Examples

```
hqcsz(1)

# code-based keys and ciphertexts are larger than ML-KEM's at the same
# category; the shared secret is 32 bytes at every level (64 in round 4)
rbind(HQC = hqcsz(3)[c("public", "ciphertext")],
      MLKEM = kemsz(768)[c("public", "ciphertext")])
rbind(v5 = hqcsz(1), round4 = hqcsz(1, "round4"))

# check a stored public key's length
key <- hqckg(1)
nchar(key$public) / 2 == hqcsz(1)[["public"]]

# an unknown level is refused rather than mapped to a default
try(hqcsz(2))
```

---

hurwzeta *Hurwitz zeta function*

Description

Computes the Hurwitz zeta function $\zeta(s, q) = \sum_{k \geq 0} (q + k)^{-s}$ for a vector of exponents `s` and one lower limit `q`. It is the normalising constant of the discrete power law truncated below at `q`, which is why it is here (see `hilltlidx()`); `zeta(s, 1)` is the Riemann zeta.

Usage

```
hurwzeta(s, q = 1)
```

Arguments

`s`	Numeric vector of exponents. The series converges for `s > 1`; below 1 the value is the analytic continuation (so `hurwzeta(0.5,0.5)` is -0.6049), and `s = 1` is a pole (`NA`).
`q`	Numeric scalar, the lower limit; default `1` (the Riemann zeta). Any real number but zero or a negative integer (poles, `NA`). For a negative non-integer `q` the value is real only for a whole `s`; a fractional `s` gives `NA`.

Details

For $s > 1$ and $q > 0$ the series is evaluated in compiled code by Euler-Maclaurin summation: direct terms, the integral of the tail, and Bernoulli-number corrections. For $s < 1$ the value is the analytic continuation: exact Bernoulli polynomials for whole $s > -5$ ($\zeta(-n, q) = -B_{n+1}(q)/(n+1)$), Riemann's functional equation at $q = 1$, Hurwitz's Fourier series for $s \leq -5$, Hermite's integral (by `stats::integrate()`) for $-5 < s < 1$, and the asymptotic series for large q, with the shift $\zeta(s, q) = q^{-s} + \zeta(s, q+1)$ to move q into range. A negative non-integer `q` uses the reflection of its negative terms.

Values that do not exist are `NA`, not errors: `s = 1` (a pole), `q` zero or a negative integer (poles), a fractional `s` with a negative `q` (a complex value), and a non-finite `s`. Errors when `s` or `q` is logical or not numeric, and when `q` is not a single finite number.

Value

A numeric vector the length of `s`.

References

The Euler-Maclaurin expansion used here – direct terms to `q + N`, then the integral tail, then the Bernoulli-number corrections – is the standard evaluation; see Abramowitz, M. and Stegun, I. A. *Handbook of Mathematical Functions*, Sec. 23.2. (Not in the local corpus; cited from the published reference. The implementation is checked against `pi^2/6`, `pi^4/90`, Apery's constant and the shift identity, which is stronger evidence than the citation.)

Examples

```
# The Riemann zeta at even integers has a closed form.
c(hurwzeta(2), pi^2 / 6)
c(hurwzeta(4), pi^4 / 90)

# Apery's constant.
hurwzeta(3)

# Shifting the lower limit removes exactly the leading term.
hurwzeta(2.5, 3) - hurwzeta(2.5, 4)
3^-2.5

# Outside the domain of convergence there is no value to return.
hurwzeta(1)

# several exponents at once, including the analytic continuation
hurwzeta(c(2, 0.5, -1))  # -1/12 at s = -1

# one lower limit per call
try(hurwzeta(2, q = c(1, 2)))
```

idcrd *Make, read and verify an identity card for your signing key*

Description

An identity card ties your public signing key to who you are: your name and, if you give them, your ORCID iD, e-mail address, web page and affiliation, all signed with the key itself. Publish the card (on your web page, in a repository, beside a paper) and give its fingerprint somewhere people already know you by (the paper, your ORCID record, a talk). Anyone can then check with `idcrdvrf()` that the card is intact and in date, that its fingerprint is the one you gave, and that an attestation or a bundle was signed by the key it carries.

`idcrd()` makes a card from a key of `fipskg()` (ML-DSA or SLH-DSA) or `pqckg()` (XMSS; making the card uses one of the key's signatures). `idcrdrd()` reads a card written with `file`. `idcrdvrf()` checks one.

Usage

```
idcrd(
  key,
  name,
  orcid = NULL,
  email = NULL,
  url = NULL,
  affil = NULL,
  days = 730,
  file = NULL
)

idcrdrd(path)

idcrdvrf(card, fpr = NULL, at = Sys.time(), atst = NULL)
```

Arguments

`key`	Your private signing key, from `fipskg()` or `pqckg()`.
`name`	Your name as you want it shown, one string.
`orcid`, `email`, `url`, `affil`	Optional, one string each: your ORCID iD (`"0000-0002-1825-0097"`, or its `https://orcid.org/` address), e-mail address, web page (`https://`) and affiliation.
`days`	How many days the card is valid for, from now. Default 730.
`file`	Optional path to write the card to, as JSON.
`path`	The path of a card file written by `idcrd()`.
`card`	A card, from `idcrd()` or `idcrdrd()`, or the path of one.

fpr	Optional: the fingerprint you were given for this person, with or without its spaces. Without it the card is checked only against itself.
at	The time to check the card's dates at: a POSIXct or anything as.POSIXct() reads, taken as UTC. Defaults to now.
atst	Optional: an attestation of cpsatst(), or a bundle of cpsbndl() or cpsbndlrd(), to check was signed by the card's key. Check the attestation itself with cpschkatst() or cpsbndlvrf().

Details

A card is self-signed. Its signature proves that the card was made by whoever holds the private key and that nothing in it has changed since; it cannot prove that the name on it is true. That is what the fingerprint is for: a card whose fingerprint matches the one you published where people know you is the key you meant. The card's signature is made under its own context string (or, for XMSS, over a payload naming the card type), so it can never be passed off as an attestation, nor an attestation as a card.

The fingerprint is the SHA-256 digest of the scheme and the public key, written as 16 groups of four hexadecimal digits; compare all of it. An ORCID iD is checked against its check digit (ISO 7064 MOD 11-2), an e-mail address for its shape and a web page for an https:// address.

idcrdvrf() records every step as a row of checks: card_complete, card_type, signature, in_date (at at), and orcid when the card has one, fingerprint when fpr is given, signed_by_card when atst is given. ok is TRUE only when every row passed; a failed check is a row, not an error.

Errors: key not a private key of fipskg() or pqckg(); name not one non-empty string; an optional field that is not one string, an ORCID iD with a wrong check digit, an e-mail address or a web page of the wrong shape; days not a positive whole number; path not a card file.

Value

idcrd() and idcrdrd(): a list of class rmbl_id_card with the card's fields (type, version, name, orcid, email, url, affil, scheme, public, context, prehash, issued_utc, expires_utc, signature) and its fingerprint; idcrd() returns it invisibly when it writes file. idcrdvrf(): a list of class rmbl_id_check with ok, checks (a data frame with check, ok and detail), fingerprint and card. print() shows a summary.

See Also

fipskg(), pqckg(), cpsatst(), cpschkatst(), cpsbndlvrf().

Examples

```
# once: make a key and a card, publish the card, give out the fingerprint
key <- fipskg("ML-DSA-44")
f <- tempfile(fileext = ".json")
idcrd(
  key,
  "Ada Lovelace",
  orcid = "0000-0002-1825-0097",
  url = "https://example.org/ada",
  file = f
```

```
)
card <- idcrdrd(f)
card
card$fingerprint

# anyone, later: is this the card I was told about, and did it sign this?
man <- mkmnf(list(project = "demo"), env = FALSE)
a <- cpsatst(man, key)
res <- idcrdvrf(f, fpr = card$fingerprint, atst = a)
res$ok
res$checks

# a card edited after signing fails
bad <- card
bad$name <- "Someone Else"
idcrdvrf(bad)$ok

# an attestation made with another key is not the card's
other <- cpsatst(man, fipskg("ML-DSA-44"))
idcrdvrf(card, atst = other)$ok

# an ORCID iD with a wrong check digit is refused
try(idcrd(key, "Ada Lovelace", orcid = "0000-0002-1825-0098"))
```

---

indgap *Compare published group estimates with a reference group*

Description

For published estimates with their confidence limits (percentages, rates, years of life expectancy: any scale, with the limits on the same scale), the ratio and the difference of each group to a reference group, with confidence intervals worked from the published ones: First Nations, Metis and Inuit against non-Indigenous people, Aboriginal and Torres Strait Islander people against non-Indigenous Australians, Maori against non-Maori, or any groups with published intervals.

Usage

```
indgap(
  data,
  grp,
  est,
  lower,
  upper,
  ref = "Non-Indigenous",
  qual = NULL,
  conflvl = 0.95,
  pooled = NULL
)
```

Arguments

`data`	A data frame, one row per group (filtered to one period, age group, sex and place: a group appearing twice is refused).
`grp`, `est`, `lower`, `upper`	Column names: the group label, the estimate, and its published confidence limits, on one scale.
`ref`	The label of the reference group (default `"Non-Indigenous"`).
`qual`	Optional column of quality flags in the package's vocabulary (`"E"`, `"F"`, `"x"`, `".."`, `"..."`; see `qltyrcd` for other agencies' flags).
`conflvl`	The confidence level of the published intervals and of the ones returned (default 0.95).
`pooled`	Optional labels of the rows that pool other groups (for example `"Aboriginal and Torres Strait Islander"` beside its parts, or a "Total Indigenous" row); by default the labels described in Details. Each label must be a group of `data`: one that matches no row (a typo) is refused rather than ignored.

Details

Each published interval gives a standard error, $s = (u - l)/(2z)$, on the scale of the estimate for the difference and on the log scale, $\sigma = (\log u - \log l)/(2z)$, for the ratio, with $z = \Phi^{-1}(1 - \alpha/2)$. This takes the interval as symmetric on that scale; for asymmetric (gamma, logit, Wilson) limits it averages the two sides. With a standard error, a coefficient of variation or a margin of error instead of limits, make limits with `limfrom` first. The groups are treated as independent samples, so the standard error of a difference is $\sqrt{s_g^2 + s_r^2}$ and of a log ratio $\sqrt{\sigma_g^2 + \sigma_r^2}$. Disjoint domains of one survey are nearly independent but not exactly (they share the design's strata and clusters): the intervals are approximations, and the exact ones need the survey's replicate weights (for Statistics Canada's surveys, in its Research Data Centres). The reference must not contain the groups: compare with non-Indigenous people, not with a total, whose covariance with each group is large and positive (the function warns when the reference looks like a total). A lower limit of 0 (a limit cut at zero) leaves the ratio's interval `NA`.

Distinctions-based: each group is compared on its own. A row that pools groups is kept but marked `poogrp = TRUE`, since a pooled figure can hide what differs between the groups it pools: name such rows in `pooled` (by default labels beginning `"Total"`, "Indigenous identity", "Aboriginal identity" or "Indigenous population" are taken as pooled). A row flagged `E` (in the group or the reference) is compared and marked; one flagged `F`, `x`, `..` or `...`, with no estimate, or whose published interval does not contain its own estimate, is not compared. Other agencies' flags: `qltyrcd`.

Value

A data frame of class `rmbl_indigenous_gap`, one row per group but the reference. The columns are `group`, `estimate`, `refest`, `ratio`, `ratiolo`, `ratioup`, `difference`, `diflo`, `difup`, `usewithcau` and `poogrp`; attributes `reference` and `conflvl`. `plot()` draws each group's difference or ratio.

References

Statistics Canada (2025). Health indicators for First Nations adults living off reserve, Metis and Inuit adults, 2015 to 2022. The Daily, 2025-02-13.

www150.statcan.gc.ca/n1/daily-quotidien/250213/dq250213f-eng.htm

Carroll SR, Garba I, Figueroa-Rodriguez OL, et al. (2020). The CARE Principles for Indigenous Data Governance. Data Science Journal 19:43. DOI 10.5334/dsj-2020-043

See Also

dstntst before pooling, limfrom, qltyrcd, agestd.

Examples

```
# Perceived mental health very good or excellent, adults, 2019/2022
# (Statistics Canada, table 13-10-0924-01; The Daily, 2025-02-13)
h <- indiex("health")
mh <- h[
  h$period == "2019/2022" &
    h$age == "Total, 18 years and over" &
    h$indicator == "Perceived mental health, very good or excellent",
]
g <- indgap(mh, "group", "estimate", "lower", "upper", qual = "quality")
g
plot(g)
plot(g, what = "ratio")
# your own table: made-up numbers in another country's layout, with relative
# standard errors instead of limits and that agency's flags
own <- data.frame(
  group = c(
    "Aboriginal",
    "Torres Strait Islander",
    "Aboriginal and Torres Strait Islander",
    "Non-Indigenous Australians"
  ),
  pct = c(31.2, 27.5, 30.8, 18.9),
  rse = c(4.1, 22.0, 3.9, 1.2),
  flag = c("", "*", "", "")
)
own <- cbind(own, limfrom(own$pct, cv = own$rse)[c("lower", "upper")])
own$q <- qltyrcd(own$flag, suppr = c("np", "**"), caution = "*")
indgap(
  own,
  "group",
  "pct",
  "lower",
  "upper",
  ref = "Non-Indigenous Australians",
  qual = "q",
  pooled = "Aboriginal and Torres Strait Islander"
)
```

indiex *First Nations, Metis and Inuit health and mental health: bundled Statistics Canada excerpts*

Description

Small excerpts of Statistics Canada tables, bundled so that the examples and vignettes run on published values without a network connection or a data service key. Values are as published, with their 95% limits and quality flags; nothing is imputed (suppressed cells are NA).

Usage

```
indiex(
  name = c("health", "mentalhealth", "wellbeing", "suicide",
           "censuspop", "censushh", "lifeexp", "infantmort")
)
```

Arguments

name	Which excerpt.

Details

"health" Table 13-10-0924-01, Health indicators for First Nations people living off reserve, Metis and Inuit, by age group and sex, four-year period estimates (Canadian Community Health Survey), 2015/2018 and 2019/2022; Canada, both sexes, the 27 indicators by age group, with non-Indigenous people for comparison: the table behind The Daily of 2025-02-13. Columns period, group, age, indicator, estimate, lower, upper, quality.

"mentalhealth" Table 41-10-0067-01, Self-perceived mental health of First Nations people living off reserve, Metis and Inuit by age group and gender, 2022 (Indigenous Peoples Survey); 15 years and older, every region. Columns region, group, rating, estimate, lower, upper, quality.

"wellbeing" Table 41-10-0082-01, Selected characteristics of well-being and impacts of rising prices by First Nations people living off reserve, Metis and Inuit, and gender, 2024; total of genders. Columns group, indicator, estimate, lower, upper, quality.

"suiide" Table 41-10-0021-01, Self-perceived mental health and suicidal thoughts by Aboriginal identity, age group and sex, 2017 (Aboriginal Peoples Survey); Canada, both sexes, by age. Columns group, age, measure, estimate, lower, upper, quality.

"censuspop" The Indigenous Population Profile, 2021 Census of Population (98-510-X2021001), Canada, the provinces and the territories, total of genders, on and off reserve: the population by age group, children and foster children, knowledge of Indigenous languages, highest certificate (25 and over), labour force status, income and low income, as published (read it with cnsprof).

"lifeexp" Table 17-10-0160-01, Life expectancy at birth of Indigenous populations in Canada, 2006/2011 to 2016/2021, both sexes, Canada: First Nations (Registered or Treaty Indians, on and off reserve, and without registered status), Metis, Inuit and non-Indigenous people. Columns period, group, measure, estimate, lower, upper, quality.

`"infmort"` Table 17-10-0161-01, Infant mortality rates of Indigenous populations in Canada (per 1,000 live births), 1994/1996 to 2014/2016, the same groups and columns.

`"censushh"` The same profile's households and dwellings (98-510-X2021016), Indigenous and non-Indigenous households: persons per room, housing suitability, dwelling condition and core housing need.

The First Nations figures of the survey excerpts are for First Nations people living off reserve: the surveys do not cover reserves, whose data are governed by First Nations (see the First Nations Information Governance Centre). Groups are as the tables name them, with "Inuk (Inuit)" written `"Inuit"`.

Value

A data frame.

Source

Statistics Canada tables, each with its DOI:

- 13-10-0924-01, DOI 10.25318/1310092401-eng;
- 41-10-0067-01, DOI 10.25318/4110006701-eng;
- 41-10-0082-01, DOI 10.25318/4110008201-eng;
- 41-10-0021-01, DOI 10.25318/4110002101-eng;
- 17-10-0160-01, DOI 10.25318/1710016001-eng;
- 17-10-0161-01, DOI 10.25318/1710016101-eng.

Indigenous Population Profile, 2021 Census of Population, catalogue 98-510-X2021001 and 98-510-X2021016. Reproduced and distributed on an "as is" basis with the permission of Statistics Canada, under the Statistics Canada Open Licence:

statcan.gc.ca/en/terms-conditions/open-licence

Built by the scripts `indigenous_mental_health.R` and `indigenous_census.R` in the `data-raw` folder of the source package.

See Also

`indtabs` for the full tables at the data service.

Examples

```
h <- indiex("health")
mh <- h[h$period == "2019/2022" & h$age == "Total, 18 years and over" &
          h$indicator == "Perceived mental health, very good or excellent", ]
plot(indgap(mh, "group", "estimate", "lower", "upper", qual = "quality"))
```

indtabs *The Indigenous health and wellbeing tables the data service holds*

Description

Lists the First Nations, Metis and Inuit tables mirrored at the data service: Statistics Canada's Indigenous-titled tables (`statcan_indigenous`) and the 2021 Census Indigenous Population Profile (`census2021_ipp`), with their descriptions, row counts and licences, searchable.

Usage

```
indtabs(pat = NULL, refresh = FALSE)
```

Arguments

`pat`	Optional regular expression matched against the descriptions (case ignored).
`refresh`	Fetch the manifest again (default `FALSE`).

Details

Needs a data service key: sign in once with `dlogin`. Only openly licensed aggregate tables are mirrored. First Nations Information Governance Centre data (the First Nations Regional Health Survey and its companions) are governed by the OCAP(R) principles and are not; ask the Centre (fnigc.ca; the site answers scripts with 403, archived copy web.archive.org/web/20261001141317/fnigc.ca).

Value

A data frame: `key`, `rows`, `description`, `license`, `source`.

Examples

```
indtabs("mental health")
```

infsch *Infer a pinnable schema from a data frame*

Description

Derives the schema `valsch()` consumes from data you already trust, so a capsule can be pinned without writing one by hand. Infer it once from the extract the analysis was run on, read it, edit it, commit it, and validate every later download against it.

Usage

```
infsch(data, slack = 0.1, maxlvl = 50L)
```

Arguments

`data`	A data frame to learn from.
`slack`	Headroom, one non-negative number (default 0.1). It is a fraction of the row count and of each numeric range, and an absolute amount added to each missingness rate (0.1 allows ten more percentage points of `NA`). `0` pins exactly to what was observed, which will reject almost any re-release.
`maxlvl`	Maximum distinct values for a categorical column to have its value set recorded, one positive whole number (default 50). Above this the column is treated as free text and no value set is pinned.

Details

What it records: the column names and their types, row-count bounds with `slack` either side, the observed value set for every low-cardinality categorical column, the observed range of every numeric column widened by `slack`, and the observed missingness rate per column with headroom.

The rules, with n rows:

- types: the first class of each column (`"integer"`, `"numeric"`, `"character"`, `"factor"`, `"Date"`, ...);
- rows: `min_data_rows` $= \lfloor n(1 - slack) \rfloor$ (at least 0) and `max_data_rows` $= \lceil n(1 + slack) \rceil$;
- numeric ranges: $[lo - pad, hi + pad]$ with $pad = slack \times (hi - lo)$, or $slack \times \max(|lo|, 1)$ for a constant column; a column with no non-missing value gets no range;
- value sets: for character, factor and logical columns with at most `maxlvl` distinct non-missing values, those values as sorted text;
- missingness: the observed share of `NA` plus `slack` (added, not multiplied), capped at 1.

Other column types (dates, lists) get a type and a missingness ceiling only.

This is a starting point, not an oracle:
An inferred schema describes ONE extract. It cannot know that a category which happens not to occur is nonetheless legal, or that a range is a physical bound rather than an accident of this sample. Read what it produces and edit it before committing – the value is in not starting from a blank file, not in trusting the output blindly.
Errors: `data` not a data frame or without columns; `slack` not one non-negative number; `maxlvl` not one positive whole number.

Value

A list of class `bricklayer_schema` with these elements:

- `expected_columns` (character);
- `expected_types` (named character);
- `structural_invariants` (list of `min_data_rows` and `max_data_rows`, integers);
- `expected_value_sets` (named list of character vectors);
- `numeric_ranges` (named list of `c(min =, max =)`);
- `max_missing_fraction` (named numeric).

Wrap it as `list(schema = <this>)` to hand to `valsch()`.

See Also

valsch(), cpsdrft() for the distributional check the schema cannot make.

Examples

```
set.seed(1)
df <- data.frame(
  id = 1:100,
  score = stats::runif(100, 0, 10),
  grade = sample(c("a", "b", "c"), 100, TRUE),
  note = paste0("free text ", 1:100),
  stringsAsFactors = FALSE
)

sch <- infsch(df)
sch

# The categorical column has its levels pinned; the free-text one does
# not, because it exceeds max_levels.
sch$expected_value_sets

# It validates the data it was learned from.
length(valsch(df, list(schema = sch)))

# And catches a column that has gone missing, or a new category.
length(valsch(df[, -3], list(schema = sch))) > 0
bad <- df; bad$grade[1] <- "z"
length(valsch(bad, list(schema = sch))) > 0

# no slack pins the exact observed range
infsch(df, slack = 0)$numeric_ranges$score

# a negative slack is refused
try(infsch(df, slack = -0.1))
```

---

inlhist *Inline histogram for a numeric vector*

Description

Draws a one-line sketch of a column's distribution, as block characters (or ASCII where the console cannot render them). The point is density of information: a mean and a standard deviation cannot tell you a column is bimodal, skewed or has a gap, and this can, in the width of a table cell. profcols() shows one for every numeric column.

Usage

```
inlhist(x, bins = 10L)
```

Arguments

x	Numeric vector drawn as an inline histogram.
bins	Number of bins, and so of characters, one positive whole number (default 10).

Details

Non-finite values (NA, NaN, Inf) are dropped. The range of the rest is cut into bins equal-width bins (as graphics::hist() with those breaks) and each bin's count is drawn as one of eight Unicode block heights, scaled to the largest bin; an empty bin is a space, and a bin with any values is at least the lowest block. When the session is not UTF-8, or the environment variable RMBL_ASCII_ONLY is set, five ASCII levels (. : - = #) are used instead. A column with no spread gives one full block in the middle; no finite values give spaces.

Errors: x that is not numeric; bins that is not one positive whole number.

Value

A length-1 character vector of bins characters.

See Also

profcols(), which includes one per numeric column.

Examples

```
set.seed(1)
# A symmetric distribution peaks in the middle.
inlhist(stats::rnorm(1000))

# A skewed one leans left.
inlhist(stats::rexp(1000))

# Bimodality is visible here and in no single summary number.
inlhist(c(stats::rnorm(500, -3), stats::rnorm(500, 3)))

# A constant column has no spread to show.
inlhist(rep(5, 10))

# more bins for more detail
inlhist(stats::rexp(1000), bins = 30)

# text is not a distribution
try(inlhist(c("a", "b")))
```

`instcli` *Install the rmoriebricklayer command-line launcher*

Description

Writes small launcher scripts into a directory on your PATH so that `rmoriebricklayer login`, `rmbl ask ...` and the other verbs of `blcli` work from any shell. Run it once after installing or upgrading the package. Nothing outside `dir` is touched, and only when you call this.

Usage

```
instcli(
  dir = file.path(path.expand("~"), ".local", "bin"),
  name = c("rmoriebricklayer", "rmbl")
)
```

Arguments

`dir`	Character scalar, the target directory; default `~/.local/bin`. Created (with its parents) if absent.
`name`	Character vector of command names to install; default both `rmoriebricklayer` and its short form `rmbl` (every verb works under either).

Details

On Unix-alikes each launcher is a `/bin/sh` script (mode 0755) that sets `RMBL_PROG` to its own name and runs `Rscript -e "rmoriebricklayer::blcli()"` with the shell's arguments, exiting with the function's status (130 on Ctrl-C). The library this copy of the package is installed in is pinned inside R with `.libPaths()`, so an `R_LIBS` set in the shell or in `~/.Renviron` cannot make the command run another copy. An existing file or link of the same name in `dir` is replaced. On Windows a `NAME.cmd` batch wrapper doing the same is written instead. A message names the files written and, when `dir` is not on the `PATH`, says to add it. Re-run it after moving the package to another library.

Errors when the package installation has no launcher in `inst/bin` (a damaged installation); when `dir` cannot be created or written, or is not one string; and when `name` is not a character vector of one or more command names (empty, `NA`, an empty string or a name with a directory part is refused).

Value

The path of the first launcher written (normally `rmoriebricklayer`), invisibly; the others are written beside it.

See Also

`blcli` for the verbs.

Examples

```
# into a scratch folder, so nothing on your PATH changes
d <- file.path(tempdir(), "bin")
p <- instcli(dir = d)
list.files(d)
readLines(p)[1:4]

# one name only
instcli(dir = d, name = "rmbl")


instcli()   # ~/.local/bin/rmoriebricklayer and ~/.local/bin/rmbl
```

---

jsgz *Compressed, base64-encoded JSON*

Description

`jsgzenc()` serialises to JSON, compresses with gzip and encodes the result as base64, so it can be embedded in another JSON document, a header, or a text column. `jsgzdec()` reverses all three steps.

Usage

```
jsgzenc(x, raw = FALSE, ...)

jsgzdec(txt, raw = FALSE, ...)
```

Arguments

`x`	Object to encode.
`raw`	Logical scalar, default `FALSE`: return (or, for the decoder, insist on) gzip bytes instead of base64 text.
`...`	For `jsgzenc()`, passed to `tojs()` (for example `auto_unbox` or `digits`); for `jsgzdec()`, passed to `fromjs()` (for example `simplifyVector`).
`txt`	Base64 string (or raw vector of gzip bytes) from `jsgzenc()`.

Details

JSON is highly compressible because every record repeats the key names, so this is usually a large saving on anything record-shaped – but it is opaque. Use it for payloads that travel, and plain `tojs()` for anything a person is meant to read or a `git diff` is meant to show.

`raw = TRUE` skips the base64 step and returns the gzip bytes, which is what to use when writing to a file rather than embedding in text.

The bytes are a real gzip member (RFC 1952, with header bytes `1f 8b`), readable by `gzip -d`, `zcat` and any HTTP client, not the zlib stream that `memCompress(type = "gzip")` returns. The

modification time in the header is zero, so the same input always gives the same bytes and a digest of the result does not depend on the clock. The base64 text uses the standard alphabet with line breaks every 72 characters (jsb64enc()).

The decoder takes the input's own type: a raw vector is read as gzip bytes whatever raw says, and text as base64. As a guard against decompression bombs it refuses a member whose declared uncompressed size is over 64 MiB. Errors: text given with raw = TRUE; text that is not base64; bytes that are not a gzip member (under 18 bytes or not starting 1f 8b); a declared size over 64 MiB; a corrupt member; invalid JSON inside.

Note that the JSON keeps data, not R types: a factor comes back as text and attributes are lost. Compress jsser() output when the object must come back exactly.

Value

jsgzenc() a length-1 character vector, or a raw vector when raw = TRUE. jsgzdec() the decoded object, as fromjs() returns it.

See Also

tojs(), jsser() for an uncompressed form that keeps types and attributes.

Examples

```
x <- list(rows = data.frame(id = 1:50, value = stats::runif(50)))

enc <- jsgzenc(x)
substring(enc, 1, 40)

# Smaller than the JSON it came from, because the keys repeat.
c(json = nchar(tojs(x)), gzipb64 = nchar(enc))

# Round trips.
identical(jsgzdec(enc)$rows$id, 1:50)

# Raw gzip bytes, for writing to a file.
bytes <- jsgzenc(x, raw = TRUE)
class(bytes)
identical(jsgzdec(bytes, raw = TRUE)$rows$id, 1:50)

# the bytes are ordinary gzip: a file any gzip tool can read
f <- tempfile(fileext = ".json.gz")
writeBin(bytes, f)
substr(readLines(gzfile(f), warn = FALSE), 1, 40)

# options pass through to the JSON writer and reader
jsgzdec(jsgzenc(list(a = 1, b = "x"), auto_unbox = TRUE),
                simplifyVector = FALSE)

# text that is not a gzip member is refused
try(jsgzdec(jsb64enc("not gzip")))
```

jsserdoc *JSON serialisation of an R object, type and attributes included*

Description

Writes an R object to JSON with its type and attributes alongside the value, so the round trip returns THE SAME OBJECT rather than something that merely prints the same. The counterpart of jsonlite's `serializeJSON()` / `unserializeJSON()`, computed by this package's own codec with no jsonlite dependency.

Usage

```
jsser(x, digits = 8, pretty = FALSE)

jsunser(txt, trusted = FALSE)
```

Arguments

`x`	Object to serialise.
`digits`	Decimal digits retained for doubles (default 8, the jsonlite default, so `1/3` is written as `0.33333333`). `digits = I(17)` writes every double exactly (17 significant digits), which makes the round trip `identical()` for doubles too; `NA` means 15 significant digits, as in jsonlite, and is not exact. A complex value with only one `NA` part comes back as `NA`, as it does in jsonlite.
`pretty`	Logical scalar, default `FALSE`: indent the output.
`txt`	Character scalar: JSON produced by `jsser()`.
`trusted`	`FALSE` (default) rebuilds data types only (vectors, lists, factors, matrices, data frames, environments, symbols) and refuses functions, calls, expressions, namespaces and S4 objects: rebuilding those runs code the JSON's author chose, through `unserialize()`, `parse()`, `getNamespace()` or an `initialize` method. Set `TRUE` only for JSON your own session produced.

Details

Use this, not `tojs()`, whenever the JSON has to reconstruct the object faithfully. `tojs()` writes the DATA – which is what an API or a human wants, and which loses factor levels, matrix dimensions, classes and every other attribute. These two keep them, at the cost of JSON no other tool will understand.

Value

`jsser()` returns a length-1 character vector of class `json`; `jsunser()` returns the original object.

See Also

`tojs()` for plain data JSON, `csha256()` for fingerprinting the result.

Examples

```
# A factor survives the round trip with its levels intact.
f <- factor(c("b", "a", "b"), levels = c("a", "b", "c"))
back <- jsunser(jsser(f))
identical(back, f)

# So does a matrix, with its dimensions.
m <- matrix(1:6, nrow = 2)
identical(jsunser(jsser(m)), m)

# Plain data JSON does not keep either, which is the trade-off.
tojs(f)

# Nested lists, names and NULLs round trip too.
x <- list(a = 1:3, b = list(c = "x", d = NULL), e = TRUE)
identical(jsunser(jsser(x)), x)

# Doubles are rounded to `digits`; I(17) keeps them exact.
jsser(1/3)
exact <- jsser(1/3, digits = I(17))
identical(jsunser(exact), 1/3)

# The serialised form is JSON, so it can be pinned like any other text.
nchar(csha256(jsser(m)))

# A function is refused unless the JSON is trusted.
try(jsunser(jsser(function(x) x + 1)))
```

`justice-data` *Court and police data shipped with the package*

Description

Small excerpts of published court and police statistics, in `inst/extdata`, used by the examples, tests and the `courts-and-crime` vignette. Each is a faithful copy of published numbers (no value altered, no row invented); the full tables, every edition and location, are on the package's data hub (`dload`, collections `ocj_statistics` and `tps_open_data`).

Details

`ocj_flow_ontario.csv` Ontario Court of Justice, criminal clearance-rate statistics, province-wide, monthly from April 2024 to March 2026 (the 2026 edition): `month`, cases `received`, cases `disposed`, cases `pending` (net of bench warrants).

`ocj_time_to_disposition.csv` The same months from the disposition-rate statistics: the shares of cases disposed in under 10 months, 10-15, 15-18 and over 18 months (cases without bench warrants), and the average days to disposition.

`ocj_dispositions.csv` Province-wide criminal cases disposed by how they ended, one row per calendar year 2012-2019, 2021, 2022, 2024 and 2025; each row adds to the year's published total. The 2020 edition covers October to September and the 2026 edition April to March; the 2023 edition publishes no Toronto figures, so it has no province-wide row.

`ocj_offence_groups.csv` The same years by offence group: cases received and disposed. The groups add to each year's province-wide total.

`ocj_courts.csv` The same years by court, with `court_series`, which names Toronto's courthouses `"Toronto"` throughout: from 2023 the Court publishes Toronto as one location (the new Toronto courthouse consolidated its criminal courts that year). Each year's courts add to the province-wide total.

`tps_break_enter_d14_2024.csv` Toronto Police Service Break and Enter open data, every break and enter that occurred in 2024 in division D14, one row per occurrence (508): `event_id`, occurrence date, premises type, the published longitude and latitude (which the Service offsets to the nearest intersection), and `offence_records`, the number of offence records the Service publishes for the occurrence (five have two). The source package builds it with the script `tps_break_enter_d14_2024.R` in its `data-raw` folder.

Value

No R object: each file is a CSV in `inst/extdata`. Read one into a data frame with `read.csv()` on the path that `system.file("extdata", <file>, package = "rmoriebricklayer")` gives.

Source

Ontario Court of Justice, court statistics, ontariocourts.ca/ocj/statistics (retrieved 2026-10-13), editions 2012 to 2026, published by the Court for public use. Toronto Police Service, Public Safety Data Portal, dataset "Break and Enter Open Data", data.tps.ca (retrieved 2026-10-13); contains information licensed under the Open Government Licence - Toronto Police Service. See `inst/COPYRIGHTS`.

See Also

`caseflow`, `lifetbl`, `dispmix`, `clrdcmp`, `crmdens`, `knoxtst`.

Examples

```
f <- read.csv(system.file("extdata", "ocj_flow_ontario.csv",
                          package = "rmoriebricklayer"))
head(f)
c <- read.csv(system.file("extdata", "ocj_courts.csv",
                          package = "rmoriebricklayer"))
tapply(c$disposed, c$year, sum)
```

kemdecap *Recover a shared secret from an ML-KEM ciphertext*

Description

Returns the 32-byte shared secret that a ciphertext from `kemencap()` carries, as hex. This is the recipient's side of the exchange: with the secret half of the key pair, the recipient recovers the same secret the sender kept, and the two can then use it as a common key.

Usage

```
kemdecap(key, ct, masked = getOption("rmoriebricklayer.masked", TRUE))
```

Arguments

`key`	A key from `kemkg()`, with its secret half.
`ct`	The `ct` element from `kemencap()`, as a hex string or a raw vector, of the length for the key's level.
`masked`	Logical, `TRUE` or `FALSE`. `rmoriebricklayer.masked`, an option, gives the default when it is set; otherwise the default is `TRUE`. `TRUE` runs the first-order masked decapsulation, `FALSE` the plain one. Both return the same secret.

Details

This is ML-KEM.Decaps of FIPS 203 (Algorithm 21). For a ciphertext of the right length there is no failure path, and that is deliberate. A ciphertext that was not produced by a correct encapsulation under this key yields a shared secret derived from a value held only inside the decapsulation key – so it is a real secret, just not the sender's. Whoever sent it learns nothing about whether it was accepted, which is what closes off a chosen-ciphertext attack. The consequence for a caller: a mismatch between the two sides' secrets is the signal that something was wrong, not an error from this function.

Masked by default. With `masked = TRUE` every value that depends on the secret key is held as two random shares from the moment the key is read (first-order masking): decryption, the decoding of the message, the hashing, the re-encryption with its noise sampling, the comparison with the ciphertext and the choice between the real and the rejection secret. No single intermediate value then depends on the key, which is what defeats first-order power and electromagnetic analysis; `inst/tvla` checks the masked gadgets against that leakage model. The result is identical to the unmasked computation for every ciphertext; it costs about 2.3 to 3.4 times the plain time depending on the CPU (4 ms for ML-KEM-768 on this package's test machine against 1.2 ms plain). `masked = FALSE` runs the plain constant-time decapsulation, and the option `rmoriebricklayer.masked` sets the default for a session.

Errors (checked before any decapsulation): `key` not from `kemkg()`, or a public key only ("decapsulation needs a key with its secret half"); a ciphertext that is not hex or raw of the exact length for the key's level, which is `kemsz()[["ciphertext"]]` bytes (768, 1088 or 1568); `masked` not `TRUE` or `FALSE`.

Value

A length-1 character vector of 64 lower-case hex characters: the 32-byte shared secret.

See Also

kemencap().

Examples

```
key <- kemkg(512)
sent <- kemencap(kempubkey(key))
identical(kemdecap(key, sent$ciphertext), sent$shared)

# a tampered ciphertext returns a secret, and it is the wrong one
bad <- sent$ciphertext
substring(bad, 3L, 4L) <- "ff"
other <- kemdecap(key, bad)
nchar(other) == 64L
identical(other, sent$shared)

# the masked and the plain decapsulation agree
identical(kemdecap(key, bad, masked = FALSE), other)

# a public key alone cannot decapsulate, and a short ciphertext is refused
try(kemdecap(kempubkey(key), sent$ciphertext))
try(kemdecap(key, "abcd"))
```

---

kemencap *Encapsulate a shared secret under an ML-KEM key*

Description

Produces a ciphertext and the 32-byte shared secret it carries. Only the public key is needed, which is the point: the sender never holds anything the recipient has to trust them with. The sender keeps the shared secret (to derive an encryption key, for example) and sends the ciphertext; the recipient gets the same secret back with kemdecap().

Usage

```
kemencap(key, m = NULL)
```

Arguments

key	A key or public key from kemkg() / kempubkey().
m	Optional raw vector of 32 bytes of encapsulation randomness. Supplying it makes the operation reproducible, which is what the standard's test vectors need; the default draws from the operating system's CSPRNG. Reusing it across encapsulations to the same key reuses the shared secret, so supply it only deliberately.

Details

This is ML-KEM.Encaps of FIPS 203 (Algorithm 20) on the 32-byte message `m`. Without `m`, fresh randomness is drawn with `rndbytes()`, so two encapsulations to one key give different ciphertexts and secrets. A full key pair works as well as a public key; only its public half is read.

Errors: `key` not from `kemkg()` or `kempubkey()`; a key whose public half is not valid hex ("key has no usable public key"); `m` that is not a raw vector of 32 bytes.

Value

A list of class `bricklayer_kem_capsule` with `ciphertext` (character, hex, of the length in bytes that `kemsz()[["ciphertext"]]` gives: send this), `shared` (character, 64 hex characters, the 32-byte shared secret: keep this) and `level` (integer). Printing it never shows the shared secret.

See Also

`kemdecap()`, `kemkg()`.

Examples

```
key <- kemkg(512)
a <- kemencap(key)
nchar(a$ciphertext) / 2 == kemsz(512)[["ciphertext"]]

# two encapsulations to one key give different secrets
b <- kemencap(key)
identical(a$shared, b$shared)

# both decapsulate correctly
identical(kemdecap(key, a$ciphertext), a$shared)
identical(kemdecap(key, b$ciphertext), b$shared)

# fixed randomness makes the result reproducible (test vectors only)
m <- as.raw(1:32)
identical(kemencap(key, m = m)$shared, kemencap(key, m = m)$shared)

# the randomness must be exactly 32 bytes
try(kemencap(key, m = as.raw(1:8)))
```

---

kemkg *Generate an ML-KEM key pair*

Description

Generates a key pair for ML-KEM (FIPS 203), the standardised post-quantum key encapsulation mechanism, and `kempubkey()` strips a key pair down to the half that can be shared. Use them when a capsule, a data file or a credential has to reach one recipient without being readable on the way: the recipient makes the key pair and publishes the public half, and the sender uses it with `kemencap()`. It is implemented in this package, with no system dependency.

Usage

```
kemkg(lvl = 768L, seed = NULL)

kempubkey(key)
```

Arguments

`lvl`	Security level, one whole number: 512 (ML-KEM-512, NIST category 1), 768 (the default, category 3) or 1024 (category 5). 768 is the level NIST and the IETF have settled on for general use.
`seed`	Optional raw vector of 64 seed bytes (`d || z`) . Supplying it makes the key reproducible, which is what the standard's test vectors need; the default draws from the operating system's CSPRNG.
`key`	For `kempubkey()`: a key pair from `kemkg()`.

Details

A KEM is not a signature scheme and not a cipher. Encapsulation produces two things: a ciphertext to send, and a 32-byte shared secret to keep. The holder of the decapsulation key recovers the same secret from the ciphertext. What either side then does with that secret – feed it to a KDF, key an AEAD – is outside the mechanism.

Key generation is ML-KEM.KeyGen of FIPS 203 (Algorithm 19) from the 64-byte seed $d\|z$: d seeds the K-PKE key and z is the implicit-rejection value kept inside the decapsulation key. With a seed the result is deterministic (the same seed gives the same key, which is how the NIST test vectors are reproduced); without one the seed is drawn with `rndbytes()` from the operating system. The key sizes per level are given by `kemsz()`; the level is not recorded in a ciphertext, so both sides must agree on it.

Errors: `lvl` other than one whole number 512, 768 or 1024; `seed` that is not a raw vector of 64 bytes; `key` passed to `kempubkey()` that is not from `kemkg()`.

Value

`kemkg()`: a list of class `c("bricklayer_kem_key", "list")` with `public` (character, the encapsulation key as lower-case hex, 800, 1184 or 1568 bytes), `secret` (character, the decapsulation key as hex, 1632, 2400 or 3168 bytes) and `level` (integer). Printing it never shows the secret. `kempubkey()`: a list of class `c("bricklayer_kem_public_key", "list")` with `public` and `level` only.

Security

The implementation is hand-written. Its standardised schemes are checked byte for byte against OpenSSL 3.5 and NIST known-answer vectors in the test suite. Side channels are checked, not assumed: `inst/ctcheck` runs every operation that touches a secret under valgrind memcheck with the secret marked undefined (the ctgrind method), so a branch or a memory address that depends on it is a reported error, and afterwards scans the dead stack for copies of the secret. Both checks run in CI on every change, with GCC and with Clang. Timing is also measured on x86-64 and arm64 hardware (`inst/dudect`), power leakage is assessed in simulation under the value and the transition models (`inst/tvla`), and ML-KEM decapsulation and ML-DSA signing are first-order masked by

default. These are checks of this code on those compilers and under those leakage models; the README's security section lists what each check does and does not cover.

References

National Institute of Standards and Technology (2024). Module-Lattice-Based Key-Encapsulation Mechanism Standard. FIPS 203. DOI 10.6028/NIST.FIPS.203

See Also

`kemencap()`, `kemdecap()`, `kemsz()`, `fipskg()` for the signature schemes.

Examples

```
key <- kemkg(768)
pub <- kempubkey(key)

# the sender holds only the public key
sent <- kemencap(pub)
# the recipient recovers the same secret from the ciphertext
got <- kemdecap(key, sent$ciphertext)
identical(sent$shared, got)

# a corrupted ciphertext yields a DIFFERENT secret, not an error
bad <- sent$ciphertext
substring(bad, 1L, 2L) <- "00"
identical(kemdecap(key, bad), got)

# a fixed seed gives the same key every time (for test vectors only)
k1 <- kemkg(512, seed = as.raw(0:63))
k2 <- kemkg(512, seed = as.raw(0:63))
identical(k1$public, k2$public)
k1

# levels other than the three FIPS 203 defines are refused
try(kemkg(256))
try(kemkg(768, seed = as.raw(1:10)))
```

---

kemsz *Byte lengths of an ML-KEM parameter set*

Description

Reports the byte sizes that FIPS 203 fixes for one ML-KEM parameter set: the public (encapsulation) key, the secret (decapsulation) key, the ciphertext, the key-generation seed and the shared secret. Use it to check a key or ciphertext before storing or sending it, or to size a buffer or a database column, so a caller never has to hard-code the numbers.

Usage

```
kemsz(lvl)
```

Arguments

`lvl`	Security level, one whole number: 512, 768 or 1024. There is no default.

Details

The values come from the native core and are those of FIPS 203, Table 3:

	512	768	1024
encapsulation key	800	1184	1568
decapsulation key	1632	2400	3168
ciphertext	768	1088	1568
seed	64	64	64
shared secret	32	32	32

Hex strings from `kemkg()` and `kemencap()` are twice these lengths in characters. Error: `lvl` other than one whole number 512, 768 or 1024.

Value

A named integer vector of five elements, all in bytes: `public` (the encapsulation key), `secret` (the decapsulation key), `ciphertext`, `seed` and `shared` (the shared secret), named as `kemkg()` and `kemencap()` name their results.

See Also

`kemkg()`.

Examples

```
kemsz(768)

# the shared secret is 32 bytes at every level: the level buys
# security margin, not a longer secret
vapply(c(512, 768, 1024),
       function(l) kemsz(l)[["shared"]], integer(1))

# check a ciphertext's length before sending it
key <- kemkg(768)
ct <- kemencap(key)$ciphertext
nchar(ct) / 2 == kemsz(768)[["ciphertext"]]

# an unknown level is an error, not a default
try(kemsz(2048))
```

keyeddgst *Keyed digest and constant-time comparison (C backend)*

Description

`chmac()` computes HMAC-SHA-256 (RFC 2104), a digest of a message under a secret key, and `cdgsteq()` compares two digests without leaking through timing how much of them matched. Use the pair to sign a manifest or a capsule with a shared secret and to check a signature someone hands you.

Usage

```
chmac(key, msg)

cdgsteq(a, b)
```

Arguments

`key`	Secret key, as a length-1 character vector or a raw vector. Keys longer than the 64-byte block are hashed down first, per RFC 2104. Use at least 32 bytes of real entropy; against a quantum adversary Grover halves the effective key strength, so a 256-bit key retains a 128-bit margin.
`msg`	Message to authenticate, as a length-1 character vector (its bytes) or a raw vector.
`a`, `b`	Digests to compare, each a length-1 character vector.

Details

The difference from a plain `csha256()` is AUTHENTICATION – anyone can recompute a SHA-256 and so anyone can forge one after editing a manifest, but only a holder of the key can produce a matching HMAC. This is what makes `cpssign()`'s `"hmac"` scheme meaningful. The construction is

$$\mathrm{HMAC}(K, m) = H((K' \oplus opad) \| H((K' \oplus ipad) \| m))$$

with H = SHA-256, K' the key padded to the 64-byte block (or hashed first when longer), $ipad$ = `0x36` and $opad$ = `0x5c` repeated. The results match the RFC 4231 test vectors.

`cdgsteq()` compares two digests in constant time (verified under valgrind memcheck in CI with both inputs marked secret: `inst/ctcheck`, case `digest-equal`). Use it instead of == whenever the comparison is against a value an attacker supplied: a short-circuiting comparison leaks, through its own timing, how many leading characters were correct, which is enough to recover a tag byte by byte. It compares the strings exactly, so case matters (`"ABC"` is not `"abc"`): lower-case both first if the digests may come in either case.

Errors: `key` or `msg` that is neither one non-NA string nor a raw vector; `a` or `b` that is not one non-NA string.

Value

`chmac()` a length-1 character vector: 64 lowercase hex characters. `cdgsteq()` a length-1 logical; `FALSE` when the two differ in length.

References

Krawczyk H, Bellare M, Canetti R (1997). HMAC: Keyed-Hashing for Message Authentication. RFC 2104. DOI 10.17487/RFC2104

Examples

```
# RFC 4231 test case 2.
chmac("Jefe", "what do ya want for nothing?")

# The key changes the digest, so a manifest cannot be re-signed
# without it.
chmac("key-a", "manifest")
chmac("key-b", "manifest")

# Raw keys and messages are accepted.
chmac(as.raw(rep(0x0b, 20)), "Hi There")

# Compare tags in constant time, never with ==.
tag <- chmac("k", "m")
cdgsteq(tag, chmac("k", "m"))
cdgsteq(tag, chmac("k", "tampered"))
cdgsteq(tag, "too-short")

# one message at a time
try(chmac("k", c("m1", "m2")))
```

---

knoxtst *Near-repeat analysis: the Knox test for space-time clustering*

Description

Asks whether incidents close together in space also happen close together in time more often than chance would give – the near-repeat pattern of burglary and shootings. Pairs of incidents are classified by distance band and time band; each cell's count is compared with the count expected if the times were shuffled among the locations, and a Monte Carlo permutation test gives its p-value.

Usage

```
knoxtst(
  x,
  y,
  time,
  spcbands,
  tmbands,
  nsim = 999L,
  seed = NULL,
  xy = c("lonlat", "planar"),
  maxprs = 5e+07
)
```

Arguments

x, y	Locations: longitude and latitude in degrees by default (coords = "lonlat"), or planar coordinates in metres.
time	Times of the incidents: Date, POSIXct, or numbers in days.
spcbands	Upper edges of the distance bands in metres, increasing: c(100, 200, 400) is 0-100, 100-200 and 200-400 m.
tmbands	Upper edges of the time bands in days, increasing.
nsim	Monte Carlo permutations. Default 999.
seed	Optional seed for the permutations, a whole number from -2147483647 to 2^53 - 1; the caller's random stream is left as it was.
xy	"lonlat" or "planar".
maxprs	Limit on candidate pairs held in memory. Default 5e7.

Details

Under the null hypothesis that the times are exchangeable among the locations, every unordered pair of locations receives a uniformly random pair of times, so the expected count in a cell is exact: $E_{st} = S_s T_t / P$, where S_s is the number of pairs in distance band s, T_t the number in time band t, and $P = n(n-1)/2$. The ratio is O_{st}/E_{st}: above 1, pairs that close in space are that close in time more often than chance. The p-value of each cell is one-sided, $(1 + \#\{O^* \geq O\})/(1 + nsim)$ over the permutations. Cells are tested separately; a table of many cells will show some small p-values by chance alone.

Only pairs within the largest distance band are needed, and they are found by hashing the locations to a grid, so the work grows with the number of close pairs rather than with n^2.

Ties in time – incidents recorded to the day – put same-day pairs in the first time band (0 to its edge). The test speaks to the timing of incidents relative to each other, not to what caused them; near-repeat patterns also arise when one offender commits a series.

Errors: x, y and time of different lengths; spcbands or tmbands not positive, finite and increasing; nsim below 19; maxprs below 1; longitude and latitude out of range under coords = "lonlat"; fewer than three incidents. With seed the random number stream is set for the call and restored afterwards.

Value

A list of class rmbl_knox: table, a long data frame with one row per distance-by-time cell (spclo, spcup, tmlo, tmup, observed, expected, ratio, p_value); the n incidents used; nsim.

References

Knox, E. G. (1964). The detection of space-time interactions. *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, 13(1), 25-30.

Mantel, N. (1967). The detection of disease clustering and a generalized regression approach. *Cancer Research*, 27(2), 209-220.

Johnson, S. D., Bernasco, W., Bowers, K. J., Elffers, H., Ratcliffe, J., Rengert, G. and Townsley, M. (2007). Space-time patterns of risk: a cross national assessment of residential burglary victimization. *Journal of Quantitative Criminology*, 23(3), 201-219.

See Also

crmdens.

Examples

```
b <- read.csv(system.file("extdata", "tps_break_enter_d14_2024.csv",
                          package = "rmoriebricklayer"))
b <- b[!is.na(b$lon), ]
kn <- knoxtst(b$lon, b$lat, as.Date(b$occ_date), spcbands = c(100, 200, 400),
              tmbands = c(7, 14, 28), nsim = 199, seed = 1)
kn
plot(kn)

# planar coordinates in metres and times in days, on simulated points
set.seed(5)
n <- 150
kp <- knoxtst(runif(n, 0, 2000), runif(n, 0, 2000), runif(n, 0, 365),
              spcbands = c(200, 400), tmbands = c(14, 28),
              nsim = 99, seed = 2, xy = "planar")
kp

# bands must increase
try(knoxtst(b$lon, b$lat, as.Date(b$occ_date), c(400, 100), c(7, 14)))
```

---

ldgrread *Read and check a proof ledger*

Description

A proof ledger records the theorems of a formal-proof build (Lean and Mathlib): each theorem's statement, file and line, its status (`proved`, `sorry` or `axiom`) and number of `sorry` placeholders, the package functions whose behaviour it states, and optionally a numeric check of one of them in R. The format is `schemapath("ledger")`, in the style of the capsule schemas; nothing in it is specific to this package, so any package can ship a ledger in `inst/ledger/` and any formal-proof project can write one.

Usage

```
ldgrread(path)

## S3 method for class 'rmbl_ledger'
format(x, ...)

ldgrchk(ledger, pkgs = NULL)

## S3 method for class 'rmbl_ledger_check'
format(x, ...)
```

```
## S3 method for class 'rmbl_ledger_check'
print(x, ...)

## S3 method for class 'rmbl_ledger'
print(x, ...)
```

Arguments

path	Character scalar, the ledger JSON file.
x	A ldgrread() or ldgrchk() result.
...	Not used.
ledger	A ldgrread() result, or the path of a ledger file.
pkgs	Character vector: check only the covered functions and checks of these packages; NULL (the default) checks every package named.

Details

ldgrread() parses the file and validates it against the schema; a file that does not validate, or that repeats a theorem id, is refused with the first problems found.

ldgrchk() returns check rows in the shape of rmbl verify (the rows of cpschk()):

proved: ID the theorem is proved with sorry_count 0. A sorry fails as an error; an axiom fails as a warning (it is assumed, not proved).

covered: ID PKG::FN the covered function is exported by the installed package; a package that is not installed here is a note, not a failure (it cannot be checked).

check: ID the theorem's check$expr evaluates within check$tol (default 1e-8) of check$expect. The expression is evaluated in a sandbox: an environment holding only the exported functions of the first R package the theorem covers and a short list of base arithmetic and constructors (c, list, seq, rep, sum, +, $, ...). Anything else is refused before evaluation: a call to any other function (system(), file(), eval(), another package), a free variable, or an expression whose outer call is not a call to that package (or an extraction from one). Evaluation is limited to 10 seconds.

ledger_commit_pinned, ledger_build_pinned when the ledger is the data file of a capsule whose provenance says kind: "ledger": the provenance's source$commit and source$build_digest equal the ledger's. The capsule's own rows (vrfcps(), the bundle and attestation through cpsbndlvrf(), the schemas) come with them, as rmbl verify reports them; rmbl verify adds the ledger rows to such a capsule too.

Value

ldgrread(): the ledger, a list of class rmbl_ledger, with the normalised path as the attribute path. ldgrchk(): a data frame (class rmbl_ledger_check, which prints as a box) with the columns capsule, part (ledger, or the capsule parts), file, check, ok, detail, severity (error, warn or note) and line.

See Also

schemapath(), cpschk(), cpsbndlvrf().

Examples

```
f <- system.file(
  "extdata",
  "ledger_sample.json",
  package = "rmoriebricklayer"
)
l <- ldgrread(f)
l
r <- ldgrchk(l)
r[, c("check", "ok", "severity", "detail")]

# a file that is not a ledger
bad <- tempfile(fileext = ".json")
writeLines('{"name": "x"}', bad)
try(ldgrread(bad))
unlink(bad)
```

---

lifetbl *Life table for grouped times: how long cases take*

Description

The actuarial (Cutler-Ederer) life table for times published only in bands – for example the share of court cases disposed within 10 months, between 10 and 15 months, and so on. It gives, at the end of each band, the proportion still open (the survival function) with a Greenwood confidence interval, the hazard within the band, and the median time.

Usage

```
lifetbl(
  breaks,
  events,
  cens = 0,
  conflvl = 0.95,
  conftype = c("log-log", "log", "plain")
)
```

Arguments

breaks	Band edges, increasing, starting at the time origin: `c(0, 10, 15, 18, Inf)` for the four bands above. The last edge may be `Inf` (an open band). Or band labels such as `c("Under 10", "10-15", "15-18", "Over 18")`, read with prsbands.

`events`	Cases that ended in each band (counts, or expected counts such as a published share times the number of cases).
`cens`	Cases withdrawn from observation in each band without the event (still open when the data were drawn); one value or one per band. Default 0.
`conflvl`	Confidence level. Default 0.95.
`conftype`	Scale of the interval: `"log-log"` (the default, which keeps the limits inside 0 to 1), `"log"` or `"plain"`.

Details

Within a band the censored cases are assumed to have been at risk for half of it, so the number at risk is $n'_j = n_j - c_j/2$; the conditional probability of the event is $q_j = d_j/n'_j$ and the survival at the end of band j is $\prod_{i \le j}(1 - q_i)$. Greenwood's formula gives $\mathrm{Var}(\hat{S}_j) = \hat{S}_j^2 \sum_{i \le j} q_i/(n'_i(1 - q_i))$. With no censoring the table equals the Kaplan-Meier estimate at the band edges, and the Greenwood variance equals the one `survival::survfit()` reports. The hazard in a closed band is $d_j/(w_j(n'_j - d_j/2))$ for band width w_j. The median is interpolated linearly inside the band where survival crosses one half, which assumes events are spread evenly across that band; when the crossing falls in an open band, the median is reported as greater than its lower edge.

Court statistics that count only disposed cases describe time to disposition among cases that were disposed, with no censoring; cases still pending are not in them, so the survival function describes the disposed cohort, not every case received.

The median is attribute `"median"` (`Inf`, with a `note`, when survival never falls to one half or crosses it in the open band), and attribute `"lifetbl"` records `conflvl` and `conftype`. Errors: band labels that cannot be read, or that are not contiguous, or with an open band other than the last; fewer than two increasing edges, or an infinite edge other than the last; `events` or `cens` that are not non-negative counts, one per band; no cases at all; `conflvl` outside (0, 1); an unknown `conftype`.

Value

A data frame of class `rmbl_life_table`, one row per band: `lower`, `upper` (the band edges), `entering`, `events`, `censored`, `atrisk` (entering less half the censored), `q` (conditional probability of the event in the band), `survival` at the end of the band with `se`, `conflo` and `confup`, `hazard` (events per unit time within the band; `NA` for an open band), and the attribute `median`.

References

Cutler, S. J. and Ederer, F. (1958). Maximum utilization of the life table method in analyzing survival. *Journal of Chronic Diseases*, 8(6), 699-712.

Greenwood, M. (1926). *The Natural Duration of Cancer*. Reports on Public Health and Medical Subjects 33. HMSO, London.

Lee, E. T. and Wang, J. W. (2003). *Statistical Methods for Survival Data Analysis*, 3rd ed. Wiley, Hoboken, NJ. Chapter 4.

See Also

`caseflow`, `prsbands`.

Examples

```
d <- read.csv(system.file("extdata", "ocj_time_to_disposition.csv",
                          package = "rmoriebricklayer"))
f <- read.csv(system.file("extdata", "ocj_flow_ontario.csv",
                          package = "rmoriebricklayer"))
last <- nrow(d)
shares <- unlist(d[last, c("under_10", "m10_15", "m15_18", "over_18")])
lt <- lifetbl(c(0, 10, 15, 18, Inf), events = shares * f$disposed[last])
lt
plot(lt)

# bands given by their labels, with some cases still open, on the log scale
lt2 <- lifetbl(c("Under 10", "10-15", "15-18", "Over 18"),
                  events = c(600, 150, 60, 90), cens = c(40, 20, 10, 0),
                  conftype = "log")
as.data.frame(lt2)[, c("lower", "upper", "atrisk", "survival",
                       "conflo", "confup")]
attr(lt2, "median")

# an open band can only come last
try(lifetbl(c(0, Inf, 20), events = c(5, 5)))
```

---

limfrom *Confidence limits from a standard error, a coefficient of variation or a margin of error*

Description

The comparisons of this family take an estimate and its published confidence limits. Agencies that publish a standard error, a relative standard error (coefficient of variation) or a margin of error instead give the same information; this turns it into limits at the level wanted.

Usage

```
limfrom(est, se = NULL, cv = NULL, moe = NULL, moelev = 0.9, conflvl = 0.95)
```

Arguments

`est`	The estimates the limits are centred on.
`se`, `cv`, `moe`	Give one: standard errors, coefficients of variation (relative standard errors, in percent), or margins of error.
`moelev`	The confidence level of the margins of error (default 0.90).
`conflvl`	The level of the limits returned (default 0.95).

Details

With an estimate e and $z = \Phi^{-1}(1 - \alpha/2)$ for the level wanted, a standard error s gives $e \pm zs$; a coefficient of variation c (in percent) gives $s = ec/100$; a margin of error m at level `moelev` (the American Community Survey publishes 90% margins) gives $s = m/z_{moe}$. The limits are symmetric on the estimate's scale.

Value

A data frame: `estimate`, `lower`, `upper`, `se`.

See Also

`qltyrcd`, `indgap`.

Examples

```
limfrom(c(24.1, 31.5), cv = c(8, 12))
limfrom(1520, moe = 210)                # a 90% margin of error
```

`llmask` *Ask a language model*

Description

Sends one prompt to the first language model that is set up on this machine and returns the reply. Bring your own key, or run a model locally: nothing is sent anywhere you did not set up. In order:

1. your own OpenAI-compatible endpoint, `MORIE_LLM_BASE_URL` (with `MORIE_LLM_API_KEY` and `MORIE_LLM_MODEL` when it needs them), e.g. LM Studio, vLLM, llama.cpp or a company gateway. An `https` address, or plain `http` to the loopback host or to a literal private LAN address (`10/8`, `172.16/12`, `192.168/16`, `fc00::/7`: an LM Studio or vLLM box on your own network); a host name over plain http is refused. It is not probed, so a configured endpoint that is down is an error, not a silent hop elsewhere;
2. a provider whose key is set (see `llmprvd`): OpenAI, Anthropic, Google Gemini, OpenRouter, Groq, Together, Fireworks, DeepSeek, Mistral, xAI, Cerebras, Perplexity, Hugging Face, OpenCode Zen, Cohere and NVIDIA, each through its own documented API with its own key variable (`ANTHROPIC_API_KEY`, `OPENAI_API_KEY`, ...);
3. a local Ollama server at `OLLAMA_HOST`, by default `http://localhost:11434`, with the model `OLLAMA_MODEL` or the first one pulled; `OLLAMA_HOST=off` skips it.

`llmstat` shows what is set up and which route a question would take.

Usage

```
llmask(
  prompt,
  model = NULL,
  tmout = 120,
  syspro = NULL,
  route = NULL,
  docs = TRUE,
  pkgs = NULL,
  fresh = FALSE
)
```

Arguments

prompt	Character scalar: the question, in plain text.
model	Character scalar or NULL (the default): the model id to use (a value shaped like a key is refused, never sent). NULL uses the route's own choice: MORIE_LLM_MODEL for your own endpoint, OLLAMA_MODEL or the first pulled model for Ollama, the saved PROVIDER.model or the provider's default model (see llmprvd).
tmout	Numeric scalar, seconds to wait for each reply; default 120. With docs = TRUE the whole exchange gets at most three times this.
syspro	Character scalar or NULL (the default): extra instructions sent as the system message (added to the package's own instructions when docs = TRUE).
route	Character scalar or NULL (the default: the saved route, else the first that is set up). "auto" uses the automatic order and ignores a saved route; "own", "ollama" or a provider id insists on that one.
docs	Logical scalar, default TRUE: give the model the packages' help and the lookup tools (see Details). FALSE sends the prompt alone.
pkgs	Character vector or NULL (the default): the installed packages whose help the model gets. NULL means "rmoriebricklayer", or the comma-separated list in the environment variable RMBL_PACKAGES (which a package built on this one sets for its own command line). Ignored when docs = FALSE.
fresh	Logical scalar, default FALSE. TRUE asks the model even when a saved answer matches (see Details), and saves the new answer in its place.

Details

With docs = TRUE (the default) the model is given the help of the packages named in pkgs (read from the help pages installed here) and may call five read-only tools before it answers: search_docs, get_help, get_examples, list_datasets and get_vignette. How much help goes into the instructions depends on the model's context size: a model with at least 32,000 tokens gets an index of every exported function (name, arguments, title) and up to six rounds of lookups; 8,000 to 32,000 tokens, the 120 functions nearest the question and four rounds; smaller, the 40 nearest and three rounds. A package with very many pages is never listed whole: its functions nearest the question are listed and the tools reach the rest. "Nearest" is a BM25 search over the help text in R ($k_1 = 1.2$, $b = 0.75$). A model that cannot call tools gets the best matching pages in the prompt instead. Nothing the model asks for is run: it answers with code for you to run.

With docs = TRUE the whole exchange is given at most three times tmout; a lookup round that would start after that is not started and the call errors. With docs = FALSE one request is made with syspro (when given) and prompt.

Saved answers: a person at a terminal (the command line with a terminal on both stdin and stdout) or in an interactive GUI session gets a saved answer back at once when the question (spacing aside; case counts), route, address, model, packages and their versions, and syspro all match. Answers are kept as owner-only files in the folder that tools::R_user_dir() gives for "rmoriebricklayer" and "cache": for 30 days, which the option rmoriebricklayer.llmcachedays changes, and at most the 200 newest, which the option rmoriebricklayer.llmcachemax changes. The first save in a session says where answers go. Nothing is saved from a script, a pipeline, a test run, R CMD check, or with options(morie.quiet = TRUE). Saving is also turned off by MORIE_LLM_CACHE=off, or by the option rmoriebricklayer.llmcache set to FALSE.

Errors: prompt is not a single non-empty string; fresh or docs is not TRUE or FALSE; route = "hosted" or an unknown route; no route is set up (the message lists what to set); no model is chosen for the route; pkgs names a package that is not installed; and, after sending, an unreachable server, a rejected key (HTTP 401 or 403) or an error reply. The key is redacted from any error text a server echoes back.

Value

A character scalar with the reply. A saved answer carries the time it was saved as attribute cached.

See Also

llmchat for a conversation without the help; llmstat and llmrte to see where a question would go.

Examples

```
old <- Sys.getenv(c("XDG_CONFIG_HOME", "OLLAMA_HOST"), unset = NA)
Sys.setenv(XDG_CONFIG_HOME = tempfile(), OLLAMA_HOST = "off")

# arguments are checked before any route is resolved
try(llmask(c("two", "questions")))

# a provider named explicitly without its key
try(llmask("What is a capsule?", route = "anthropic"))

# no route set up: the error lists what to set up
try(llmask("What is a capsule?", route = "ollama", docs = FALSE))

for (v in names(old)) {
  if (is.na(old[[v]])) {
    Sys.unsetenv(v)
  } else {
    do.call(Sys.setenv, as.list(old[v]))
  }
}
```

```
# needs your own key or a local model, and the network
Sys.setenv(ANTHROPIC_API_KEY = "...")
llmask("What does a SHA-256 provenance record protect against?")
llmask("Explain an E-value of 2.1", route = "ollama")
llmask("Summarise Benford's law in two lines", docs = FALSE,
                   syspro = "You are terse.")
```

`llmcfg` *Show or save the language-model settings*

Description

Every setting that decides how `llmask` reaches a model, in one place: which route to use; your key and model for each provider (bring your own key: see `llmprvd`); the address, key and model of your own OpenAI-compatible server; and of a local or LAN Ollama server. Called with no arguments it only reads: it returns a table of every setting, its current value, and where that value comes from. Called with `key = value` pairs it saves those settings first. Keys are never printed, only whether one is set.

Usage

```
llmcfg(...)
```

Arguments

`...`	Settings to save, as `key = value`. The keys are: • `route`: `"auto"`, `"own"`, `"ollama"` or a provider id such as `"anthropic"` (case is ignored); • `<provider>.key` and `<provider>.model`, for example `anthropic.key` or `openai.model`; • `own.url`, `own.key` and `own.model`; • `ollama.url`, `ollama.model` and `ollama.key`. Each value is a single string (leading and trailing spaces are removed). `NULL`, `NA` or `""` removes a saved setting. With no arguments nothing is written.

Details

Where settings live. Saved settings are kept in `llm.json` in the folder `$XDG_CONFIG_HOME/morie` (`~/.config/morie` when `XDG_CONFIG_HOME` is unset), a private file (mode 0600, in a private directory) next to the shared `credentials.json`. The file is written only when you pass a setting; only the settings that change are rewritten in place, so the rest of the file stays exactly as it was. An empty set of settings removes the file.

Precedence. An environment variable that is set (the `env` column, for example `ANTHROPIC_API_KEY` or `OLLAMA_HOST`) wins over a saved value, which wins over the built-in default. Saving a setting whose variable is set warns that the variable still wins. A provider key is also read from the

provider's alternative variable where it has one: GOOGLE_API_KEY for Gemini, CO_API_KEY for Cohere, and for Hugging Face HUGGINGFACE_HUB_TOKEN.

Checks when saving. Names must be among the settings listed in the key column ("unknown setting" otherwise) and every value passed as key = value with one value each. route must be "auto", "own", "ollama" or a provider id; the retired "hosted" route is refused. own.url must be an http(s):// address and ollama.url an address or host:port; both are refused if they contain spaces, a fragment or user information, and, unless they point at this machine or a private LAN address, they get the package's public-address check (a metadata or link-local address is refused). "off", "none" or "disabled" turns a server off. A key with a line break or control character in it is refused. A new own.url or ollama.url drops the key saved for the old server (with a message), so a key is never sent to a server it was not saved for. A settings file that is not a JSON object, or holds a setting that is not a non-empty string, stops every call until it is fixed or moved aside with rmbl config reset.

Reading the settings takes a few milliseconds: the table of settings is built once per session and the file is read once per call. Nothing is sent over the network by this function.

The same is available from the shell: rmbl config lists the settings, rmbl config set KEY VALUE and rmbl config unset KEY change one (a key is asked for without echo when no value is given), and rmbl config setup walks through them.

Value

A data frame with one row per setting (seven general settings followed by a key and a model row for each provider) and five character columns: key (the setting's name), value (its current value, the default spelled out; a key only as "set" or "(not set)"), source ("environment", "saved" or "default"), env (the environment variable that overrides it) and help (what it is for). Returned visibly when called with no settings, invisibly after saving.

See Also

llmprvd for the providers and their variables, llmstat for which routes are reachable, llmask which uses these settings.

Examples

```
# Work in a temporary settings directory so your own settings are
# untouched.
old <- Sys.getenv("XDG_CONFIG_HOME", unset = NA)
Sys.setenv(XDG_CONFIG_HOME = tempfile("cfg"))

# Reading the settings writes nothing.
cfg <- llmcfg()
head(cfg[, c("key", "value", "source")])
nrow(cfg)

# Save a local Ollama model and use it for every request; the key column
# shows where each value now comes from.
llmcfg(ollama.model = "qwen3:8b", route = "ollama")
cfg <- llmcfg()
cfg[cfg$key %in% c("route", "ollama.model"), c("key", "value", "source")]
```

```
# A key is shown only as set or not set.
llmcfg(own.url = "http://localhost:1234/v1", own.key = "k-123")
cfg <- llmcfg()
cfg$value[cfg$key == "own.key"]

# Back to the defaults: NULL removes a saved setting.
llmcfg(route = NULL, ollama.model = NULL, own.url = NULL, own.key = NULL)

# Unknown settings and retired routes are refused.
try(llmcfg(colour = "blue"))
try(llmcfg(route = "hosted"))

if (is.na(old)) {
  Sys.unsetenv("XDG_CONFIG_HOME")
} else {
  Sys.setenv(XDG_CONFIG_HOME = old)
}


# your Anthropic key and model, and always use them
llmcfg(anthropic.key = "sk-ant-...", route = "anthropic")
# an Ollama server on another machine
llmcfg(ollama.url = "http://192.168.1.20:11434", ollama.model = "qwen3:8b")
```

`llmchat` *A conversation with a language model*

Description

Sends a whole conversation (system, user and earlier assistant turns) to the route `llmask` would take, and returns the model's next reply. Use it when you manage the conversation yourself, for example to keep a multi-turn exchange or to give the model earlier answers as context. Unlike `llmask()` it adds nothing to the conversation (no help pages, no tools) and it never reads or writes the saved-answer cache.

Usage

```
llmchat(msgs, model = NULL, route = NULL, tmout = 120)
```

Arguments

`msgs`	A non-empty list of turns, each a list with two character scalars: `role`, one of `"system"`, `"user"` or `"assistant"`, and `content`, the text of that turn. The last turn is normally a `"user"` turn. Other elements of a turn are dropped.
`model`	Character scalar or `NULL` (the default): the model id to use (a value shaped like a key is refused, never sent). `NULL` uses the route's own choice: `MORIE_LLM_MODEL` for your own endpoint, `OLLAMA_MODEL` or the first pulled model for Ollama, the saved `PROVIDER.model` or the provider's default model (see `llmprvd`).

route	Character scalar or NULL (the default: the saved route, else the first that is set up). "auto" uses the automatic order and ignores a saved route; "own", "ollama" or a provider id insists on that one.
tmout	Numeric scalar, seconds to wait for each reply; default 120. With docs = TRUE the whole exchange gets at most three times this.

Details

The route is resolved exactly as in llmrte. The turns are sent in the provider's own wire format (the OpenAI chat completions format for most providers, your own endpoint and Ollama; Anthropic's Messages API; Google's Gemini generateContent). One request is made, through the package's own libcurl binding, and waits at most tmout seconds.

Errors, before anything is sent: msgs is not a non-empty list of turns each with a single role among "system", "user" and "assistant" and a single non-missing character content; route = "hosted" or an unknown route; no route is set up (the message says what to set); the route has no model and model is not given. After sending: the server cannot be reached, rejects the key (HTTP 401 or 403), or answers with an error. The key is redacted from any error text a server echoes back.

Value

A character scalar: the text of the model's reply.

See Also

llmask for one question with the package help at hand.

Examples

```
old <- Sys.getenv(c("XDG_CONFIG_HOME", "OLLAMA_HOST"), unset = NA)
Sys.setenv(XDG_CONFIG_HOME = tempfile(), OLLAMA_HOST = "off")

# a conversation is a list of role/content turns
turns <- list(
  list(role = "system", content = "Answer in one sentence."),
  list(role = "user", content = "Name one type of survey weight.")
)
str(turns)

# a malformed turn is refused before anything is sent
try(llmchat(list("Name one type of survey weight.")))

# with no route set up the error says what to set up
try(llmchat(turns, route = "ollama"))

for (v in names(old)) {
  if (is.na(old[[v]])) {
    Sys.unsetenv(v)
  } else {
    do.call(Sys.setenv, as.list(old[v]))
  }
}
```

```
# needs a key or a local model, and the network
Sys.setenv(ANTHROPIC_API_KEY = "...")
llmchat(turns)
llmchat(c(turns, list(
  list(role = "assistant", content = "A design weight."),
  list(role = "user", content = "And a second one?")
)))
```

`llmmdls` *Models a language-model route offers*

Description

Asks the provider (or your own server, or a local Ollama server) which models your key can use, so you can pick one for the `model` argument of `llmask` or save it with `rmoriebricklayer config set PROVIDER.model NAME`. The request goes through the package's own libcurl binding; no prompt is sent.

Usage

```
llmmdls(route = NULL, tmout = 15)
```

Arguments

`route`	Character scalar or `NULL` (the default: the route `llmask` would take). Otherwise `"auto"`, `"own"`, `"ollama"` or a provider id from `llmprvd`.
`tmout`	Numeric scalar, seconds to wait for the model list; default `15`.

Details

The route is resolved as in `llmrte`. The list comes from the provider's model endpoint: `GET /models` in the OpenAI format (own endpoint and most providers), `GET /v1/models` for Anthropic, `GET /v1beta/models` for Gemini (only models that support `generateContent` are kept), and `GET /api/tags` for Ollama. Perplexity publishes no model list, so its route returns an empty vector. A failed request, a non-200 answer or a reply that is not JSON also gives an empty vector, not an error, so check `llmstat` when the list is unexpectedly empty. Names are cleaned of control characters before they are returned.

Errors only while resolving the route: `"hosted"`, an unknown route, a provider named with no key set, or a `tmout` that is not a single positive number.

Value

A character vector of model ids (possibly empty) with two attributes: `"default"`, the model `llmask()` uses when `model` is not given (character, or `NULL` when none is chosen), and `"route"`, the readable name of the route asked (such as `"OpenAI"` or `"local Ollama"`; `NULL` when no route is set up).

Examples

```
old <- Sys.getenv(c("XDG_CONFIG_HOME", "OLLAMA_HOST"), unset = NA)
Sys.setenv(XDG_CONFIG_HOME = tempfile(), OLLAMA_HOST = "off")

# no route set up: an empty vector, nothing contacted
m <- llmmdls(route = "ollama")
length(m)
attributes(m)

# a provider named without its key is an error
try(llmmdls(route = "anthropic"))

for (v in names(old)) {
  if (is.na(old[[v]])) {
    Sys.unsetenv(v)
  } else {
    do.call(Sys.setenv, as.list(old[v]))
  }
}


# with a key set, the provider is asked (network)
m <- llmmdls(route = "openai", tmout = 10)
head(m)
attr(m, "default")
```

`llmprvd` *The language-model providers a key can be brought for*

Description

Lists every provider `llmask` can reach with your own key: the environment variable that holds the key, the address the key and prompts are sent to, the wire format used, and whether a key is set on this machine. Use it to find the variable name to set for a provider, or the `id` to pass as `route`. Nothing is contacted.

Usage

```
llmprvd()
```

Details

The table is fixed in the package (the providers' documented addresses and key variables). A key counts as set when the provider's variable (`key_env`), its alternative spelling (`alt_env`), or a key saved with `rmoriebricklayer config set ID.key` is non-empty; key values are never returned. A key with a line break or control character in it is an error. A server that speaks the OpenAI format but is not in the table is used as your own endpoint (`MORIE_LLM_BASE_URL`).

Value

A data frame with one row per provider (16 rows) and eight columns: `id` (character, the value `route` = takes), `label` (character, readable name), `base` (character, the API base address), `api` (character, the wire format: `"openai"`, `"anthropic"` or `"gemini"`), `key_env` (character, the environment variable for the key), `alt_env` (character, an alternative variable, `""` when none), `default_model` (character, the model used when none is chosen, `NA` when the provider has no default) and `key_set` (logical, whether a key is set here).

See Also

`llmstat`, `llmrte`.

Examples

```
p <- llmprvd()
p[, c("id", "key_env", "key_set")]

# the providers that use the Anthropic or Gemini wire format
p[p$api != "openai", c("id", "api", "base")]

# providers with a default model
p[!is.na(p$default_model), c("id", "default_model")]
```

---

llmrte *The language-model route that would answer*

Description

Reports which route `llmask` and `llmchat` would take right now, without sending a prompt anywhere. Use it to check, before asking a question, that the question would go to the server you expect (your own endpoint, a provider whose key is set, or a local Ollama server) and which model it would use.

Usage

```
llmrte(route = NULL)
```

Arguments

`route`	Character scalar or `NULL` (the default). `NULL` uses the saved route, else the automatic order; `"auto"` uses the automatic order and ignores a saved route; `"own"`, `"ollama"` or a provider id from the `id` column of `llmprvd` (such as `"anthropic"`) asks about that one route only.

Details

The automatic order is: an OpenAI-compatible endpoint of your own (MORIE_LLM_BASE_URL, or own.url saved with rmoriebricklayer config set); then every provider of llmprvd whose key is set, in the order of that table; then a local Ollama server, at OLLAMA_HOST or by default at http://localhost:11434, when it answers with at least one pulled model or OLLAMA_MODEL is set. A saved route replaces the automatic order when route = NULL; it comes from MORIE_LLM_ROUTE or from the route saved in the settings file.

Nothing is sent to a provider. The only network contact is the Ollama probe, a GET /api/tags to the Ollama address with a 2 second timeout; OLLAMA_HOST=off skips it. Your own endpoint is not probed.

Errors: route = "hosted" (the hosted MORIE tier is retired); a route that is not one of the accepted names; a provider route named explicitly with no key set; a MORIE_LLM_BASE_URL that is not https to a public address or plain http to the loopback host or a literal private LAN address. When automatic resolution skips a route that is half set up (a model or key without an address, an Ollama that does not answer) a message says so.

Value

NULL when no route is set up (or the named one is not available), else a list of four elements: route (character, the id that route = takes: "own", "ollama" or a provider id), name (character, a readable label such as "own endpoint", "local Ollama" or "Anthropic"), model (character, or NULL when no model is chosen) and base (character, the base address requests go to, without a trailing /v1 for your own endpoint).

See Also

llmstat for every route at once, llmprvd for the provider ids.

Examples

```
# keep the examples away from your real settings and from the network
old <- Sys.getenv(
  c(
    "XDG_CONFIG_HOME",
    "OLLAMA_HOST",
    "MORIE_LLM_BASE_URL",
    "MORIE_LLM_MODEL"
  ),
  unset = NA
)
Sys.setenv(XDG_CONFIG_HOME = tempfile(), OLLAMA_HOST = "off")

# Ollama switched off: that route is not available
llmrte("ollama")

# an OpenAI-compatible server of your own on this machine (not contacted
# here)
Sys.setenv(
  MORIE_LLM_BASE_URL = "http://localhost:8080/v1",
  MORIE_LLM_MODEL = "llama3"
```

```
)
str(llmrte())

# the retired hosted tier, and a route name that does not exist
try(llmrte("hosted"))
try(llmrte("nope"))

for (v in names(old)) {
  if (is.na(old[[v]])) {
    Sys.unsetenv(v)
  } else {
    do.call(Sys.setenv, as.list(old[v]))
  }
}
```

`llmstat` *Report the language-model routes available from this machine*

Description

Shows, in one table, every way `llmask` could reach a language model from this machine and whether each is ready: your own endpoint, each provider whose key is set, and a local Ollama server. Use it to find out why a question fails with "no route", or which model a route would use. The command line prints it as part of `rmoriebricklayer doctor`.

Usage

```
llmstat()
```

Details

Rows come in the order `llmask()` tries the routes. The providers without a key share one row that names each one with its key variable. Key values are never shown. Nothing is sent to a provider; the only network contact is a `GET /api/tags` to the Ollama address (`OLLAMA_HOST`, default `http://localhost:11434`; 2 second timeout) unless `OLLAMA_HOST=off`.

A settings file (`llm.json` next to the credentials file) that cannot be read does not make the call fail: the result is then a single row with `route = "settings"` and `status = "unreadable"`, with the reason and the command that moves the file aside.

Value

A data frame with one row per route and three character columns: `route` (`"own endpoint"`, a provider label such as `"Anthropic"`, `"other providers"` or `"local Ollama"`), `status` (`"not set"` or `"configured"` for your own endpoint; `"key set"` for a provider; `"no key"` for the shared row; `"disabled"`, `"not running"`, `"no models"` or `"available"` for Ollama) and `detail` (the address and model, or what to set up).

See Also

`llmrte` for the one route a question would take; `llmprvd`.

Examples

```
old <- Sys.getenv(
  c(
    "XDG_CONFIG_HOME",
    "OLLAMA_HOST",
    "MORIE_LLM_BASE_URL",
    "MORIE_LLM_MODEL"
  ),
  unset = NA
)
# a scratch settings folder and the local route off: no network at all
Sys.setenv(XDG_CONFIG_HOME = tempfile(), OLLAMA_HOST = "off")
st <- llmstat()
st[, c("route", "status")]

# your own OpenAI-compatible server shows as configured (it is not contacted)
Sys.setenv(
  MORIE_LLM_BASE_URL = "http://localhost:8080",
  MORIE_LLM_MODEL = "llama3"
)
llmstat()[1, ]

for (v in names(old)) {
  if (is.na(old[[v]])) {
    Sys.unsetenv(v)
  } else {
    do.call(Sys.setenv, as.list(old[v]))
  }
}
```

`loadprov` *Load a Pinned Data-Provenance Record*

Description

Reads a `data_provenance.json` file, the record that pins where a project's data came from: the portal endpoint (CKAN, Socrata or ArcGIS), the resource name pattern, the expected SHA-256 of the file, a Wayback Machine snapshot, the schema the data must match and the recipe for a synthetic stand-in. The resolvers (`rslvckan()`, `rslvsoc()`, `rslvarc()`) and the checks (`vrfsha256()`, `valsch()`) all take the list this returns.

Usage

```
loadprov(path)
```

Arguments

path	Path to the provenance JSON file, a single string.

Details

The file is parsed with the package's own JSON codec (see fromjs()) without simplification: JSON arrays stay lists, so a pinned digest such as "sha256": "abc" is a length-1 character and an array of column names is a list of strings (use unlist() where a vector is needed). Nothing is checked about the content here; valsch() reads the schema block.

A missing file is not an error: it returns NULL, so a project can run without a provenance record and every resolver treats NULL as "nothing pinned".

Errors: path that is not a single non-empty string; path that is a directory; a file that is not valid JSON.

Value

The parsed provenance as a nested named list (JSON objects become named lists, arrays become unnamed lists, strings, numbers and booleans become length-1 vectors), or NULL if the file does not exist. The list has class c("rmbl_provenance", "list"), which only changes how it prints (a box); its elements, and the JSON written from it, are those of a plain list.

Examples

```
provfile <- tempfile(fileext = ".json")
writeLines('{"dataset": {"title": "demo"}, "sha256": "abc"}', provfile)
prov <- loadprov(provfile)
prov$dataset$title
loadprov(file.path(tempdir(), "no-such-file.json")) # NULL

# arrays stay lists: unlist() them for a vector
writeLines('{"schema": {"expected_columns": ["id", "year"]}}', provfile)
p <- loadprov(provfile)
unlist(p$schema$expected_columns)
length(valsch(data.frame(id = 1, year = 2020), p))

# a directory is not a provenance file
try(loadprov(tempdir()))
```

---

loounit *Leave-one-out estimates of a difference in differences*

Description

Re-estimates plndid leaving out one treated unit at a time, to show that no single unit drives the result.

Usage

```
loounit(
  data,
  unit,
  time,
  y,
  trtd,
  covs = NULL,
  cl = NULL,
  units = NULL,
  w = NULL
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`trtd`	Column name of the treatment indicator (0 or 1).
`covs`	Optional column names of time-varying controls.
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.
`units`	Which units to leave out (default every treated unit).
`w`	Optional column name of non-negative weights, such as population or matching weights, for weighted least squares with weighted within-means. Rows of weight 0 take no part in the fit.

Value

A data frame of class `rmbl_loo`: `dropped`, `estimate`, `se`; attribute `full` (the estimate with every unit). `plot()` draws the histogram with the full-sample estimate and a summary.

Examples

```
p <- polex()
l <- loounit(p, "district", "month", "crmrate", "reformed")
range(l$estimate)
plot(l)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
```

```
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# each adopting town left out in turn, weighted by population
loounit(own, "town", "year", "rate", "bylaw", w = "pop")
```

`ltrlrng` *Search theory for search and rescue planning*

Description

The quantities a search and rescue planner works with (Koopman's search theory, as used in the U.S. Coast Guard and IAMSAR planning methods):

- `ltrlrng()` (lateral range): the probability that a sensor passing at distance `x` detects the object, under the inverse cube, definite range or M-Beta model with sweep width `W`;
- `swpwdth()` (sweep width): the area under a sampled lateral range curve;
- `srchpod()` (search POD): the probability of detection from coverage, for random search, a definite-range sweep and an inverse cube parallel sweep;
- `pswppod()` (parallel sweep POD): the POD of a parallel sweep at track spacing `S`, computed for any model, with optional normal navigation error;
- `srchscs()` (search success): the probability of success, the cumulative POD of repeated searches and the containment probabilities updated after an unsuccessful search;
- `datumpmap()` (datum probability map): the probability that each grid cell contains the object, for a point, line or area datum;
- `optcrcsrch()` (optimal circular search): Koopman's optimal effort for a circular normal datum;
- `opteffort()` (optimal effort): the allocation of a fixed effort over cells that maximises the probability of success.

Usage

```
ltrlrng(x, W, model = c("invcube", "definite", "mbeta"), m = 1, navsd = 0)

swpwdth(x, p = NULL)

srchpod(
```

```
  coverage = NULL,
  model = c("random", "definite", "invcube", "mbeta"),
  m = 1,
  W = NULL,
  effort = NULL,
  area = NULL
)

pswppod(
  W,
  S,
  model = c("invcube", "definite", "mbeta"),
  m = 1,
  navsd = 0,
  npts = 2001,
  ntrk = 60
)

srchscs(poc, pod)

datumpmap(
  xedg,
  yedg,
  datum = c("point", "line", "area", "corridor", "general"),
  center = c(0, 0),
  sigma = 1,
  end = NULL,
  nodes = 64,
  wts = NULL
)

optcrcsrch(sigma, W, effort)

opteffort(poc, area = NULL, W, effort, tol = 1e-14)
```

Arguments

x	Numeric vector: lateral ranges (distances from the track), any sign. For `swpwdth()`, strictly increasing abscissae, or an `rmbl_ltrlrng` object (then `p` is taken from it).
W	Positive number: the sweep width, in the distance unit of the grid and the effort (nautical miles, kilometres: one unit throughout). For `srchpod()`, give it with `effort` and `area` in place of `coverage`; default `NULL`.
model	For `ltrlrng()` and `pswppod()`: `"invcube"` (the default), `"definite"` or `"mbeta"`. For `srchpod()`: `"random"` (the default), `"definite"`, `"invcube"` or `"mbeta"`.
m	Number in (0, 1]: the M-Beta detection probability inside its range. Default 1.
navsd	Non-negative number: the standard deviation of the cross-track navigation error, in the units of `x`. Default 0.

p — Numeric vector in [0, 1]: the lateral range curve at x.

coverage — Non-negative numeric vector of coverages (sweep width times track length over the area searched; 1 means the effort equals the area). Default NULL: then W, effort and area give it, W * effort / area. Give one or the other, not both.

effort — Non-negative number: the total effort, as track length (the distance the searchers travel while searching, in the unit of W). For srchpod(), the track length in each area, one number or one per area (a matrix laid out like the map when area is one); default NULL.

area — Positive numeric vector, one per cell: the cell areas, in the square of the unit of W. May be NULL when poc is an rmbl_datummap (its cells' areas are used). For srchpod(), the areas searched, or a map from datumpmap() whose cell areas are taken; default NULL.

S — Positive number: the track spacing.

npts, ntrk — Whole numbers: the quadrature points across one spacing (at least 3, raised to the next odd number; default 2001) and the tracks on each side (at least 1; default 60).

poc — Numeric vector or matrix in [0, 1], summing to at most 1: the probability that each area contains the object (your own areas' shares, for example from a consensus of the planners' scenarios). For srchscs() and opteffort() also a probability map from datumpmap(): its cells are the areas, in the order of as.vector(map$poc).

pod — Numeric vector in [0, 1], one per area, or a matrix with one row per search and one column per area. When poc is a map, also a matrix laid out like the map (one search, cell by cell), a srchpod() result with one value per cell, or an opteffort() result (its cells' PODs).

xedg, yedg — Strictly increasing numeric vectors: the cell edges of the grid laid over your search area, east-west and north-south, in the distance unit (seq(-5, 5, by = 0.5): half-unit cells over ten units). Cells may differ in size.

datum — "point" (the default), "line", "area", "corridor" or "general".

center — Two numbers, x then y in the grid's coordinates: the datum point (for example the last known position), the start of a line datum or a corner of an area datum. Default c(0, 0).

sigma — Positive number: the standard deviation of the circular normal position error, in the distance unit (a total probable error E is 1.1774 sigma). Default 1.

end — Two numbers: the end of a line datum or the opposite corner of an area datum; NULL for a point datum.

nodes — Whole number from 2 to 1000: Gauss-Legendre nodes along a line datum. Default 64.

wts — For datum = "general": a non-negative matrix of cell weights, one row per y cell and one column per x cell.

tol — Positive number: the bisection tolerance on log lambda.

Details

Coverage is `C = W z / A`: sweep width times track length over the area searched (for a parallel sweep, `W / S`). The lateral range curves are: definite range, `p(x) = 1` for `|x| <= W / 2`; M-Beta, `p(x) = m` for `|x| <= W / (2 m)`; inverse cube, `p(x) = 1 - exp(-W^2 / (4 pi x^2))`. Each has area `W`, the sweep width. `swpwdth()` integrates by the trapezoid rule.

With `navsd > 0`, `ltrlrng()` gives the expected curve seen through a normal cross-track navigation error: the convolution of the curve with that normal (Frost 1996, equation 5-14), exact for the rectangular curves and by the trapezoid rule for the inverse cube. Its area is still `W`; the curve is lower and wider.

`srchpod()`: random search `1 - exp(-C)`; definite range `min(C, 1)`; inverse cube parallel sweep `erf(sqrt(pi) C / 2)` (Koopman 1956); M-Beta parallel sweep `C` for `C < M` and `1 - (1 - M)^(C / M)` above it (Frost 1996, equation 5-11), which is exact wherever the tracks overlap a whole number of times.

`pswppod()` integrates over one track spacing (Simpson's rule on `npts` points) the probability that at least one of the tracks `-ntrk, ..., ntrk` detects the object; for the inverse cube model the tracks beyond `ntrk` are added in closed form from `sum_k 1 / (x - k S)^2 = (pi / S)^2 / sin^2(pi x / S)`. With `navsd > 0` the lateral range curve is averaged over a normal cross-track error (40-point Gauss-Hermite quadrature). Without navigation error the inverse cube result equals `srchpod(W / S, "invcube")`.

`srchscs()`: the probability of success is `POS = sum(POC * POD)`; after an unsuccessful search the containment probabilities become `POC (1 - POD) / (1 - POS)` (Bayes). With several searches (`pod` a matrix, one row per search) the update is applied in turn and the cumulative POD is `1 - prod(1 - POD)` per area. When a search is certain to succeed (`POS = 1`) the update is undefined: it is returned as `NA`, with a warning.

`datumpmap()`: a point datum is a circular normal of standard deviation `sigma` about `center`; a line datum the same spread along the segment from `center` to `end` (Gauss-Legendre quadrature on `nodes` points); an area datum a uniform distribution over the rectangle with corners `center` and `end`; a corridor datum normal across the segment from `center` to `end` and uniform along it, with nothing beyond its ends (Frost 1996, figure 3-8; the segment must run along x or along y); a general datum the cell weights `wts` scaled to sum to 1. The share of the probability outside the grid is returned as `outside`.

`optcrcsrch()`: with effort (track length) `L`, the optimal search of a circular normal datum covers a disc of radius `R = (4 sigma^2 W L / pi)^(1/4)`, and succeeds with probability `1 - (1 + u) exp(-u)`, `u = R^2 / (2 sigma^2)` (Koopman 1957).

`opteffort()`: under random search detection the optimal effort in cell `i` is `z_i = (A_i / W) log(r_i / lambda)` where `r_i = POC_i W / A_i` exceeds `lambda`, and 0 elsewhere; the Lagrange multiplier `lambda` is found by bisection on its logarithm so that the efforts add up to `effort` (Stone 1975, chapter 2). It assumes random search detection in every cell (the exponential POD curve, `srchpod(model = "random")`), the conservative choice when the tracks are not laid evenly; a plan searched by parallel sweeps does at least as well.

All distances (`x`, `W`, `S`, `navsd`, the grid edges, `center`, `end`, `sigma`, `effort`) are in one unit of your choice, and areas in its square.

Every input is checked: numbers must be finite and of the right length, sweep widths, spacings and standard deviations positive, probabilities in [0, 1] and containment probabilities summing to at most 1; unknown models and datum types are errors.

Value

`ltrlrng()`: a numeric vector of class `rmbl_ltrlrng` (the detection probabilities, with the inputs as attributes; `as.numeric()` drops them). `srchpod()`: a numeric vector of class `rmbl_srchpod`. `swpwdth()` and `pswppod()`: one number. `srchscs()`: a list of class `rmbl_srchscs` with `pos` (the probability of success), `posarea` (per area), `pocupd` (the updated containment probabilities) and `cumpod` (the cumulative POD per area), and when `poc` is a map, `map`: the map after the unsuccessful search, ready for `opteffort()` and `plot()`. `datumpmap()`: a list of class `rmbl_datummap` with `poc`, a matrix with one row per `y` cell and one column per `x` cell, `outside`, and the inputs. `optcrcsrch()`: a list of class `rmbl_optcrc` with `radius`, `pos` and `cntrdns` (the coverage at the centre). `opteffort()`: a list of class `rmbl_opteffort` with `effort`, `coverage` and `pod` per cell (a matrix when `poc` is one), `pos` and `lambda`. Every class has a `print()` method, and `plot()` draws the lateral range curve (`equiv = TRUE` adds the equivalent definite-range rectangle, `spacing` a second track), the POD by coverage for any of the models (`models`, `m`, `cmax`), the map as an image with contours, a surface banded by height, the surface and its top view side by side (the device's two panels, or the next two of a layout the caller set with `par(mfrow = )`), or its cells with their percentages (`view`), and the allocation (see `plot-methods`). `srchfig` draws every figure of Frost (1996) with them.

References

Koopman, B. O. (1946). Search and Screening. OEG Report 56, Operations Evaluation Group, Office of the Chief of Naval Operations.

Koopman, B. O. (1956). The theory of search. I. Kinematic bases. Operations Research 4(3), 324-346. DOI 10.1287/opre.4.3.324

Koopman, B. O. (1956). The theory of search. II. Target detection. Operations Research 4(5), 503-531. DOI 10.1287/opre.4.5.503

Koopman, B. O. (1957). The theory of search. III. The optimum distribution of searching effort. Operations Research 5(5), 613-626. DOI 10.1287/opre.5.5.613

Frost, J. R. (1996). The Theory of Search: A Simplified Explanation. Soza and Company and U.S. Coast Guard Office of Search and Rescue.

Stone, L. D. (1975). Theory of Optimal Search. Academic Press.

Examples

```
# inverse cube detection 0.5, 1 and 2 sweep widths off the track
ltrlrng(c(0.5, 1, 2), W = 1)
plot(ltrlrng(seq(-3, 3, by = 0.5), W = 2))

# the area under the curve is the sweep width
xs <- seq(-200, 200, by = 0.01)
swpwdth(xs, ltrlrng(xs, W = 1, model = "definite"))

# POD at coverage 1 under the three models
srchpod(1)
srchpod(c(0.5, 1, 2), model = "invcube")
plot(srchpod(c(0.5, 1, 1.5)))
# the book's M-Beta curves against random search and the definite range
plot(
```

```
  srchpod(1, "mbeta", m = 0.5),
  models = c("random", "definite", "mbeta"),
  m = c(0.5, 0.25)
)
# the expected curve under a navigation error as large as half the sweep
# width
plot(ltrlrng(0, W = 1, navsd = 0.5), equiv = TRUE)

# a parallel sweep at spacing 1.3 W, without and with navigation error
pswppod(1, 1.3)
pswppod(1, 1.3, navsd = 0.3, npts = 201)

# three areas searched once; the containment after finding nothing
srchscs(c(0.5, 0.3, 0.2), c(0.8, 0.5, 0))

# a point datum on a 10 x 10 grid, and the best use of 30 units of track
m <- datumpmap(seq(-5, 5), seq(-5, 5), sigma = 2)
plot(m)
plot(m, view = "both")
# the book's line datum: normal across the line, uniform along it
plot(
  datumpmap(0:10, seq(-3, 3, by = 1.2), "corridor", c(0, 0), end = c(10, 0)),
  view = "cells",
  digits = 1
)
a <- opteffort(m, W = 1, effort = 30)
a$pos
plot(a)
optcrcsrch(sigma = 2, W = 1, effort = 30)$pos

# your own search, step by step: a last known position 2 km east and 3 km
# north of the base, known to about 1.5 km, on a grid of 1 km cells
own <- datumpmap(seq(-3, 7), seq(-2, 8), center = c(2, 3), sigma = 1.5)
own
# 0.6 km sweep width; 12 km of track walked in each of the four cells round
# the position, none elsewhere
walked <- matrix(0, nrow(own$poc), ncol(own$poc))
walked[5:6, 5:6] <- 12
pd <- srchpod(W = 0.6, effort = walked, area = own, model = "invcube")
# the chance that search found the hiker, and the map if it did not
day1 <- srchscs(own, pd)
day1
# the best use of tomorrow's 40 km on what is left
day2 <- opteffort(day1$map, W = 0.6, effort = 40)
day2
plot(day1$map)

# inputs that cannot be right are refused
try(ltrlrng(1, W = -1))
try(srchpod(1, W = 0.6))
try(srchscs(c(0.8, 0.8), c(0.5, 0.5)))
```

`mahaoutl` *Multivariate outliers by Mahalanobis distance*

Description

Finds the rows of a numeric table that are unusual as a combination of values, not just on one variable at a time, and gives each row a distance, a p-value and an outlier flag. Use it before modelling to find records that are probably data-entry or linkage errors.

Usage

```
mahaoutl(data, alpha = 0.001, robust = TRUE)
```

Arguments

`data`	A data frame or numeric matrix; non-numeric columns are dropped. Rows with any missing value are skipped, and reported as `NA`.
`alpha`	Significance level for the `outlier` flag (default 0.001, deliberately strict: at 0.05 one row in twenty is flagged by construction).
`robust`	Use the median/MAD centre and scale (default `TRUE`) .

Details

Ranks rows by how far they sit from the centre of the data ONCE THE CORRELATIONS ARE ACCOUNTED FOR, which is what a per-column check cannot do: a row can be unremarkable on every variable separately and still be impossible jointly – a person 1.5 m tall weighing 140 kg is inside both marginal ranges and outside the cloud.

With centre c and scatter matrix S, the squared distance of row x is

$$d^2 = (x - c)^\top S^{-1} (x - c).$$

Under multivariate normality the squared distance is chi-square on p degrees of freedom (the number of numeric columns), which gives the p-value and the cut-off.

`robust = TRUE` centres each column on its median and scales it by its MAD (`cmad()`, the 1.4826-scaled median absolute deviation; a column whose MAD is zero falls back to its standard deviation), then takes S as the sample covariance of the standardised data. `robust = FALSE` uses the column means and the sample covariance. The robust centre and scales resist outliers; the correlation shape in S is still a classical estimate, so this is not a high-breakdown estimator such as the minimum covariance determinant. The choice matters more than it sounds: outliers inflate the very covariance used to judge them, so with several of them the classical distance hides exactly the rows it is meant to find (the masking effect).

Exactly collinear columns are refused rather than repaired: the distance is undefined there, and flooring the covariance's eigenvalues to make it invertible would return numbers governed by the floor rather than by the data. Drop the redundant column first – `topcors()` will identify it.

Errors: `data` neither a data frame nor a matrix; no numeric columns; no more complete rows than columns; collinear columns (smallest eigenvalue of S at most 10^{-8} times the largest); `alpha` outside (0, 1). A progress indicator is shown in an interactive session.

Value

A data frame of class `bricklayer_outliers` with `row`, `distance` (the square root of the squared Mahalanobis distance), `p_value`, `logpvalue` (the log of the chi-square upper tail, which keeps its information where `p_value` underflows to 0) and `outlier` (logical), ordered by descending distance, with skipped rows (`NA` distance) last. `row` is the row number in `data`. The attributes `robust` and `alpha` record the settings. The print method shows the number of flagged rows and the ten most distant.

References

Mahalanobis PC (1936). On the generalised distance in statistics. *Proceedings of the National Institute of Sciences of India* 2(1), 49–55.

See Also

`ctkyfnc()` for the per-column version, `rulemad()` to turn it into a validation rule.

Examples

```
set.seed(1)
n <- 200
df <- data.frame(height = stats::rnorm(n, 170, 10))
df$weight <- df$height * 0.5 + stats::rnorm(n, 0, 5)

# A row that is ordinary on each variable but unlikely jointly:
# short and heavy.
df[1, ] <- list(height = 155, weight = 100)

out <- mahaoutl(df)
head(out, 3)

# It is flagged, although both values lie inside the other rows' ranges.
out$row[1] == 1
range(df$height[-1])
range(df$weight[-1])

# A cluster of bad rows drags the mean toward itself; the median and
# MAD move less, so the robust distance of such a row is the larger.
bad <- df
bad[2:21, ] <- list(height = 150, weight = 110)
dclassical <- mahaoutl(bad, robust = FALSE)
drobust <- mahaoutl(bad, robust = TRUE)
dclassical$distance[dclassical$row == 2] < drobust$distance[drobust$row == 2]

# A duplicated column has no distance defined, and is refused.
dup <- df
dup$height2 <- dup$height
try(mahaoutl(dup))
```

`mcartst` *Little's test for data missing completely at random*

Description

Tests the MCAR assumption: that whether a value is missing is unrelated to any value in the data, observed or not.

The assumption matters because it is what licenses the easy options. Dropping incomplete rows is unbiased under MCAR and biased otherwise; mean imputation understates variance under MCAR and distorts the centre as well otherwise. A small p-value here says the easy options are not available. Run it on the numeric columns of a table before deciding how to handle its missing values.

Usage

```
mcartst(data, maxit = 500L, tol = 1e-07)
```

Arguments

`data`	A data frame or numeric matrix. Non-numeric columns are dropped with a warning, since the statistic is defined on moments.
`maxit`	Whole number, default 500: the maximum number of EM iterations.
`tol`	Number, default 1e-7: EM stops when the largest absolute change in any mean or covariance entry between iterations falls below it.

Details

How it works:

Rows are grouped by their pattern of missingness. If missingness is unrelated to the values, then each pattern's observed-variable means should agree with the overall estimates, up to sampling error. The statistic sums those disagreements,

$$d^2 = \sum_j n_j (\bar{x}_j - \hat{\mu}_j)' \hat{\Sigma}_j^{-1} (\bar{x}_j - \hat{\mu}_j),$$

over the variables observed in pattern j, and is compared with a chi-square distribution on $\sum_j p_j - p$ degrees of freedom.

The overall $\hat{\mu}$ and $\hat{\Sigma}$ are the MAXIMUM-LIKELIHOOD estimates under multivariate normality WITH the missing data, obtained by expectation-maximisation – not the complete-case estimates, which would already embed the bias the test is looking for.

What it cannot do:

A large p-value is NOT evidence that the data are MCAR; it is a failure to detect a departure, and the test has little power on small samples or with many patterns. The test also assumes multivariate normality, so on markedly non-normal columns a rejection may be telling you about the distribution rather than the missingness.

Neither this test nor any other can distinguish missing-at-random from missing-NOT-at-random, because that distinction depends on the values that were never observed. Only knowledge of how the data were collected settles it.

Collinear or constant columns are refused rather than worked around: the likelihood is degenerate there, and the EM step's eigenvalue floor would otherwise return a statistic governed by that floor instead of by the data.

Practicalities:

Columns that are entirely missing and rows with no observed value are dropped before the test. The EM loop is written in R and visits every row on every iteration, so it takes about $n\times$ iterations small matrix solves: seconds for a few thousand rows and a handful of columns, longer for large tables. A progress indicator is shown in an interactive session. If EM has not converged after `maxit` iterations the last estimates are used; `iterations` then equals `maxit`.

Errors: `data` neither a data frame nor a matrix; fewer than two numeric columns, or fewer than two usable rows or columns after the drops; collinear or constant columns (smallest covariance eigenvalue at most 10^{-8} times the largest); EM estimates that cannot be inverted. Non-numeric columns are dropped with a warning.

Value

A list of class `bricklayer_mcar`: `statistic`, `df`, `p_value`, `logpvalue` (the log of `p_value`, finite where `p_value` underflows to 0), `npat` (distinct missingness patterns), `nused` (rows in the patterns that entered the statistic), `nvars` (columns used), `iterations` (EM iterations), `mu` (named ML means), `sigma` (ML covariance matrix), and `note` (a caveat when the test is degenerate or a pattern was skipped, else `NA`). The print method gives the verdict in words. With complete data `df` is 0 and `p_value` is `NA` – there is nothing to test.

References

Little RJA (1988). A test of missing completely at random for multivariate data with missing values. *Journal of the American Statistical Association* 83(404), 1198–1202.
DOI 10.1080/01621459.1988.10478722

Dempster AP, Laird NM, Rubin DB (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society B* 39(1), 1–38.

See Also

`misspat()` for the patterns themselves, `misssmry()` for the rates.

Examples

```
set.seed(1)
n <- 300
x <- stats::rnorm(n)
y <- x + stats::rnorm(n)

# Missing completely at random: a coin flip decides, so the test
# should not reject.
mcar <- data.frame(x = x, y = y)
mcar$y[sample(n, 90)] <- NA
```

```
mcartst(mcar)

# Missing depending on the OTHER, observed variable: not MCAR, and
# detectable, because the pattern's mean of x is shifted.
mar <- data.frame(x = x, y = y)
mar$y[x > 0.4] <- NA
mcartst(mar)

# Complete data has one pattern and nothing to test.
mcartst(data.frame(a = x, b = y))$df

# A duplicated column makes the likelihood degenerate, and is refused.
dup <- mcar
dup$x2 <- dup$x
try(mcartst(dup))

# The EM estimates are the ML ones: with no missingness they are the
# column means and the ML (1/n) covariance.
fit <- mcartst(data.frame(a = x, b = y))
all.equal(fit$mu, c(mean(x), mean(y)), check.attributes = FALSE)
```

`mhcont` *The four directions of the First Nations Mental Wellness Continuum, from published indicators*

Description

Lays out a group's published indicators under the four outcomes of the First Nations Mental Wellness Continuum Framework (hope, belonging, meaning, purpose), each with its interval, and says which directions the tables have no measure of.

Usage

```
mhcont(
  data,
  grp,
  ind,
  est,
  lower,
  upper,
  hope = NULL,
  belong = NULL,
  meaning = NULL,
  purp = NULL,
  qual = NULL,
  conflvl = 0.95
)
```

Arguments

data	A data frame, one row per group and indicator.
grp, ind, est, lower, upper	Column names: the group, the indicator, the estimate (percent) and its published limits.
hope, belong, meaning, purp	The indicators taken as proxies for each direction, or NULL where the tables have none.
qual	Optional column of quality flags: "E" rows are marked use with caution, and suppressed ones ("F", "x", ". .") get no estimate.
conflvl	Confidence level of the published intervals (default 0.95).

Details

The Framework (Health Canada and Assembly of First Nations 2015) describes mental wellness as hope for the future, a sense of belonging and connectedness to family, community and culture, a sense of meaning and of how one's life fits into creation, and purpose in daily life. It has no numeric scoring, and no survey table measures the four directions as the Framework defines them. This function takes the indicators a user names as proxies for each (hope = "Always or often has a hopeful outlook", ...), reports each direction's level (the mean of its proxies, their standard errors combined as independent; the proxies share respondents, so if they are positively correlated that standard error is too small), marks every row as a proxy, and reports a direction given no proxy as missing rather than filling it from another. One indicator named for two directions (Statistics Canada asks about "meaning and purpose" together) is used for both and marked shared.

The result plots as a welprof. For the balance across the directions, call holbal on the rows with a proxy (x[x$domain != "no published proxy",], with "domain" as its domain column): a missing direction otherwise leaves the balance NA, and a shared proxy counts once per direction it serves, so give such directions half weight. Other frameworks of wellness (Te Whare Tapa Wha, the Social and Emotional Wellbeing domains, a Nation's own) take the same route through welprof and holbal, which accept any set of domains.

Value

A data frame with the classes rmbl_wellness_continuum and rmbl_wellness_profile. Its columns are group, indicator (the direction), side ("wellness"), domain (the proxies), estimate, lower, upper, usewithcau, shared, proxy (always TRUE).

References

Health Canada, Assembly of First Nations (2015). First Nations Mental Wellness Continuum Framework. Ottawa: Health Canada. Summary report: sac-isc.gc.ca/eng/1576093687903/1576093725971

See Also

welprof, holbal.

Examples

```
# well-being, 2024 (table 41-10-0082-01): hope, and meaning and purpose
# (asked together); this table has no measure of belonging, so that direction
# is reported as missing
wb <- indiex("wellbeing")
mw <- mhcont(
  wb,
  "group",
  "indicator",
  "estimate",
  "lower",
  "upper",
  hope = "Always or often has a hopeful outlook",
  meaning = "Sense of meaning and purpose rating of 8, 9 or 10",
  purp = "Sense of meaning and purpose rating of 8, 9 or 10",
  qual = "quality"
)
mw
plot(mw)

# your own table: made-up percentages for two communities, with limits;
# one item stands for both meaning and purpose, and none for belonging
own <- data.frame(
  community = rep(c("Lakeside", "Riverbend"), each = 2),
  item = rep(c("Hopeful", "Life has meaning"), 2),
  pct = c(64, 70, 58, 75),
  lo = c(59, 65, 52, 70),
  hi = c(69, 75, 64, 80)
)
mhcont(own, "community", "item", "pct", "lo", "hi", hope = "Hopeful",
       meaning = "Life has meaning", purp = "Life has meaning")
```

---

missgrid *Text map of where the missing values are*

Description

Draws the missingness of a whole table as a grid, one character per cell block – a console counterpart of `visdat::vis_miss()` that needs no graphics device, so it works over SSH, in a log, and inside a capsule's plain-text summary.

Rows are binned so the map fits `height` lines; a block is drawn at the shade its missing proportion falls in. Seeing the table at once is the point: a diagonal band, a block of rows, or a single ragged column are all instantly recognisable shapes that a column of percentages is not.

Usage

```
missgrid(data, height = 20L, width = 12L)
```

Arguments

data	A data frame whose missing cells are drawn as a grid.
height	Whole number, default 20: the maximum number of lines (row blocks) in the map.
width	Whole number, default 12: column names are cut to this many characters in the vertical legend.

Details

The rows are split into at most height consecutive blocks of nearly equal size; each map line is one block, labelled with its first row number, and each map column is one column of data. A cell's shade is its block's proportion missing rounded to the nearest quarter: blank (0), light, medium, dark, full (1). Where the console cannot show Unicode the shades are " .-+#". Column names are written vertically above the map, cut to width characters. A table with no rows prints "(no rows)". Errors: data not a data frame or without columns; height not a positive whole number.

Value

A character vector of the map's lines, invisibly; the map is printed with cat() as a side effect.

See Also

misspat(), missruns()

Examples

```
set.seed(1)
df <- data.frame(
  complete = 1:100,
  block = c(rep(NA, 30), 31:100),
  scattered = ifelse(stats::runif(100) < 0.3, NA, 1),
  mostlygone = c(1:10, rep(NA, 90))
)
missgrid(df)

# a coarser map with short labels
missgrid(df, height = 5, width = 4)

try(missgrid(df, height = 0))
```

---

misspat *Which columns are missing together*

Description

Counts the distinct PATTERNS of missingness across rows – which combination of columns is missing in each row – rather than the per-column rates `profcols()` reports. Use it before deciding how to handle gaps (drop rows, impute, model separately), because the right choice depends on whether columns are missing together.

Usage

```
misspat(data, maxpat = 20L)
```

Arguments

`data`	A data frame whose missingness patterns are counted.
`maxpat`	Maximum patterns to return, most frequent first, one positive whole number (default 20).

Details

The distinction decides what to do about the gaps. Two columns each 30% missing at random need different handling from two columns 30% missing in THE SAME rows – the second is one structural gap (a join that failed, a form section nobody filled in) and often means those rows should be dropped or modelled separately, while the first does not. A per-column rate cannot tell the two apart; this can.

Each row is coded as a string with one character per column, `X` where `is.na()` is true and `.` otherwise, and the codes are tabulated. Patterns are sorted by descending frequency and the first `maxpat` kept, so `pctrows` may not sum to 100 when patterns are cut. A frame with no rows gives an empty table. The work is one pass over the rows; very wide frames give long codes.

Errors: `data` not a data frame, without columns, or with missing or duplicated names; `maxpat` not one positive whole number.

Value

A data frame of class `bricklayer_missingness`, one row per pattern: `pattern` (character, `.` for present and `X` for missing, in column order), `nrows` (integer), `pctrows` (numeric, percent of all rows), `n_missing` (integer, columns missing in that pattern) and `columns` (character, their names comma-separated, `""` for the complete pattern). Carries the column order as the `"columns"` attribute.

See Also

`profcols()` for per-column rates.

Examples

```
# Two columns missing in the SAME rows: one structural gap.
structural <- data.frame(
  id = 1:10,
  a = c(rep(NA, 3), 4:10),
  b = c(rep(NA, 3), 4:10)
```

```
)
misspat(structural)

# The same per-column rates, but missing independently.
scattered <- data.frame(
  id = 1:10,
  a = c(rep(NA, 3), 4:10),
  b = c(1:7, rep(NA, 3))
)
misspat(scattered)

# A complete frame has exactly one pattern.
misspat(data.frame(x = 1:3, y = 4:6))

# keep only the most common pattern
misspat(scattered, maxpat = 1)

# a frame without columns has no patterns
try(misspat(data.frame()))
```

`missruns` *Runs of consecutive missing values*

Description

Finds the maximal stretches of consecutive `NA` in each column, with where each begins and how long it is.

Row order carries meaning in a capsule far more often than people allow for – a time series, an ordered export, a paginated download – and a long unbroken run of missingness means something different from the same count scattered about. A run says an instrument was down, a page failed to fetch, or a period was never collected; scattered gaps say individual records failed. The rate cannot distinguish them.

Usage

```
missruns(data, minrun = 2L)
```

Arguments

`data`	A data frame, in the row order that carries meaning.
`minrun`	Whole number, default 2: report only runs at least this long (a run of 1 is an isolated gap).

Details

Each column's missingness indicator `is.na(x)` is run-length encoded with `rle()`; every run of `TRUE` with length at least `minrun` is reported. Rows are taken in their current order, so sort the data by time (or by download order) first. A list column counts an element as missing only when `is.na()` says so. Errors: `data` not a data frame or without columns, or with duplicated or empty names; `minrun` not a positive whole number.

Value

A data frame of class `bricklayer_runs` with `column` (character), `start` and `end` (integer row numbers, inclusive) and `length` (integer), longest first and then by column name. Zero rows when there are no qualifying runs.

See Also

`misspat()` for which columns are missing together, `misssmry()` for the rates.

Examples

```
# One long outage and two isolated gaps, with the same total count.
df <- data.frame(
  outage = c(1, 2, NA, NA, NA, NA, 7, 8),
  scattered = c(1, NA, 3, 4, NA, 6, NA, NA)
)
sum(is.na(df$outage)) == sum(is.na(df$scattered))

missruns(df)

# Isolated gaps too, by lowering the threshold.
missruns(df, minrun = 1)

# A complete column has no runs.
missruns(data.frame(x = 1:5))

try(missruns(df, minrun = 0))
```

---

misssmry *Missingness in one line per question*

Description

The scalar summaries of missingness: how much of the table is missing, how many rows are complete, and how many columns have any or only missing values. Use it as a first look at a new extract, or to report data completeness in a methods section; `misspat()` then shows which columns are missing together.

Usage

```
misssmry(data)
```

Arguments

data	A data frame with at least one column.

Details

A cell is missing when `is.na()` is `TRUE` (so `NaN` counts, and an empty string does not). A row is complete when no cell in it is missing (`stats::complete.cases()`). Percentages are of all cells and of all rows. A table with no rows gives zeros. Columns are taken by position, so duplicate or empty column names are fine. Errors when `data` is not a data frame or has no columns.

Value

A named numeric vector: `nrows`, `n_cols`, `n_missing` (missing cells), `pct_missing` (percent of cells), `n_complete_rows`, `pct_complete_rows` (percent of rows), `n_cols_any_missing` (columns with at least one missing value) and `n_cols_all_missing` (columns with no value at all).

See Also

`misspat()` for which columns are missing together, `profcols()` for per-column rates.

Examples

```
df <- data.frame(a = c(1, NA, 3), b = c(NA, NA, 3), c = 1:3)
misssmry(df)

# A complete table is all zeros but for its dimensions.
misssmry(data.frame(x = 1:3, y = 4:6))

# a wholly empty column
ms <- misssmry(data.frame(x = 1:4, note = NA))
ms[c("pct_missing", "n_cols_all_missing")]

# a vector is not a table
try(misssmry(c(1, NA, 3)))
```

---

mkbndl *Build a Reproducibility Bundle*

Description

Packs a project folder into one zip that anyone can unzip and run with a double-click: '`START_HERE.command`' on macOS, '`START_HERE.bat`' on Windows, '`start_here.sh`' on Linux. The launchers find R, the runner ('`setup_and_run.R`') checks the packages the project lists, finds or downloads the data (checking it against the pinned SHA-256), or generates synthetic data from the project's recipe, runs the project's analysis script and summarises its '`manifest.json`' in plain language.

Usage

```
mkbndl(
  project,
  dest,
  data = NULL,
  version = "1",
  overwrite = FALSE,
  secrets = FALSE
)
```

Arguments

project	A project folder, or the name of an example shipped with the package ("otis-mrp", the Ontario restrictive confinement analysis).
dest	An existing folder to write the zip into.
data	Optional path to the data file, copied into the bundle (under the file name the provenance pins) so it runs offline. NULL (default) leaves it out: the runner asks for it, offers to download it, or uses synthetic data.
version	Version label in the zip's name, default "1".
overwrite	Replace a zip of the same name, default FALSE.
secrets	Pack files whose names look like credentials (see Details), default FALSE: they are left out with a warning.

Details

A project is a folder holding at least 'config.json' and the analysis script it names (default 'analysis.R'); mkbndlproj() writes a working starter. 'data_provenance.json' pins the data file (name, URL, SHA-256, licence) and may carry a schema$synthetic_recipe for the offline synthetic route. Every other file in the folder is copied into the bundle too, except hidden files, 'results_*' folders and zip files.

Symbolic links are not followed: a link to a file inside the project is packed as that file, and a link to a folder, or to anything outside the project, is left out with a warning that names it. Files whose names look like credentials are left out too, with a warning that lists them, unless secrets = TRUE: 'llm.json', 'credentials.json', 'secrets.yaml', 'token.txt', 'api_key*', 'id_rsa' and the other SSH private keys, '.env' and '*.env', '.netrc', '.Renviron', '.pgpass', 'kubeconfig' and 'kube.config', key stores ('.p12', '.pfx', '.jks', '.ppk'), '.pem' or '.key' files holding a private key, and any other file (not a data table such as '.csv' or '.rds', nor a document such as '.md', '.html', '.Rmd' or '.qmd') whose name, split into words at '-', '_', '.' and spaces, has the word key, keys, apikey, token, tokens, secret, secrets, credential, credentials, creds, password, passwords, passwd, pat or adc ('openai_key.txt', 'github_pat.txt', 'access_token', 'aws_credentials', 'gcloud_adc.json', 'keys.R'; not 'keyboard.csv'). Public keys ('.pub') are packed.

The runner calls the analysis script as Rscript analysis.R <data file> <results folder>, with BRICKLAYER_SYNTHETIC=1 set on synthetic data. The script writes 'manifest.json' into the results folder: mkmnf(), recrd() and wrmnfjs() build it, and they are in the bundle (the script sources them from its own folder, as the starter does), so the reviewer needs only R and the packages listed in r_packages. To call other functions of this package from the analysis, list "rmoriebricklayer" in r_packages.

Optional keys of 'config.json': project (name, title, paper_title, work_type, author, contact, orcid, affiliation, supervisor, licence), analysis$r_script, r_packages, env_packages (packages needed only when an environment variable is set), optional_stages (opt-in stages with their own packages), synthetic_env (another variable that turns on the synthetic route), data$default_candidates, network$endpoints, summary$what_was_done and expected (total_checks, pass_real).

Before zipping, every internal name the bundled R files call is checked to be defined by a bundled file, and by one the runner itself sources when the runner calls it: a bundle that would stop with "could not find function" on the reviewer's machine is refused here instead.

Nothing is written except the zip in dest; the staging folder is a temporary directory removed on exit. The zip is written under a temporary name in dest and renamed when it is complete, so an interrupted build leaves no partial zip (and keeps the one it was to replace). Send the SHA-256 that is printed to recipients separately, so they can check the zip was not altered.

Errors: project that is neither a folder nor a bundled example; a project without 'config.json' or without its analysis script; a project file whose name a bundle file already uses; dest that is not an existing folder; a zip that exists when overwrite is FALSE; data that is not an existing file; a version, or a project$name in 'config.json', that is not a plain name (letters, digits, dot, dash and underscore, not starting with a dot, no .., at most 64 characters); a bundle of more than 65535 files and folders or over 4 GB (more than a plain zip holds).

Value

An object of class rmbl_bundle: a list with path (the zip), sha256, bytes, files (the paths inside the zip), project (the project name) and data (whether the data file is inside). It prints as a box.

See Also

mkbndlproj() for a starter project; mkmnf() and recrd() for the manifest the analysis writes.

Examples

```
# a starter project, bundled
proj <- file.path(tempdir(), "demo-project")
mkbndlproj(proj, title = "Demo analysis", author = "A. Researcher")
b <- mkbndl(proj, dest = tempdir(), overwrite = TRUE)
b
head(b$files)

# the example shipped with the package
otis <- mkbndl("otis-mrp", dest = tempdir(), overwrite = TRUE)
otis$sha256

# an existing zip is kept unless overwrite = TRUE
try(mkbndl(proj, dest = tempdir()))
```

mkbndlchk *Check a reproducibility bundle before shipping it*

Description

Unzips a bundle built by mkbndl() into a new folder, runs it the way a recipient would (Rscript setup_and_run.R --quick) and reads the 'manifest.json' the run writes. The bundle passes when the run finishes, writes its manifest, no cross-check is DIFFER and, on real data, the number of checks and of passes is the one the expected block of its 'config.json' gives (pass_csv for a CSV input, pass_real for the '.RData'). A run on synthetic data marks every check INFO, so it is not compared with expected. A launcher ('start_here.sh', 'START_HERE.command') the zip does not mark executable is reported as a warning: it still runs as bash start_here.sh.

Usage

```
mkbndlchk(
  bundle,
  data = NULL,
  synthetic = FALSE,
  dir = NULL,
  keep = FALSE,
  container = NULL
)
```

Arguments

bundle	Path to the bundle zip.
data	Optional path to the data file, passed to the run as --data. NULL (default): the bundle finds the file, downloads it or, with synthetic = TRUE, generates data.
synthetic	Run on synthetic data (--synthetic), default FALSE; needs no data file and no network.
dir	A folder to unzip into, which must not exist yet. Default: a new folder under tempdir().
keep	Keep that folder after the check, default FALSE.
container	Run the bundle in a container: NULL or FALSE (default) do not, TRUE uses podman, else docker, and "podman" or "docker" names the runtime. See Details.

Details

Warning: this runs the bundle's code on your machine, with your user's rights: its 'setup_and_run.R', its analysis script and anything they start. Check only bundles you built or whose author you trust. To limit what a hostile bundle can reach, the run gets a minimal environment: PATH, the locale variables (LANG, LANGUAGE, LC_ALL and the LC_ categories such as LC_CTYPE, no other LC_ name; TZ), the library paths of this session (R_LIBS), the proxy variables (http_proxy, https_proxy, no_proxy) when they hold no user name or password, a fresh temporary HOME and TMPDIR, and no other variable, so API keys, tokens and passwords in the environment (or in '~/.Renviron') do

not reach it. On Windows the variables that look like credentials are removed instead (a best effort): names holding KEY, TOKEN, SECRET, PASS, CREDENTIAL, AUTH, COOKIE or SESSION, a service's prefix (GITHUB_, AWS_, ...), connection strings (DATABASE_URL, *_DATABASE_URL, *_WEBHOOK_URL, SENTRY_DSN, *_DSN), *_PWD, *_SK, other LC_ names, and any value that is a URL with a password in it. This is not a sandbox: the files your user can read stay readable.

The run is stopped after `getOption("rmbl.bundle.timeout")` seconds (default 1800, 30 minutes; the exit status is then 124). How the processes it starts are contained (the `isolation` of the result, and a row of the checks):

container with `container = TRUE` (podman, else docker) or a runtime named: the run in a container, removed afterwards, with the unzipped bundle mounted read-only and copied into a writable folder, no network unless the real data is to be downloaded (no `data`, not synthetic), 4 GB of memory, 2 CPUs, 512 processes, a read-only root file system, no capabilities and your user id. The image is `getOption("rmbl.bundle.image")`, default R 4.6.1 from the Rocker project pinned by digest; it holds base R only, so a bundle that lists packages in `r_packages` (or downloads its data, which needs rmoriebricklayer) needs an image that has them. Without the runtime the check falls back to the next mode and says so.

pid namespace Linux, when `unshare` can make an unprivileged user and PID namespace: every process the run started, also one that started its own session or forked twice, is killed when the run ends or times out.

job object Windows: the run is started inside a job object that kills every process in it when the run ends or times out.

process group Linux without user namespaces, and macOS: the run leads a process group, killed when it ends, times out or is interrupted. A process can leave the group (setsid, a double fork) and clear its environment. On Linux this R session is made the run's child subreaper, so such a process is re-parented to it rather than to init: every process the run left behind is found by the process tree, killed by its PID and fails the check (the isolation row is an error). Where that cannot be done (macOS, or a Linux kernel older than 3.4) escaped processes cannot be detected, and the isolation row fails: use `container = TRUE` there.

Only a '`results_*`' folder the run itself created counts: one the zip already carries (which `mkbndl()` never packs) is not read, and fails the check. A run that writes no manifest, or whose runner reports "NOT completed", fails.

Errors: `bundle` that is not an existing '`.zip`' file, or not a zip this function reads (more than 65535 entries, an entry name with a NUL byte, one too long, absolute or leaving the folder, or an entry given twice); a zip without '`setup_and_run.R`'; `data` that is not an existing file; `dir` that already exists.

Value

An object of class `rmbl_bundle_audit`: a list with `ok`, `checks` (a data frame of `check`, `ok`, `detail` and `severity`), `mode` (the input the run used, from its manifest: `synthetic` when it is marked synthetic, else its `meta$input_mode` when it records one, such as `csv`, else `real`), `counts` (`total`, `pass`, `differ`, `info`), `log` (the last 30 lines the run printed), `isolation` (how the run was contained: see Details), `bundle` and `dir` (the unzipped bundle, or for a container run the folder it wrote into; `NA` once removed). It prints as a box.

See Also

`mkbndl()` builds the bundle; `mkbndlproj()` starts a project.

Examples

```
proj <- file.path(tempdir(), "audit-demo")
mkbndlproj(proj, title = "Demo analysis")
b <- mkbndl(proj, dest = tempdir(), overwrite = TRUE)
a <- mkbndlchk(b$path, synthetic = TRUE)
a
a$counts
```

---

mkbndlproj *Start a Reproducibility Bundle Project*

Description

Writes a starter project that `mkbndl()` packs as it is: a '`config.json`' with the project's details, a '`data_provenance.json`' to pin the data file and describe synthetic data, and an '`analysis.R`' that reads the data, records its checks with `recrd()` and writes '`manifest.json`'. Edit the three files for the real project; the starter runs from the first build, on synthetic data, so every step can be tried before the analysis is written.

Usage

```
mkbndlproj(
  dir,
  name = basename(dir),
  title = name,
  author = "",
  contact = "",
  overwrite = FALSE
)
```

Arguments

`dir`	Folder for the project, created if needed.
`name`	Short project name used in the zip's name, default the folder's name.
`title`	Title shown in the bundle's pages, default `name`.
`author`, `contact`	Who the reviewer contacts; default empty.
`overwrite`	Replace starter files that exist, default `FALSE`.

Details

The starter's data is a CSV with columns `group` and `value`; its one check is the row count, so the first run on synthetic data shows one INFO. Set `resource$filename`, `resource$direct_url` and `resource$sha256` in 'data_provenance.json' (see `sha256file()`) and the runner finds, downloads and checks the real file.

Errors: `dir` holding any of the three files when `overwrite` is `FALSE`; a `name` with characters other than letters, digits, dot, dash and underscore, one that starts with a dot or holds `..`, or one longer than 64 characters.

Value

An object of class `rmbl_bundle_project`: a list with `dir` and `files` (the files written). It prints as a box.

See Also

`mkbndl()` to build the bundle.

Examples

```
p <- mkbndlproj(file.path(tempdir(), "my-study"), title = "My study")
p
readLines(file.path(p$dir, "analysis.R"), n = 5)

# files are kept unless overwrite = TRUE
try(mkbndlproj(p$dir))
```

`mkmnf` *Construct a Reproducibility Manifest*

Description

Creates an empty manifest: the record of one analysis run that collects the run's metadata, the cross-checks of its results against expected values (added with `recrd()`), and optionally a snapshot of the R environment it ran in. Write it out with `wrmnfjs()` and `wrsmrytxt()`, or hash and sign it with `mnfdgst()` and `cpsatst()`, so a reader can later see what was run, where, and whether the numbers matched.

Usage

```
mkmnf(meta, env = TRUE)
```

Arguments

meta	A list of run metadata, normally named (for example project, author, run_at, synthetic). May be list().
env	Logical scalar; when TRUE (the default) the manifest also records the analysis environment via capenv(): R version, platform, OS, UTC timestamp, random number generator kind and whether it was seeded, and the version of every loaded package. Any other value leaves it out.

Details

The manifest is a plain list, so you can add your own elements to meta before writing it. The package owns the synthetic-data flag: meta$synthetic is set to TRUE when you pass it, when the environment variable BRICKLAYER_SYNTHETIC is set to a value other than 0, false, no or off, or when the package generated synthetic data earlier in the session. It is written only when TRUE, so a manifest of real data keeps the digest it had before the flag existed. On a synthetic manifest every recrd() entry is INFO.

The fields of meta that wrsmrytxt() reads are project, author, run_at, os, r_version and, as a named list or vector of versions, r_package_versions.

Errors when meta is not a list. Nothing is written.

Value

A list with elements meta (the list given, plus synthetic = TRUE when flagged), results (an empty list, filled by recrd()) and, when environment = TRUE, environment (the list from capenv()). The list has class c("rmbl_manifest", "list"), which only changes how it prints (a box); its elements, and the JSON written from it, are those of a plain list.

See Also

recrd(), wrmnfjs(), wrsmrytxt(), mnfdgst().

Examples

```
# Minimal manifest, no environment capture.
man <- mkmnf(list(project = "demo-study", author = "A. Author"),
  env = FALSE
)
names(man) # "meta" "results"
man$meta$project

# With environment = TRUE it also records R version / platform / packages.
full <- mkmnf(list(project = "demo"), env = TRUE)
names(full) # adds "environment"
full$environment$r_version

# a run on generated data is flagged, and its checks become INFO
syn <- mkmnf(list(project = "demo", synthetic = TRUE), env = FALSE)
syn$meta$synthetic
```

```
# meta must be a list
try(mkmnf("demo"))
```

`mkset` *Combine Results into One Report*

Description

Puts several results together: printed, a box of contents and then every result's own box, all at the console's width; written to a file, one self-contained HTML page (each result's box and, when it has one, its plot), Markdown ready for a GitHub comment or a paper, or plain text.

Usage

```
mkset(..., title = "Analysis results", file = NULL, plots = TRUE)
```

Arguments

`...`	The results. A name gives a result its heading (`"Change by year" = anltab(...)`); an unnamed one is headed by its box's own title. One unnamed plain list is read as the list of results.
`title`	The report's heading, default `"Analysis results"`.
`file`	`NULL` (default) to print, or a file to write: the extension picks the format (`.html` or `.htm`, `.md`, `.txt`). An existing file is replaced.
`plots`	In HTML, draw each result's plot, default `TRUE`.

Details

Any object this package returns can go in, and any other R object that prints (a model, a data frame): each is shown as it prints. A year-over-year table from `yoy()` becomes a real table in HTML and Markdown.

In a file the boxes are drawn 80 characters wide and without colour, so the report looks the same whatever console made it. HTML draws each result's `plot()` under its box (`plots = FALSE` leaves them out); a plot that fails is left out and the box is kept. The page needs no network and nothing installed to open.

Errors: no results; a `file` that does not end in `.html`, `.htm`, `.md` or `.txt`; a `title` that is not one string; `plots` that is not `TRUE` or `FALSE`.

Value

An object of class `rmbl_set`: a list with `title` and `items` (the results, named by their headings). It prints as the report. With `file`, it is returned invisibly with the written file's path in attribute `"path"`.

See Also

rptanl() for the report of one anltab() analysis; rmblapp(), whose dashboard gathers results the same way.

Examples

```
x <- c(3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)
d <- data.frame(year = rep(2020:2023, 2), grp = rep(c("a", "b"), each = 4),
                n = c(5, 7, 9, 11, 4, 6, 8, 7), pop = 1000)
s <- mkset(
  "How unequal" = ginicoef(x),
  "Rates by group" = rtcalc(d, "n", "pop", by = "grp"),
  title = "My study"
)
s

# one page, one Markdown file, one text file
mkset(s$items, title = "My study", file = file.path(tempdir(), "r.html"))
mkset(s$items, title = "My study", file = file.path(tempdir(), "r.md"))
mkset(s$items, title = "My study", file = file.path(tempdir(), "r.txt"))

# the extension picks the format
try(mkset(ginicoef(x), file = file.path(tempdir(), "r.pdf")))
```

---

mksyncol *Generate One Synthetic Column From a Spec*

Description

Builds one column of synthetic (made-up) data from a short specification, such as "sample these categories with these weights" or "Poisson counts with mean 25". It is the building block of mksyncsv, which assembles a whole table from a recipe. Use synthetic columns to develop and test an analysis pipeline before, or instead of, touching restricted data.

Usage

```
mksyncol(spec, n, ctx = list(), basep = NULL)
```

Arguments

spec	A list describing the column: type and the fields listed in Details. Lists as parsed from JSON (for example values = list("a", "b")) are accepted.
n	Whole number: how many values to generate.
ctx	Named list of already-generated columns. Later columns can then reference earlier ones, as id_pattern does with a year_col. Defaults to an empty list.
basep	Optional numeric vector of per-row latent propensities in $[0, 1]$, length n, used by the "bernoulli" type to add row-level variation. Default NULL.

Details

The supported `spec$types` and the fields each reads (defaults in parentheses):

`"sample"` **(the default type)** `values`, a vector or list of categories; `weights` (default equal), their probabilities. Drawn with replacement.

`"bernoulli"` `p` (default 0.1), the probability of the first label; `labels` (default `c("Yes", "No")`); `p_with_baserate` (default 0), added as `p_with_baserate * basep` to give row-level variation when `basep` is supplied. A per-row probability above 1 behaves as 1.

`"poisson"` `lambda` (default 1); `min` (default 0), a floor: draws below it are raised to it (so the result is not Poisson below the floor).

`"id_pattern"` `pattern` (default `"id-\{seq:05d\}"`), in which `{seq:05d}` becomes a five-digit zero-padded sequence number and `{year}` the year; `year_col`, the name of a column in `ctx` whose values supply `{year}` and restart the sequence within each year. With `year_col`, that column must have `n` values. No other tokens are recognised.

`"sequence"` `from` (default 1): `from, from + 1, ....`

`"fitted"` `fit`, a `dstfit` result, and `family` (default the first-ranked converged fit): draws from that fitted distribution.

Random draws use R's random number generator, so call `set.seed()` first for a reproducible column. Errors: `spec` not a list; an unknown `type`; errors from `sample()` or `rpois()` on invalid weights or rates.

Value

A vector for the requested column type: character for `"sample"` of strings, `"bernoulli"` and `"id_pattern"`, numeric or integer for `"poisson"` and `"sequence"` (and for `"sample"` of numbers). Its length is `n`, except for `"id_pattern"` with a `year_col`.

See Also

`mksyncsv`.

Examples

```
set.seed(1)
# "sample": categorical draw, optionally weighted.
mksyncol(list(
  type = "sample", values = list("a", "b"),
  weights = list(0.7, 0.3)
), 5)

# "bernoulli": two-label draw at probability p.
mksyncol(list(
  type = "bernoulli", p = 0.5,
  labels = list("Yes", "No")
), 5)

# "poisson": counts with a floor via min.
mksyncol(list(type = "poisson", lambda = 3, min = 1), 5)
```

```
# "id_pattern": templated IDs (the {seq:05d} token is zero-padded).
mksyncol(list(
  type = "id_pattern",
  pattern = "case-{seq:05d}"
), 3)

# ... numbered within each year of an earlier column
yrs <- c(2024, 2024, 2025)
unname(mksyncol(list(type = "id_pattern",
  pattern = "{year}-{seq:05d}", year_col = "year"), 3,
  ctx = list(year = yrs)))

# "fitted": draws from a distribution fitted by dstfit()
f <- dstfit(stats::rexp(100, 2), fams = "exponential", nboot = 0)
mksyncol(list(type = "fitted", fit = f), 4)

# "sequence": a running integer sequence from from.
mksyncol(list(type = "sequence", from = 100), 4)

# An unknown type errors.
try(mksyncol(list(type = "nope"), 3))
```

`mksyncsv` *Generate a Synthetic CSV From a Schema Recipe*

Description

Generates a reproducible synthetic data set from a recipe (the `schema$synthetic_recipe` block of a provenance object, or a list written by hand) and writes it to a CSV file. Each column is described as in `mksyncol`. Next to the CSV it writes a `.synthetic` marker file, so the generated rows can never be mistaken for real data later. Use it to share a runnable version of an analysis without sharing restricted records.

Usage

```
mksyncsv(schema, outpath, nrows = NULL, seed = NULL)
```

Arguments

`schema`	The synthetic recipe: a list with `columns` (a named list of column specs), and optional `nrows`, `seed` and `row_replication`.
`outpath`	Character string: the path of the CSV to write.
`nrows`	Whole number: the number of rows (persons, if replicating) to generate, from 1 to 100 million (the table is built in memory). Defaults to `schema$nrows`, then 50000.
`seed`	Random seed for reproducibility. Defaults to `schema$seed`, then 91735246.

Details

Columns are produced in declaration order so later columns can reference earlier ones. A shared per-person latent propensity, uniform on $[0, 1]$, drives every `"bernoulli"` column through the column's `p_with_baserate` field, so those columns are correlated with each other. An optional `row_replication` block (`values` and `weights`) draws how many rows each person gets: person-level columns are drawn once per person and repeated, while `"bernoulli"` columns are drawn once per row. With replication the file therefore has more than `nrows` rows.

The data are drawn under `seed` and the caller's random number stream is restored afterwards, so the same recipe and seed give the same file and the surrounding analysis is not disturbed.

Side effects: writes `outpath` (overwriting it) and `paste0(outpath, ".synthetic")`, a small JSON file with `synthetic: true`, the row count, the seed, the recipe's SHA-256 and the time; and marks the R session as having made synthetic data, so a manifest built later in the same session is labelled synthetic. A progress bar is shown in an interactive session.

Errors: `schema` not a list; any error from `mksyncol`; an unwritable `outpath`.

Value

Invisibly, a list of class `bricklayer_synthetic` with `path`, `sidecar` (the `<path>.synthetic` marker file; `vrfcps` treats its presence as a failed check), `rows` (rows written), `seed` (the seed used) and `recipe_sha256` (the recipe's digest, `NA` when it could not be computed). Its print method labels the file as synthetic.

See Also

`mksyncol`, `vrfcps`.

Examples

```
recipe <- list(
  n_rows = 20, seed = 42,
  columns = list(
    year   = list(type = "sample", values = list(2024, 2025)),
    alert  = list(type = "bernoulli", p = 0.2),
    visits = list(type = "poisson", lambda = 3, min = 1),
    id     = list(type = "id_pattern", pattern = "p-{seq:05d}")
  )
)
out <- tempfile(fileext = ".csv")
res <- mksyncsv(recipe, out)
res$rows # 20
res$seed # 42 (reproducible)

# The written CSV round-trips and has the declared columns.
df <- utils::read.csv(out)
dim(df)
names(df)

# and the marker file says it is synthetic
file.exists(res$sidecar)
```

```
# nrows overrides the recipe's own row count.
mksyncsv(recipe, tempfile(fileext = ".csv"), nrows = 5)$rows

# one to three rows per person
recipe$row_replication <- list(values = list(1, 2, 3))
mksyncsv(recipe, tempfile(fileext = ".csv"), nrows = 5)$rows

try(mksyncsv("not a recipe", tempfile()))
```

`mnfcanon` *The canonical serialisation of a manifest, and its digest*

Description

`mnfcanon()` renders a manifest as one line of JSON with every object's keys in sorted order and every number at full double precision. `mnfdgst()` is the SHA-256 of those bytes.

Usage

```
mnfcanon(man)

mnfdgst(man)
```

Arguments

`man`	A manifest (a list), as from `mkmnf()`; any list is accepted.

Details

Why this is needed. Two manifests that record the same thing can easily differ as bytes: R lists keep insertion order, so building `meta` before `results` or the other way round gives different JSON, and a signature over the JSON would then depend on the order a script happened to assemble the list. Sorting the keys removes that. The precision matters for a different reason: the default JSON writer emits four significant digits, which is right for a human-readable report and wrong for a record something will later be checked against, because `1/3` comes back as `0.3333` and no recomputation can match it.

Sign `mnfdgst()`, not the pretty JSON. The digest is stable across the assembly order, across `pretty`, and across a round trip through a file.

Full precision means full precision on every platform, which took more than writing enough digits. Seventeen significant digits recover any double exactly, but only through a reader that converts decimal to binary with correct rounding, and not every C library does – macOS arm64 reads the correct decimal for the largest double as infinity. So this package converts decimals itself rather than asking the platform, in integer arithmetic with a remainder that decides the rounding. A manifest written on one machine reads back bit-identically on another, and a caller does nothing to get that.

Errors when `man` is not a list.

Value

`mnfcanon()`: a character scalar, the one-line JSON. `mnfdgst()`: a character scalar of 64 lower-case hexadecimal characters, the SHA-256 of the canonical bytes.

See Also

`mkmnf()`, `wrmnfjs()`, `cpsatst()`.

Examples

```
a <- mkmnf(list(b = 2, a = 1), env = FALSE)
b <- mkmnf(list(a = 1, b = 2), env = FALSE)
# the same content in a different order has the same digest
identical(mnfdgst(a), mnfdgst(b))

# full precision, so a recorded number can be checked later
m <- mkmnf(list(x = 1 / 3), env = FALSE)
grepl("0.33333333333333331", mnfcanon(m), fixed = TRUE)

# any change to a recorded value changes the digest
m2 <- recrd(m, "check", obs = 1, expd = 1)
mnfdgst(m) == mnfdgst(m2)

# only lists can be serialised this way
try(mnfdgst("not a manifest"))
```

---

mnfdiff *Compare the resources of two manifests*

Description

Lists the resources (data files) of two manifests side by side and says which were added, removed or changed: a different SHA-256, or a different size where both sizes are recorded. A manifest is a provenance record (its `resource`), a capsule bundle (its `files`), a data frame with the columns `path` and `sha256` (and optionally `size_bytes`), or the path of a JSON file holding one of the first two. `cpsrfrsh` and `rmbl drift` report what a fetched source changed with it.

Usage

```
mnfdiff(old, new)

## S3 method for class 'rmbl_mnfdiff'
format(x, ...)

## S3 method for class 'rmbl_mnfdiff'
print(x, ...)
```

Arguments

old, new	The two manifests, as described above. A list with neither resource nor files has no resources.
x	An rmbl_mnfdiff result.
...	Further arguments, ignored when printing the comparison.

Value

A data frame of class rmbl_mnfdiff, one row per resource in either manifest, with the columns resource (the file name), change ("added", "removed", "changed" or "unchanged"), old_sha256, new_sha256, old_size and new_size (NA where not recorded). It prints the counts and one line per change. Errors: a manifest that is not a list, a data frame or a file; a data frame without path and sha256.

See Also

cpsrfrsh, cpsbndl.

Examples

```
a <- data.frame(
  path = c("a.csv", "b.csv"),
  sha256 = c("01", "02"),
  size_bytes = c(10, 20)
)
b <- data.frame(
  path = c("a.csv", "c.csv"),
  sha256 = c("09", "03"),
  size_bytes = c(12, 30)
)
d <- mnfdiff(a, b)
d
d$change

# a provenance record and its re-pinned copy
p <- list(
  resource = list(filename = "a.csv", sha256 = "01", size_bytes = 10)
)
q <- p
q$resource$sha256 <- "09"
mnfdiff(p, q)$change

try(mnfdiff(1, b))
```

`mnfrcmp` *Recompute a manifest's recorded statistics against the data*

Description

Re-evaluates named statistics against `data` and compares each to what the manifest recorded. This is the check `vrfcps()` cannot make: that one confirms the record is internally consistent, which an edited-at-the-time manifest also is. Use it when receiving a capsule, or after a data refresh, to confirm that the numbers in the record still come out of the data.

Usage

```
mnfrcmp(man, data, stats, tol = NULL)
```

Arguments

`man`	A manifest, as from `mkmnf()` and `recrd()`.
`data`	The data the statistics are computed from (usually a data frame); passed unchanged to each function.
`stats`	Named list of functions of `data`, each returning one value. Names must match the recorded result names.
`tol`	Optional numeric tolerance (absolute) overriding each result's own recorded `tol`. Supply it to check more strictly than the original run did. `NULL` (the default) uses the recorded one.

Details

Every recorded result that was NOT recomputed is reported too, as `unchecked`. An analysis that recorded twenty statistics and re-derives three has seventeen it has not re-derived, and a report that quietly omitted them would read as a clean bill of health.

For each function in `stats`, its value on `data` is compared with the `observed` value recorded under the same name. Numeric results match when they are identical (so the same `NA`, `NaN` or infinite value on both sides matches) or when $|recorded - recomputed| \leq tol$, with `tol` the argument if given, else the result's recorded `tol`, else 0 (exact); any other pair with a missing or non-finite value is `"DIFFER"`. A non-numeric result (a label, a verdict) is compared as text and must be identical. A function that errors gives the status `"ERROR"` with the message; a name the manifest does not have gives `"NOT_RECORDED"`. `ok` is `TRUE` only when every row is `"MATCH"` and nothing recorded was left unchecked. The functions are run as given, so they can read files or draw random numbers; nothing else is changed.

Errors: `man` not a list with `results`; `stats` not a non-empty fully named list, or with an element that is not a function.

Value

A list of class `c("bricklayer_recompute", "list")`:

`ok` logical, every statistic matched and none was left unchecked.

`results` data frame, one row per recomputed statistic, with the columns `name` (character), `recorded` and `recomputed` (numeric, `NA` for non-numeric values), `delta` (numeric absolute difference), `tol` (numeric), `status` (`"MATCH"`, `"DIFFER"`, `"NOT_RECORDED"` or `"ERROR"`) and `detail` (character explanation).

`unchecked` character, the names recorded but not recomputed.

See Also

`vrfcps()`, `cpsfals()`, `recrd()`.

Examples

```
d <- data.frame(x = 1:10)
m <- mkmnf(list(dataset = "demo"), env = FALSE)
m <- recrd(m, "mean_x", obs = mean(d$x), expd = 5.5)
m <- recrd(m, "n", obs = nrow(d), expd = 10)

# recomputing both reproduces them
res <- mnfrcmp(m, d, list(mean_x = function(z) mean(z$x),
                                    n = function(z) nrow(z)))
res$ok
res$results[, c("name", "recorded", "recomputed", "status")]

# recomputing one leaves the other reported as unchecked
mnfrcmp(m, d, list(n = function(z) nrow(z)))$unchecked

# and data that no longer matches the record is caught
mnfrcmp(m, data.frame(x = 1:11),
                  list(n = function(z) nrow(z)))$ok

# a stricter tolerance than the recorded one
mnfrcmp(m, d, list(mean_x = function(z) mean(z$x) + 1e-9,
                             n = nrow), tol = 0)$results$status

# a function that fails is reported, not thrown
mnfrcmp(m, d, list(mean_x = function(z) stop("column y is gone"),
                             n = nrow))$results[, c("name", "status")]

# statistics must be named
try(mnfrcmp(m, d, list(function(z) nrow(z))))
```

`mnfrecseed` *Record and restore the random number generator state*

Description

`mnfrecseed()` stores the generator's kind and its full state in the manifest; `mnfrstseed()` puts both back. Together they let a run that used randomness (a bootstrap, a permutation test, a simulation, a random split) be repeated exactly, even without knowing what seed was set or how many draws came before.

Usage

```
mnfrecseed(man)

mnfrstseed(man)
```

Arguments

`man`	A manifest, as from `mkmnf()` (a list).

Details

Why the state and not just a seed. `set.seed(1)` is reproducible only if everything before it is too: a single extra draw anywhere upstream shifts every subsequent value. Recording `.Random.seed` as it stood pins the actual position in the stream, and recording `RNGkind()` alongside it pins what that position means – R has changed its default `sample()` algorithm before, and a seed replayed under a different kind gives different numbers with no warning.

This is opt-in because the state is 626 integers for R's default Mersenne-Twister generator, which is a lot of manifest for an analysis with no randomness in it.

Side effects: if no generator state exists yet (`.Random.seed` is absent from the global environment), `mnfrecseed()` draws one uniform number to create it, and records `was_initialised = FALSE`. `mnfrstseed()` calls `RNGkind()` with the recorded kinds and assigns `.Random.seed` in the global environment, replacing the current state.

Errors: `man` that is not a list (or is a data frame); for `mnfrstseed()`, a manifest without a recorded state or with an empty kind.

Value

`mnfrecseed()`: the manifest with an `rng` element, a list of `kind` (character, the three `RNGkind()` values), `state` (integer, the `.Random.seed` vector) and `was_initialised` (logical, whether a state existed before the call). `mnfrstseed()`: the manifest, invisibly, having set the generator.

See Also

`capenv()`, `cpsfals()`.

Examples

```
set.seed(42)
m <- mnfrecseed(mkmnf(list(a = 1),
                                          env = FALSE))
first <- runif(3)

# any amount of other work can happen in between
invisible(runif(1000))

mnfrstseed(m)
identical(runif(3), first)

# what is stored
m$rng$kind
length(m$rng$state)

# a manifest without a recorded state cannot restore one
try(mnfrstseed(mkmnf(list(a = 1), env = FALSE)))
```

`morani` *Global Moran's I over a neighbour list*

Description

Measures spatial autocorrelation in a variable measured on areas: whether neighbouring areas tend to have similar values (positive I), dissimilar values (negative I), or neither. The p-value comes from randomly permuting the values over the areas. You must supply the neighbour structure (a list or a weight matrix); none is invented, because an administrative extract keyed on a region ships no geometry, and guessing adjacency would make the answer a property of the guess.

Usage

```
morani(x, nbrs, style = c("W", "B"), nperm = 9999L)
```

Arguments

`x`	Numeric vector, one value per area, no missing values.
`nbrs`	Either a list with one integer vector of neighbour indices per area (in the order of `x`), or a square $n \times n$ weight matrix.
`style`	`"W"` (the default) row-standardises the weights, so each area's neighbours carry a total weight of one; `"B"` leaves them as given (binary for a list). Row standardisation is the usual choice, and stops an area with many neighbours from dominating.
`nperm`	Integer, default `9999`: the number of random permutations for the null distribution.

Details

With weights w_{ij}, total $W = \sum_{ij} w_{ij}$ and $z_i = x_i - \bar{x}$,

$$I = \frac{n}{W} \frac{\sum_i \sum_j w_{ij} z_i z_j}{\sum_i z_i^2}.$$

The expectation of I under the null is $-1/(n-1)$, not zero, so a small negative I is what independence looks like in a small set of areas. The p-value comes from permuting the values over the areas, which needs no distributional assumption – and with a handful of regions no distributional assumption is safe. It is two-sided around $-1/(n-1)$, $(1 + \mathrm{count}(|I^* - E| \geq |I - E|))/(1 + B)$ over the B finite permuted values, so it is never exactly zero.

A neighbour list is binary; a weight matrix keeps its magnitudes (for example inverse distances). Self-neighbours and the diagonal are dropped. `style = "W"` divides each area's weights by their total. The permutations use R's random number generator, so call `set.seed()` first for a reproducible p-value. The work is done in compiled code.

Errors: fewer than three areas; missing values in `x`; a constant `x`; a neighbour index outside `1..n`; a list of the wrong length; a weight matrix that is not $n \times n$, not finite, or has negative weights; no area with any neighbour.

Value

A list with `I` (the statistic), `expectation` ($-1/(n-1)$), `nullmean` and `nullsd` (mean and standard deviation of the permuted values), `z` (`(I - nullmean) / nullsd`), `p_value`, `W` (the total weight), `n` (number of areas), `nperm` (permutations used) and `style`.

References

Moran, P. A. P. (1950). Notes on continuous stochastic phenomena. *Biometrika* 37(1/2), 17-23. Laniyonu (2017), on policing practices in gentrifying neighbourhoods, uses the statistic as here, to establish that areal residuals are spatially dependent.

Examples

```
# Six areas in a line, each adjacent to the next.
nb <- list(2L, c(1L, 3L), c(2L, 4L), c(3L, 5L), c(4L, 6L), 5L)

# A smooth gradient is strongly positively autocorrelated.
set.seed(1)
morani(c(1, 2, 3, 4, 5, 6), nb, nperm = 999L)[c("I", "p_value")]

# An alternating pattern is negatively autocorrelated.
set.seed(1)
morani(c(1, 6, 1, 6, 1, 6), nb, nperm = 999L)[c("I", "p_value")]

# And the null expectation is not zero.
-1 / (6 - 1)

# the same line as a binary weight matrix, unstandardised
w <- matrix(0, 6, 6)
for (i in 1:5) w[i, i + 1] <- w[i + 1, i] <- 1
```

```
set.seed(1)
morani(c(1, 2, 3, 4, 5, 6), w, style = "B", nperm = 999L)$I

# a constant variable has no autocorrelation to measure
try(morani(c(2, 2, 2), list(2L, c(1L, 3L), 2L)))
```

`mortdisp` *Deaths avoided under a counterfactual exposure reduction*

Description

The expected number of deaths (or cases) a year that a given drop in exposure would avoid in a population, from the baseline rate and a log-linear concentration-response coefficient. This is the health-impact function used in regulatory benefit assessments, for questions such as "how many deaths would a 5 microgram cut in PM2.5 avoid in this city?".

Usage

```
mortdisp(expdel, pop, brate, betaperunit, rare = FALSE, hirate = FALSE)
```

Arguments

`expdel`	Counterfactual reduction in exposure, in the unit of `betaperunit` (positive means the exposure goes down).
`pop`	At-risk population, a non-negative number of persons.
`brate`	Baseline rate in cases per person-year (0.008 is 800 per 100,000).
`betaperunit`	Log relative risk per unit of exposure (per unit, not per 10), a single finite number.
`rare`, `hirate`	`TRUE` to admit a baseline rate below 1e-5 (one per 100,000) or above 0.1 (10,000 per 100,000) per person-year; without them such a rate is refused, being also what a rate in the wrong unit looks like.

Details

The BenMAP-CE health-impact function (US EPA 2018; Anenberg et al. 2010):

$$\Delta Y = y_0 N(1 - e^{-\beta \Delta x}),$$

with y_0 the baseline rate per person-year, N the population, β the log relative risk per unit of exposure and Δx the reduction. A negative `expdel` (an increase) gives a negative number, the extra cases. A coefficient published per 10 units must be divided by 10 first (for example `log(1.08) / 10` for the PM2.5 all-cause RR of 1.08 per 10).

The baseline rate is per person-year, not per 100,000: 800 deaths per 100,000 a year is 0.008. A rate below 1e-5 or above 0.1 is refused unless `rare` or `hirate` says it is meant, because it is also what a rate in the wrong unit looks like; a rate above 1, or below 1e-11, is always refused. Every input must be a single finite number, and the population and rate non-negative.

Value

A single number: the expected cases avoided per year (negative for an increase in exposure).

References

US EPA (2018). Environmental Benefits Mapping and Analysis Program - Community Edition, user manual. Anenberg, S. C. et al. (2010). An estimate of the global burden of anthropogenic ozone and fine particulate matter on premature human mortality using atmospheric modeling. Environmental Health Perspectives 118(9), 1189-1195.

See Also

`polbrdn` for the cases attributable to the whole excess exposure, `vrfpol` which runs both.

Examples

```
# A 10-unit cut, a million people, 800 deaths per 100,000 a year.
mortdisp(10, 1e6, 0.008, 0.0039)

# The PM2.5 all-cause coefficient is published per 10 micrograms.
mortdisp(5, 2.8e6, 0.0075, log(1.08) / 10)

# A rare outcome must be declared.
mortdisp(5, 1e6, 5e-6, 0.01, rare = TRUE)

# A rate per 100,000 passed as a rate per person is refused.
try(mortdisp(10, 1e6, 800, 0.0039))
```

---

mrk *Merkle tree over capsule chunks (C backend)*

Description

A single SHA-256 over a whole file tells you it changed. A Merkle tree over its chunks tells you WHICH chunk changed, and proves that one chunk belongs to the pinned file without re-reading the rest of it. Use these functions to pin a large data file as a root digest, to find the part of a re-released file that differs, or to hand someone one chunk with a short proof that it is part of the pinned file.

Usage

```
mrkroot(chunks, scheme = 2L)

mrklvs(chunks, scheme = 2L)

mrkprf(chunks, index, scheme = 2L)

mrkvrf(leaf, proof, root, scheme = NULL)
```

Arguments

chunks	Chunk contents, in order: a character vector (one chunk per element), a single raw vector (one chunk), or a list of raw vectors and single strings, such as chnkfile() returns.
scheme	The hashing scheme: 2 (the default for building, with leaf and node prefixes) or 1 (no prefixes, as 0.5.12 and earlier built them). For mrkvrf(), NULL (the default) reads the scheme from proof; a number requires that scheme, and a proof that records a different one does not verify.
index	The chunk to prove, one whole number from 1 to the number of chunks.
leaf	The chunk whose membership is being verified: one string or one raw vector.
proof	The list returned by mrkprf().
root	The expected root digest, a 64-character hex string as mrkroot() returns.

Details

mrkroot() reduces the chunks to one root digest. mrklvs() returns the per-chunk digests the root is built from, so two capsules can be diffed chunk by chunk. mrkprf() returns the sibling digests on the path from one leaf to the root, and mrkvrf() replays that path.

Each leaf is the SHA-256 of the byte 0x00 followed by a chunk's bytes. Each parent is the SHA-256 of the byte 0x01 followed by the 64 bytes of its left and right child digests (the bytes, not their hex spelling). Levels are paired left to right. This is scheme = 2, the default from 0.5.13 on.

An unpaired node at an odd level is PROMOTED unchanged rather than hashed against a duplicate of itself. Duplicating it would let two different chunk lists produce the same root – the weakness behind CVE-2012-2459 – so promotion is a correctness requirement, not a preference. A promoted node adds no step to a proof, so a proof can be shorter than $\lceil \log_2 n \rceil$.

The two prefix bytes are the leaf and node domain separation of RFC 6962: without them an interior node, given as the 64 bytes of its two child digests, verifies as a "leaf" with the shorter proof from its parent. Roots and proofs made by 0.5.12 and earlier have no prefixes (scheme = 1, the weaker scheme). Pass scheme = 1 to mrkroot(), mrklvs() or mrkprf() to recompute such a root. mrkprf() records its scheme in the proof, and mrkvrf() reads it from there unless scheme is given. A proof with no recorded scheme was made by 0.5.12 or earlier: it is verified under scheme 1, with a warning. Pass scheme = 2 to mrkvrf() to refuse such a proof, since a scheme-1 check still accepts an interior node as a leaf.

Character chunks are taken as their bytes, so a root computed from character input equals the root of the same bytes as raw vectors. The comparison in mrkvrf() is cdgsteq() (constant time, case-sensitive).

Errors: chunks that is not character, raw or a list of raw vectors and single strings, or that contains NA; index outside 1..length(chunks); a scheme other than 1 or 2; a leaf that is more than one chunk; a proof without sibling and side, or with the two of different lengths.

Value

mrkroot() a length-1 character vector (64 lowercase hex characters), or NA for no chunks. mrklvs() a character vector of per-chunk digests, one per chunk (character(0) for none). mrkprf() a list with sibling (character, the sibling digests from the leaf upward), side (character, "left" or "right": the side the sibling is on) and scheme (integer). mrkvrf() a length-1 logical.

Examples

```
chunks <- c("row1,row2", "row3,row4", "row5,row6", "row7,row8")

root <- mrkroot(chunks)
root

# A single chunk's root is just its own leaf digest.
mrkroot("only") == csha256(c(as.raw(0), charToRaw("only")))

# The leaves are the per-chunk digests, so a diff names the culprit.
before <- mrklvs(chunks)
after <- mrklvs(c(chunks[1:2], "row5,row6-EDITED", chunks[4]))
which(before != after)

# Prove chunk 3 belongs, without holding chunks 1, 2 or 4.
pr <- mrkprf(chunks, 3)
pr
mrkvrf(chunks[3], pr, root)

# The proof fails for a chunk that was not in the tree.
mrkvrf("row5,row6-EDITED", pr, root)

# Any change to any chunk changes the root.
mrkroot(chunks) == mrkroot(c(chunks[1:3], "row7,row8 "))

# an odd count: the last node is promoted, so its proof has one step
mrkprf(c("a", "b", "c"), 3)

# an index past the end is refused
try(mrkprf(chunks, 5))

# a root pinned by 0.5.12 or earlier recomputes with scheme = 1, and its
# proofs (which record no scheme) verify with a warning
old <- mrkprf(chunks, 3, scheme = 1)
old$scheme <- NULL
withCallingHandlers(
  mrkvrf(chunks[3], old, mrkroot(chunks, scheme = 1)),
  warning = function(w) invokeRestart("muffleWarning"))
```

`mtchdid` *Difference in differences after matching on pre-treatment trajectories*

Description

Matches each treated unit to the comparison units most like it before treatment (its outcomes in the pre-period and, optionally, covariates), within an exact grouping if given, and estimates the difference in differences on the matched sample with the matching weights.

Usage

```
mtchdid(
  data,
  unit,
  time,
  y,
  ftreat,
  covs = NULL,
  ratio = 1L,
  exact = NULL,
  caliper = NULL,
  covm = c("pooled", "full"),
  conflvl = 0.95,
  cl = NULL
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`ftreat`	Column name of the period a unit is first treated (`NA`, 0 or `Inf` for a unit never treated). Treatment is taken as staying on.
`covs`	Optional covariate columns to match on (their pre-period means).
`ratio`	Comparison units per treated unit (default 1).
`exact`	Optional column name: match only within its values (a region, a district).
`caliper`	Optional maximum distance, in standard deviations of the distances.
`covm`	`"pooled"` (default; the pooled within-group covariance) or `"full"` (the whole sample's).
`conflvl`	Confidence level of the intervals reported (default 0.95).
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.

Details

Each unit is described by its outcome in each pre-period (the periods before the first adoption) and the pre-period means of the covariates. Distances are Mahalanobis, with the pooled within-group covariance of those features (`covariance = "pooled"`, as `MatchIt` computes it) or their covariance in the whole sample (`"full"`, as `stats::mahalanobis()` is usually given it); each treated unit takes its `ratio` nearest comparison units, with replacement (so the matches do not depend on the order of the treated units), only within its `exact` group, and only within `caliper` (in units of the

distance's standard deviation) if given. A comparison unit's weight is the sum over the treated units it serves of $1/\texttt{ratio}$. The estimate is `plndid` on the matched units with those weights, with errors clustered by unit; matching on pre-period outcomes makes parallel trends more credible but can bring in regression to the mean when the treated were chosen for an unusual pre-period (Daw and Hatfield 2018).

Value

A list of class `rmbl_matched_did`: `estimate` (a `plndid` result), `matches` (treated unit, comparison unit, distance), `weights` (per unit), `balance` (a `baltab` of the matching features before and after).

References

Daw JR, Hatfield LA (2018). Matching and regression to the mean in difference-in-differences analysis. Health Services Research 53(6):4138-4156. DOI 10.1111/1475-6773.12993

Applied with spatial nearest-facility matching within judicial districts: Zorro Medina A (2020). The failed war on pre-trial detention: evidence from a quasi-experimental reform. DOI 10.2139/ssrn.3686855

Examples

```
p <- polex()
m <- mtchdid(p, "district", "month", "crmrate", "refmonth", ratio = 2)
m$estimate
m$balance

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# each adopting town matched to one never treated on its path before 2019,
# the standard errors clustered by county: with two counties the package
# warns that so few clusters cannot be trusted
mtchdid(own, "town", "year", "rate", "first", cl = "county")$estimate
```

onl *Summary statistics accumulated in blocks*

Description

An accumulator for the count, mean, variance, skewness and kurtosis of a numeric column that does not fit in memory, or that arrives in pieces (a file read in chunks, results from several machines). Start one with `onlsmry()`, feed it blocks with `smryupd()`, combine two with `smrymrg()` and read the result with `smrystat()`.

Usage

```
onlsmry(x = NULL)

smryupd(acc, x)

smrymrg(a, b)

smrystat(acc)
```

Arguments

`x`	Numeric vector (finite; `NA` is skipped and an infinite value refused): the first block, or `NULL` (the default) for an empty accumulator; for `smryupd()`, the next block.
`acc`, `a`, `b`	Accumulators from `onlsmry()`.

Details

The merge is exact algebra, not an approximation: it uses Chan, Golub and LeVeque's parallel combination of central sums, extended to the third and fourth moments by Terriberry, so accumulating a column in blocks gives the answer of `cmom()` over the whole column at once, to the rounding of double arithmetic. Each block's sums are taken in two passes, as `var()` takes them, so values far from zero keep their digits: values near 10^8 with a spread of 1, in blocks of 100, give the variance of `var()` to a relative 10^{-11} (one pass lost nine digits). That is what makes it usable for a capsule member of any size: memory stays constant in the number of blocks.

The state is $(n, \bar{x}, M_2, M_3, M_4)$ with $M_k = \sum (x_i - \bar{x})^k$. `smrystat()` reports the sample variance $M_2/(n-1)$, the moment skewness $g_1 = (M_3/n)/(M_2/n)^{3/2}$ and the excess kurtosis $g_2 = (M_4/n)/(M_2/n)^2 - 3$ (the biased moment estimators, without small-sample correction). `NA` values are dropped from every block. Undefined quantities are `NaN`: the mean for $n = 0$, the variance for $n < 2$, the skewness for $n < 3$ and the kurtosis for $n < 4$, and both shape statistics for a constant column.

The object is a plain list and is copied on assignment like any other R value, so `smryupd()` returns the new accumulator and you must keep it: `acc <- smryupd(acc, block)`.

Errors: a block that is not numeric; an accumulator that does not come from `onlsmry()`.

Value

`onlsmry()`, `smryupd()` and `smrymrg()` return an object of class `bricklayer_online` (its `state` element is the numeric vector described in Details); `smrystat()` returns a named numeric vector with `n`, `mean`, `variance`, `sd`, `skewness` and `kurtosis` (excess).

References

Chan TF, Golub GH, LeVeque RJ (1983). Algorithms for computing the sample variance: analysis and recommendations. *The American Statistician* 37(3), 242–247.
DOI 10.1080/00031305.1983.10483115

See Also

`cmom()` for the single-pass batch version.

Examples

```
set.seed(1)
x <- stats::rnorm(1000)

# Accumulate in blocks of 100.
acc <- onlsmry()
for (i in seq(1, 1000, by = 100)) {
  acc <- smryupd(acc, x[i:(i + 99)])
}
smrystat(acc)

# The batch answer to rounding (the formulas are exact; the arithmetic is
# double):
all.equal(smrystat(acc)[["variance"]], stats::var(x))
all.equal(smrystat(acc)[["mean"]], mean(x))

# Two accumulators built independently can be merged, so blocks can be
# processed in any order or on different machines.
a <- smryupd(onlsmry(), x[1:400])
b <- smryupd(onlsmry(), x[401:1000])
all.equal(smrystat(smrymrg(a, b)), smrystat(acc))

# An empty accumulator reports nothing rather than zero.
smrystat(onlsmry())

# NA values are skipped; text that is not a number is refused
smrystat(onlsmry(c(1, NA, 3)))[c("n", "mean")]
try(onlsmry(c("a", "b")))
```

`optsqr` *The optimal search square for a circular normal datum*

Description

For a point datum with a circular normal distribution of position error, the square centred on the datum that a given effort, spread evenly, should cover to make the probability of success as large as possible (Frost 1996, chapter 6). With several efforts, each search is optimized in turn against the distribution as earlier unsuccessful searches left it.

Usage

```
optsqr(
  effort,
  model = c("invcube", "random", "definite", "mbeta"),
  m = 1,
  E = 1
)
```

Arguments

`effort`	Positive numeric vector: the effort (sweep width times track length) of each search in turn, in the square of the units of `E`.
`model`	`"invcube"` (the default; Frost's ideal search conditions), `"random"` (poor conditions), `"definite"` or `"mbeta"`.
`m`	Number in (0, 1]: the M-Beta height. Default 1.
`E`	Positive number: the total probable error of position. Default 1.

Details

The search factor `f` is the half side of the square in units of the total probable error of position `E` (the radius of the circle holding half the distribution, `E = 1.1774 sigma`). For relative effort `z = Z / E^2` the coverage is `C = z / (2 f)^2`, the containment is the circular normal's mass in the square, `(2 Phi(f E / sigma) - 1)^2` for a first search, and the probability of success `POC x POD(C)` with the POD curve of `model` (srchpod). The factor is found on a grid of 241 values from 0.02 to 10 and refined by `optimize()`. After a search the density inside its square is multiplied by `1 - POD`; every later search is optimized against what is left, and its `poc` and `pos` are conditional on the earlier searches having failed. Frost's figures 6-3 to 6-6 are drawn from it by `srchfig`.

Value

A data frame with one row per search: `search`, `effort`, `releff` (effort over `E^2`), `cumeff` (cumulative relative effort), `factor`, `side` (`2 f E`), `poc`, `coverage`, `pod`, `pos` and `cumpos` (the probability that one of the searches so far succeeded).

References

Frost, J. R. (1996). The Theory of Search: A Simplified Explanation. Soza and Company and U.S. Coast Guard Office of Search and Rescue. Chapter 6.

Koopman, B. O. (1957). The theory of search. III. The optimum distribution of searching effort. Operations Research 5(5), 613-626. DOI 10.1287/opre.5.5.613

Examples

```
# Frost's example: an effort of 4 E^2 on a point datum
optsqr(4)
# poor search conditions: the random search curve
optsqr(4, model = "random")
# three searches in a row, each optimal for what the last ones left
optsqr(c(4, 4, 8), E = 2)
try(optsqr(-1))
```

---

oqskg *Generate a standardised post-quantum signing key (deprecated name)*

Description

Kept so code written against the liboqs-backed version keeps working. The schemes are no longer reached through liboqs – they are implemented in this package – and the name no longer describes anything, so use `fipskg()` instead. New code should not call these functions.

Usage

```
oqskg(scheme = "SLH-DSA-SHA2-128s")

oqspubkey(key)
```

Arguments

`scheme`	Scheme name; see `fipskg()`.
`key`	A key from `oqskg()`.

Details

`oqskg()` calls `fipskg()` (with a fresh random seed) and `oqspubkey()` calls `fipspubkey()`, each after a deprecation warning from `.Deprecated()`. The keys are the same objects `fipskg()` returns, with the same classes.

Value

As `fipskg()` (a `bricklayer_fips_key`) and `fipspubkey()` (a `bricklayer_fips_public_key`).

Security

The implementation is hand-written. Its standardised schemes are checked byte for byte against OpenSSL 3.5 and NIST known-answer vectors in the test suite. Side channels are checked, not assumed: `inst/ctcheck` runs every operation that touches a secret under valgrind memcheck with the secret marked undefined (the ctgrind method), so a branch or a memory address that depends on it is a reported error, and afterwards scans the dead stack for copies of the secret. Both checks run in CI on every change, with GCC and with Clang. Timing is also measured on x86-64 and arm64 hardware (`inst/dudect`), power leakage is assessed in simulation under the value and the transition models (`inst/tvla`), and ML-KEM decapsulation and ML-DSA signing are first-order masked by default. These are checks of this code on those compilers and under those leakage models; the README's security section lists what each check does and does not cover.

See Also

`fipskg()`.

Examples

```
# Deprecated: use fipskg().
key <- suppressWarnings(oqskg("ML-DSA-65"))
key$scheme

# The same as the current name, after a warning.
pub <- suppressWarnings(oqspubkey(key))
identical(pub, fipspubkey(key))

# The warning names the replacement.
tryCatch(oqskg(), warning = function(w) conditionMessage(w))
```

---

orchk — *Check reported odds ratios against a labelled table, under every relabelling*

Description

Recomputes the odds ratios a report gives for each group from the underlying table of counts, and says whether the reported values follow from the table as it is labelled. If they do not, it tries every reassignment of the group labels (and the two outcome columns swapped) and lists the relabellings that do reproduce the reported values. Use it when a published odds ratio looks implausible: a four-fold ratio reported as thirty-six-fold is typically reproduced exactly by one such relabelling, which shows the groups were mislabelled rather than the arithmetic wrong.

Usage

```
orchk(cnts, ref = NULL, rptd, tol = 0.05)
```

Arguments

`cnts`	Numeric matrix, data frame or table of counts with row names (the groups) and exactly two columns: outcome absent, then outcome present.
`ref`	Character string: the row name of the reference group.
`rptd`	Named numeric vector of the odds ratios the report states, one per non-reference group, named by group. Extra names are ignored.
`tol`	Number, default `0.05`: the relative tolerance for a match (5 percent), to allow for rounding in the report.

Details

With row r holding a_r (outcome absent) and b_r (outcome present), the odds ratio of group r against the reference 0 is

$$OR_r = \frac{b_r/a_r}{b_0/a_0}.$$

A reported value matches when $|OR_r - \widehat{OR}_r| \le$ `tol` $\times |\widehat{OR}_r|$ for every non-reference group, and all values are finite. The search covers all $k!$ permutations of the row labels, each with and without the outcome columns swapped (the identity without a swap is the "as labelled" check), and is only run for $k \le 7$ groups. A zero cell gives an infinite or undefined odds ratio, which never matches.

Errors: `cnts` not a matrix, data frame or table with two columns and row names; `ref` not a row name; `rptd` not named by every non-reference row.

Value

A list of class `rmbl_or_check` (it prints as a box) with elements `computed` (named numeric odds ratios from the table as labelled), `reported` (the reported values, in the same order), `consistent` (logical), `matches` (a data frame with character `relabelling`, for example `"B -> C, C -> B"`, and logical `outcome_columns_swapped`, one row per relabelling that reproduces the reported values) and `verdict` (a one-sentence character summary).

See Also

`vrfmrg`, `rlblfrns`.

Examples

```
tab <- matrix(c(900, 100, 700, 300, 400, 600),
  ncol = 2, byrow = TRUE,
  dimnames = list(c("A", "B", "C"), c("no", "yes"))
)
# the odds ratios as labelled: B 3.86, C 13.5
orchk(tab, "A", c(B = 3.857, C = 13.5))$consistent

# a report that swapped B and C
chk <- orchk(tab, "A", c(B = 13.5, C = 3.857))
chk$matches
chk$verdict
```

```
# every non-reference group needs a reported value
try(orchk(tab, "A", c(B = 3.857)))
```

`ordcmp` *Compare rated scales (perceived mental health, life satisfaction) between groups*

Description

For a rated scale published as the percentage in each category with its confidence limits (perceived mental health from poor to excellent, life satisfaction from 0 to 10, agreement with a statement), the probability that a person in each group rates higher than a person in the reference group, with its confidence interval: a comparison of the whole distribution, not of one category cut from it.

Usage

```
ordcmp(
  data,
  grp,
  cat,
  est,
  lower,
  upper,
  lvls,
  ref,
  qual = NULL,
  conflvl = 0.95,
  pooled = NULL
)
```

Arguments

`data`	A data frame, one row per group and category (as `scests` gives, filtered to one geography, age group and gender).
`grp`, `cat`, `est`, `lower`, `upper`	Column names: the group, the category, the share in it (percent or proportion: only relative shares matter) and its published limits on the same scale.
`lvls`	The categories, from the lowest (least favourable) to the highest.
`ref`	The label of the reference group (required: any group of `data`).
`qual`	Optional column of quality flags (see `qltyrcd`).
`conflvl`	Confidence level of the published limits and of the result (default 0.95).
`pooled`	Optional labels of rows that pool other groups (see `indgap`).

Details

With shares p_i in the group and q_j in the reference over categories ordered from lowest to highest, the probability of a higher rating, ties counted half, is $A = \sum_i p_i(Q_{i-1} + q_i/2)$ with Q_i the reference's cumulative shares and $Q_0 = 0$: the Mann-Whitney probability, equal to $\alpha/(1+\alpha)$ for Agresti's (1980) generalized odds ratio α with ties split equally, and to the mean ridit of the group with the reference as the standard (Bross 1958). 0.5 is no difference; $2A - 1$ is Cliff's δ, the share rating higher less the share rating lower.

Its variance is that of the two-sample U statistic, $\sigma_1^2/n_1 + \sigma_0^2/n_0$, with σ_1^2 the variance over the group of each category's placement in the reference and σ_0^2 the converse (Brunner and Munzel 2000). The sample sizes are the effective ones the published limits imply: for each category, $n = p(1-p)/s^2$ with s the standard error from the limits' width, taking the median over the categories. That carries the design effect of the survey's weights and clusters, which the counts of persons (population totals) do not. The interval is built on the logit scale, so it stays inside 0 and 1.

The categories given in `lvls` are taken as the whole scale: rows for totals, for combined categories ("Excellent or very good") and for "not specified" are left out, and the shares are rescaled to sum to one over the levels. A group with a suppressed or missing category is not compared, since its distribution is unknown; one with an `E` category (in the group or the reference) is compared and marked, and one whose published interval misses its own estimate is not compared.

A higher rating means better only on a scale that runs that way. Give `lvls` from the least to the most favourable so that `probbet` reads as its name says.

Value

A data frame of class `rmbl_ordinal_compare`: `group`, `reference`, `probbet`, `lower`, `upper`, `delta`, `neff`, `neffref`, `usewithcau`, `poogrp`; the attribute `shares` is the matrix of shares (groups by levels), which `plot()` draws.

References

Agresti A (1980). Generalized odds ratios for ordinal data. Biometrics 36(1):59-67. DOI 10.2307/2530495

Bross IDJ (1958). How to use ridit analysis. Biometrics 14(1):18-38. DOI 10.2307/2527727

Brunner E, Munzel U (2000). The nonparametric Behrens-Fisher problem: asymptotic theory and a small-sample approximation. Biometrical Journal 42(1):17-25. DOI 10.1002/(SICI)1521-4036(200001)42:1<17::AID-BIMJ17>3.0.CO;2-U

See Also

`scests`, `indgap`.

Examples

```
# Self-perceived mental health, 15 years and older, Canada, 2022 (table
# 41-10-0067-01)
mh <- indiex("mentalhealth")
mh <- mh[mh$region == "Canada", ]
oc <- ordcmp(
  mh,
  "group",
```

```
  "rating",
  "estimate",
  "lower",
  "upper",
  lvls = c("Fair or poor", "Good", "Very good", "Excellent"),
  ref = "M\u00e9tis",
  qual = "quality"
)
oc
plot(oc)

# your own table: a made-up four-level rating for two groups, with limits
own <- data.frame(
  group = rep(c("Nation A", "Everyone else"), each = 4),
  rating = rep(c("Poor", "Fair", "Good", "Excellent"), 2),
  pct = c(10, 25, 40, 25, 6, 20, 44, 30),
  lo = c(7, 21, 35, 21, 5, 18, 42, 28),
  hi = c(13, 29, 45, 29, 7, 22, 46, 32)
)
ordcmp(own, "group", "rating", "pct", "lo", "hi",
       lvls = c("Poor", "Fair", "Good", "Excellent"), ref = "Everyone else")
```

`perdays` *Period length in days, from dates*

Description

Counts the days from one date to another, both ends included, so the result can be passed as the period length `t` to `adp()`, `admits()`, `adpcnts()` and `stkflow()`. It saves getting a fiscal year, a quarter or a leap year wrong by a day.

Usage

```
perdays(from, to)
```

Arguments

`from`	Start of the period: a single `Date`, or a single value `as.Date()` accepts (such as `"2024-01-01"`). Numbers are refused.
`to`	End of the period, inclusive, in the same forms; must not be before `from`.

Details

Inclusive of both ends, because a period running from the 1st to the 31st is thirty-one days of exposure, not thirty. Leap years need no special handling: the arithmetic is on dates, so 2024 comes out at 366 and 2023 at 365 without anyone choosing.

Errors when either argument is numeric, has a length other than one, is `NA`, or when `to` precedes `from`. Text that is not a date fails inside `as.Date()` with base R's own message.

Value

A numeric scalar: the number of days in the period, `to - from + 1`, for use as `t`. It has class `c("rmbl_perdays", "rmbl_measure", "numeric")` and an attribute `"lakner"` (`from` and `to` as dates), so it prints as a box; arithmetic, `round()`, `sum()`, `mean()`, `[`, `c()` and `as.numeric()` give a plain number, and every `t` argument takes it as it is.

See Also

`adp()`

Examples

```
perdays("2024-01-01", "2024-12-31") # a leap year
perdays("2023-01-01", "2023-12-31")
perdays("2025-04-01", "2026-03-31") # a fiscal year

# a quarter as the period of an average daily population
adp(4200, t = perdays(as.Date("2025-01-01"), as.Date("2025-03-31")))

# numbers are not dates, and the end cannot come first
try(perdays(1, 31))
try(perdays("2024-12-31", "2024-01-01"))
```

---

perminf *Permutation inference for a policy's adoption*

Description

Re-assigns the adoption dates across clusters (all the distinct assignments when there are few, random ones otherwise), re-estimates the difference in differences each time, and gives the share of assignments with an effect at least as large; for several outcomes, also the Westfall-Young step-down p-values that hold the family-wise error rate.

Usage

```
perminf(
  data,
  unit,
  time,
  y,
  ftreat,
  cl = NULL,
  stat = c("t", "estimate"),
  reps = 999L,
  seed = NULL,
  ties = c("count", "strict")
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`	Column names: the unit and the period (numeric).
`y`	One outcome column, or several.
`ftreat`	Column name of the period a unit is first treated (`NA`, 0 or `Inf` for a unit never treated). Treatment is taken as staying on.
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.
`stat`	`"t"` (default) or `"estimate"`.
`reps`	The number of random assignments (default 999).
`seed`	Optional seed for the random assignments, or NULL for none.
`ties`	`"count"` (default): an assignment whose statistic equals the observed one counts (the observed assignment is among those enumerated, so the exact p-value is at least $1/N$); `"strict"`: only larger ones count.

Details

The adoption period of each cluster (for example a judicial district's stage of a reform) is shuffled among the clusters, keeping the number of clusters adopting at each date, and `plndid`'s estimate (`statistic = "estimate"`) or t statistic (`"t"`, the default) is recomputed. With b of R random assignments at least as extreme as observed, $p = (b + 1)/(R + 1)$; when every distinct assignment is enumerated (their number is at most `reps`), p is the exact share. The test is Fisher's: it is valid in any sample when adoption dates were assigned at random across clusters.

For several outcomes, the same assignments give `stpdnpv`.

A placebo date is the same analysis with `ftreat` moved earlier, on the periods before the real adoption.

Value

A data frame of class `rmbl_permutation`, one row per outcome: `outcome`, `estimate`, `statistic`, `p_value`, `pstpdn`, `assignments`, `exact`; attribute `null` (the matrix of permuted statistics).

References

Fisher RA (1935). The Design of Experiments. Oliver and Boyd, Edinburgh.

Applied with the four stages of a reform permuted across districts: Acosta, Mejia and Zorro Medina (2023), see `plndid`.

See Also

`stpdnpv`.

Examples

```
p <- polex()
perminf(p, "district", "month", c("crmrate", "arrrate"),
                     "refmonth",
                     cl = "region", reps = 199, seed = 3)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# every assignment of the two adoption dates to the eight towns
perminf(own, "town", "year", "rate", "first", reps = 99, seed = 1)
```

`pgsig` *Atmospheric dispersion: sigmas, plume rise, plume, puff, grid and particles*

Description

Six standard models of how a pollutant released into the air spreads downwind, for exposure estimates around a stack, a fire or an industrial site (for example when relating air quality to health or justice outcomes by neighbourhood):

- `pgsig()`: how wide and how tall a plume has grown at a given distance, by atmospheric stability class;
- `briggs()`: how far a hot plume rises above the stack top;
- `gplume()`: the steady concentration at receptor points from a continuous release;
- `gspuff()`: the concentration from a single instantaneous release, some time after it;
- `advdif2d()`: a concentration field on a grid moved by wind and spread by diffusion, step by step;
- `lagrprt()`: the same physics as a random walk of many particles.

They are screening models with textbook assumptions (flat terrain, steady uniform wind, no chemistry or deposition), not regulatory model runs.

Usage

```
pgsig(x, stab = "D", set = "rural")

briggs(
  x,
  u,
  diam,
  exitvel,
  stktmp,
  ambtmp,
  stab = "D",
  dthdz = NULL,
  g = 9.80616
)

gplume(q, u, h, rcpt, stab = "D", set = "rural", mixh = NULL, nimg = 3)

gspuff(mass, u, h, rcpt, t, stab = "D", set = "rural", mixh = NULL, nimg = 3)

advdif2d(c0, u, v, kx, ky, dx, dy, dt, nsteps, source = NULL, maxwork = 1e+08)

lagrprt(
  n,
  x0,
  y0,
  u,
  v,
  kx,
  ky,
  dt,
  nsteps,
  seed = 0,
  grid = NULL,
  normals = NULL,
  maxwork = 1e+08
)
```

Arguments

x	Downwind distance(s) in metres.
stab	Pasquill stability class, one of "A" to "F".
set	"rural" (the default) or "urban": which set of Briggs coefficients to use.
u	Wind speed in m/s (the x velocity for the grid models).
diam, exitvel, stktmp, ambtmp	Stack diameter (m), exit velocity (m/s), stack gas and ambient temperatures (K). Only buoyant rise is modelled: a stack at the air's temperature (within half a kelvin) has zero rise, and momentum rise from the exit velocity is not added.

dthdz	Potential temperature gradient (K/m) for the stable classes; the defaults are 0.020 (E) and 0.035 (F).
g	Gravitational acceleration in m/s^2, default 9.80616.
q	Emission rate (mass per second, for example g/s).
h	Effective stack height (m): physical height plus plume rise.
rcpt	A matrix with columns x, y, z, or a list of length-3 vectors.
mixh	Mixing-lid height in metres, or NULL (the default) for no lid. Under a lid the plume is reflected between the ground and the lid.
nimg	Whole number, default 3: the number of lid image pairs (0 to 1,000). The terms decay quickly; a few are enough.
mass	Puff mass (for example g).
t	Time since release (s).
c0	Initial concentration grid: a non-negative numeric matrix, one row per x cell and one column per y cell.
v	Wind velocity along y (m/s), may be negative.
kx, ky	Eddy diffusivities along x and y (m^2/s), non-negative.
dx, dy	Grid spacings (m), positive.
dt	Time step (s), positive.
nsteps	Whole number of time steps (0 returns the start).
source	Source grid (concentration per unit time), or NULL.
maxwork	The most cell updates (cells x steps x sub-steps; for lagrprt(), particles x steps) a run may take, 100 million by default (a few minutes of the explicit loop); Inf lifts it. The same physical run costs the same whether the caller sub-steps it or the scheme does, so the cap reads the same either way.
n	Whole number of particles, at most 100 million.
x0, y0	Release point (m).
seed	Number, default 0: the seed of the particle random walk.
grid	NULL, or c(xmin, x_max, y_min, y_max, nx, ny) to count particles into a concentration grid.
normals	A function (n, seed, stream) returning n standard normals; step k (from 0) draws streams 2k for x and 2k + 1 for y. The default draws from the package's shared uniform stream (cunif) through the normal quantile. rmorie's LagrangianParticles() uses its own Philox stream, so its paths differ from the default here; pass its generator to reproduce them.

Details

pgsig(): Briggs (1973) fits of the Pasquill-Gifford dispersion coefficients by stability class A-F, rural sigmay = a_y x (1 + 0.0001 x)^(-1/2) (urban: 0.0004) and sigmaz = a_z x (1 + b_z x)^(e_z). briggs(): buoyancy flux F = g v_s d^2 (T_s - T_a) / (4 T_s); final rise 21.425 F^(3/4) / u with x_f = 49 F^(5/8) when F < 55, else 38.71 F^(3/5) / u with x_f = 119 F^(2/5), for classes A-D; 2.6 (F / (u s))^(1/3) with s = g (dtheta/dz) / T_a and x_f = 2.0715 u / sqrt(s) for E and F;

transitional rise `1.6 F^(1/3) x^(2/3) / u` before `x_f` (EPA ISC3). `gplume()`: `Q / (2 pi u sigmay sigmaz) exp(-y^2 / (2 sigmay^2)) V`, with `V` the vertical term with ground reflection and, under a mixing lid, `nimg` pairs of image terms. `gspuff()`: an instantaneous release with `sigma_x = sigmay` evaluated at the travel distance `u t`. `advdif2d()`: explicit upwind advection, central diffusion, forward Euler, zero boundaries. The explicit scheme is stable when `|u dt/dx| + |v dt/dy| + 2 (kx dt/dx^2 + ky dt/dy^2) <= 1` (the von Neumann bound for upwind advection with forward-time centred-space diffusion); that sum is returned as `stabn` with `stable`. A call outside the bound is not run as given: each step is split into enough sub-steps of a smaller `dt` to satisfy it (reported as `substeps` and `dt`), with a warning, so the field is a solution over the same physical time rather than a numerical explosion. The separate `cfl` and `difn` are reported for reference; each below 1 is necessary, not sufficient. `lagrprt()`: the random walk `x + u dt + sqrt(2 K dt) xi` (Thomson 1987) with an optional counting grid.

The stability classes run from `"A"` (very unstable, sunny with light wind) through `"D"` (neutral, overcast or windy) to `"F"` (stable, clear night with light wind). Units are SI: metres, seconds, kelvin; with `q` in g/s, `gplume()` returns g/m^3, and with `mass` in g, `gspuff()` returns g/m^3. The plume is not defined at or upwind of the source: a receptor with $x \leq 0$ gets 0. The plume's `h` is the effective height, the stack height plus the rise from `briggs()`.

Every physical input is checked: emission rates, masses, heights and distances are non-negative, wind speed, travel time and grid steps positive, `nimg` a whole number up to 1,000, the stack as hot as the air or hotter (temperatures below 150 are refused as probably Celsius), and `x0`, `y0` single numbers; a release above `mixh` warns, and so does a receptor below ground or above the lid (the reflection terms give mirror values there). Unknown stability classes or settings are errors.

Cost: `gplume()` and `gspuff()` are fast for thousands of receptors. `advdif2d()` is an R loop at roughly 0.8 million cell updates a second, and both grid models refuse a run above `maxwork` cell or particle updates. The particle walk is reproducible from `seed` and does not touch R's random number stream. Progress bars are shown in an interactive session.

Value

`pgsig()`: a list with `sigmay` and `sigmaz`. `briggs()`: a list with `flux`, `finalrise`, `xfinal` and `rise` (at each `x`). `gplume()` and `gspuff()`: a numeric vector of concentrations, one per receptor. `advdif2d()`: a list with `field`, `mass`, `cfl`, `difn`, `stabn`, `stable`, `dt`, `substeps` and the axes, as a list of class `rmbl_field` that `plot()` draws as an image; `lagrprt()` a list of class `rmbl_particles` that `plot()` draws as points, with `x`, `y` (final particle positions), `mean_x`, `meany`, `grid` (as given) and, with a grid, `concentration` (an `nx` by `ny` matrix of particle density per unit area, summing to the share of particles inside the grid divided by the cell area).

Boundary

the outermost ring of cells is held at zero (an open boundary): mass that reaches it leaves the domain and `mass` falls accordingly. Pad `c0` with empty cells so the plume of interest stays inside the ring.

References

Briggs, G. A. (1973). Diffusion estimation for small emissions. ATDL Contribution 79. Briggs, G. A. (1975). Plume rise predictions. American Meteorological Society. Turner, D. B. (1970). Workbook of Atmospheric Dispersion Estimates. EPA AP-26. Seinfeld, J. H. and Pandis, S. N. (2016). Atmospheric Chemistry and Physics, 3rd ed. Thomson, D. J. (1987). Criteria for the

selection of stochastic models of particle trajectories in turbulent flows. Journal of Fluid Mechanics 180, 529-556.

Examples

```
# plume widths 1 km downwind: neutral, then very stable, rural and urban
pgsig(1000, "D")
pgsig(c(500, 1000, 2000), "F", set = "urban")

# rise of a hot plume, and the effective height it gives a 50 m stack
rise <- briggs(c(100, 1000), u = 4, diam = 1, exitvel = 12,
                          stktmp = 420, ambtmp = 285)
rise$rise
heff <- 50 + rise$finalrise

# ground-level concentration (g/m^3) from 100 g/s along the centreline
rec <- cbind(x = c(500, 1000, 2000, 8000), y = 0, z = 0)
gplume(100, u = 4, h = heff, rcpt = rec)

# under a 120 m mixing lid the far receptor sees a trapped plume
gplume(100, u = 4, h = heff, rcpt = rec, mixh = 120)

# a 1 kg puff, 10 minutes after release, at the puff centre
gspuff(1000, u = 3, h = 10, rcpt = rbind(c(1800, 0, 0)), t = 600)

# a small grid: a blob moved by wind and spread by diffusion
c0 <- matrix(0, 15, 15)
c0[4, 8] <- 1
f <- advdif2d(c0, u = 1, v = 0, kx = 0.5, ky = 0.5,
                            dx = 1, dy = 1, dt = 0.2, nsteps = 20)
f$stable
round(f$mass, 3)

# 2,000 particles, counted on a grid
p <- lagrprt(2000, x0 = 0, y0 = 0, u = 1, v = 0, kx = 1, ky = 1,
                           dt = 1, nsteps = 10, seed = 1,
                           grid = c(-10, 30, -20, 20, 8, 8))
c(p$mean_x, p$meany)

# inputs that cannot be right are refused
try(pgsig(100, "G"))
try(briggs(100, u = 4, diam = 1, exitvel = 12,
                       stktmp = 150 + 5, ambtmp = 20))
```

`piperate` *Justice-pipeline rates with a lag between stages*

Description

The rate at which one stage of the justice process follows another (charges per reported offence, settlements per charge, convictions per trial), summing each over a window and setting the outcome's window after the input's, because cases take time to move through.

Usage

```
piperate(num, den, win = 12L, lag = 0L, grp = NULL)
```

Arguments

num, den	The outcome and input counts, in period order.
win	The number of periods summed (default 12).
lag	The number of periods the input's window precedes the outcome's (default 0).
grp	Optional grouping (for example a court or municipality), the counts in period order within it.

Details

For period t, a window of W periods and a lag of L,

$$R_t = \frac{\sum_{s=t-W+1}^{t} O_s}{\sum_{s=t-L-W+1}^{t-L} I_s},$$

with O the outcome stage and I the input stage. With $W = 1, L = 0$ it is the same-period ratio (cases with detention over cases charged; acquittals over acquittals and convictions). Rates are NA until the windows are filled, and when the denominator is zero. These are administrative counts of every case, so no sampling interval is given.

Value

A data frame of class rmbl_pipeline_rate: group, index (the period's position within its group), numerator, denominator (the window sums), rate.

References

Council of Europe, European Commission for the Efficiency of Justice (CEPEJ) (2014). European judicial systems: efficiency and quality of justice. CEPEJ Studies 20.

Applied to the stages of a Colombian criminal-procedure reform: Acosta, Mejia and Zorro Medina (2023), see plndid.

See Also

caseflow for clearance rates and inventories.

Examples

```
set.seed(2)
reports <- rpois(36, 400)
charges <- rpois(36, 90)
piperate(charges, reports, win = 12, lag = 3)[30:36, ]
```

---

plmfld *A Gaussian-plume concentration field on a grid*

Description

Evaluates gplume at every point of an x by y ground-level grid (or at height z) and returns the field as a matrix with the axes, ready for plot(). Use it to map where the plume from a single stack touches the ground, how far downwind the highest concentration lies, and how wide the affected area is.

Usage

```
plmfld(q, u, h, x, y, z = 0, stab = "D", set = "rural", mixh = NULL, nimg = 3)
```

Arguments

q, u, h, stab, set, mixh, nimg	As in gplume: emission rate (mass per second), wind speed (m/s), effective stack height (m), Pasquill class "A" to "F" (default "D"), "rural" (default) or "urban", mixing-lid height in metres (default NULL, no lid), number of lid image pairs (default 3).
x, y	Numeric vectors, strictly increasing: the downwind and crosswind coordinates of the grid, in metres from the source.
z	Numeric scalar, the receptor height in metres; default 0 (ground level).

Details

Every grid point (x_i, y_j, z) is passed to gplume in one call, with the source at the origin and the wind along $+x$; the field is therefore zero (or vanishingly small) upwind and at $x \leq 0$. The concentration is in the mass unit of q per cubic metre (grams per cubic metre for q in grams per second). dx and dy are the median grid spacings, used by the plot method for the cell size. The cost is one plume evaluation per grid point, so a 100 by 100 grid is 10,000 evaluations.

Errors: x or y not numeric or not strictly increasing; z negative; and every check of gplume (q, u, h, the stability class, the setting, the mixing height).

Value

A list of class rmbl_field: field (a numeric matrix of concentrations, rows along x, columns along y), x, y, dx and dy (median spacings, NA for a single coordinate), units ("concentration") and title (character, used by the plot method).

See Also

gplume; plot() draws the field as an image with contours (see plot-methods).

Examples

```
f <- plmfld(
  100,
  5,
  50,
  x = seq(100, 3000, by = 100),
  y = seq(-500, 500, by = 50)
)
dim(f$field)
plot(f)

# where the ground-level maximum lies, downwind
f$x[which.max(apply(f$field, 1, max))]

# a more unstable atmosphere brings the plume down sooner
fb <- plmfld(
  100,
  5,
  50,
  x = seq(100, 3000, by = 100),
  y = seq(-500, 500, by = 50),
  stab = "B"
)
fb$x[which.max(apply(fb$field, 1, max))]

# the coordinates must increase
try(plmfld(100, 5, 50, x = c(300, 100), y = c(-50, 0, 50)))
```

---

plndid *Difference in differences with two-way fixed effects*

Description

The effect of a policy on a unit-by-period panel (municipalities by month, blocks by quarter, courts by year), from a regression of the outcome on the treatment indicator with unit and period fixed effects, and standard errors clustered by unit or by any coarser grouping.

Usage

```
plndid(
  data,
  unit,
  time,
  y,
```

```
  trtd,
  covs = NULL,
  cl = NULL,
  fe = c("unit", "time"),
  w = NULL,
  comb = NULL,
  conflvl = 0.95
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`trtd`	Column name of the treatment indicator (0 or 1).
`covs`	Optional column names of time-varying controls.
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.
`fe`	`c("unit", "time")` (default) or `"time"`.
`w`	Optional column name of non-negative weights, such as population or matching weights, for weighted least squares with weighted within-means. Rows of weight 0 take no part in the fit.
`comb`	Optional linear combination of the terms, named by them, reported with its delta-method standard error as the attribute `combined`. For a phase-in model with an exposure term $E_{it} = \min(t - G_i, K)$, `comb = c(treated = 1, exposure = 12)` is the effect after twelve periods, with its error, and as a percentage of the comparison units' pre-period mean (that mean taken as fixed).
`conflvl`	Confidence level of the intervals reported (default 0.95).

Details

The model is

$$Y_{it} = \beta T_{it} + X_{it}'\gamma + \delta_i + \mu_t + \varepsilon_{it},$$

fitted by ordinary least squares after the within transformation (alternating projections, so an unbalanced panel is fine). The variance is the one-way cluster-robust CR1 estimator,

$$\hat{V} = \frac{G}{G-1}\frac{N-1}{N-K}(\tilde{X}'\tilde{X})^{-1}\Big(\sum_g \tilde{X}_g'\hat{u}_g\hat{u}_g'\tilde{X}_g\Big)(\tilde{X}'\tilde{X})^{-1},$$

with K counting the regressors and the fixed effects not nested in the clusters, and tests and intervals use Student's t on $G - 1$ degrees of freedom. These are the conventions of Stata's `reghdfe` and R's `fixest`.

The relative effect is $\hat{\beta}$ over the mean outcome of the units never treated (or not yet treated) in the periods before the first treatment, the way an effect on a rate is usually reported ("a 12% fall").

With `fe = "time"` the unit effects are left out, and the model is the cross-section with period effects of a comparison between kinds of units (for example blocks in dynastic and in other wards).

When treatment starts at different times, this estimator weights earlier-treated units as controls for later ones, and can be biased if the effect changes over time: see `bacondcmp` for the weights and `stgatt` for estimators that avoid them.

Value

A data frame of class `rmbl_did`, one row per term (the treatment first, then the covariates): `term`, `estimate`, `se`, `lower`, `upper`, `t`, `p_value`. Attributes: `n`, `clusters`, `ctlmean` (the pre-period mean of the comparison units), `releff` (the estimate over it), `vcov`, and with `comb` also `combined`.

References

Bertrand M, Duflo E, Mullainathan S (2004). How much should we trust differences-in-differences estimates? Quarterly Journal of Economics 119(1):249-275. DOI 10.1162/003355304772839588

Cameron AC, Gelbach JB, Miller DL (2011). Robust inference with multiway clustering. Journal of Business and Economic Statistics 29(2):238-249. DOI 10.1198/jbes.2010.07136

Cameron AC, Miller DL (2015). A practitioner's guide to cluster-robust inference. Journal of Human Resources 50(2):317-372. DOI 10.3368/jhr.50.2.317

Applied to a staggered criminal-procedure reform, municipality by month: Acosta C, Mejia D, Zorro Medina A (2023). On the tension between due process protection and public safety: the case of an extensive procedural reform in Colombia. DOI 10.2139/ssrn.4585682

See Also

`evstudy`, `stgatt`, `perminf`, `plnpois`.

Examples

```
p <- polex()
plndid(p, "district", "month", "crmrate", "reformed")
# clustered by region instead of district: only four regions, so the
# package warns, and wcb() is the tool for so few clusters
plndid(p, "district", "month", "crmrate", "reformed", cl = "region")

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
```

```
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# the effect of the by-law, with income as a control, clustered by town
# (the default), weighted by population
plndid(own, "town", "year", "rate", "bylaw", covs = "income", w = "pop")
```

`plndml` *Double machine learning for panels with fixed effects*

Description

The effect of a treatment (continuous or binary) on an outcome in a unit-by-period panel, with unit and period fixed effects removed and the confounding through covariates fitted by a machine learner, cross-fitted by unit, with an orthogonal (Neyman) score and errors clustered by unit.

Usage

```
plndml(
  data,
  unit,
  time,
  y,
  trt,
  covs,
  fe = c("unit", "time"),
  lrn = "lasso",
  basis = c("linear", "poly2"),
  folds = 5L,
  nrep = 1L,
  seed = 1L,
  cl = NULL,
  conflvl = 0.95
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`trt`	Column name of the treatment (continuous, or 0/1).
`covs`	Column names of the time-varying covariates.
`fe`	`c("unit", "time")` (default) or `"time"`.
`lrn`	`"lasso"` (default), `"ridge"`, `"ols"`, or a function.
`basis`	`"linear"` (default) or `"poly2"`.

`folds`	Cross-fitting folds of units (default 5).
`nrep`	Repeated sample splits (default 1).
`seed`	Seed for the folds (default 1).
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.
`conflvl`	Confidence level of the intervals reported (default 0.95).

Details

The model is the partially linear panel model

$$Y_{it} = \theta D_{it} + g(X_{it}) + \alpha_i + \gamma_t + \varepsilon_{it}.$$

Following Clarke and Polselli (2025), the fixed effects are removed by the within transformation ($\tilde{Y}$, $\tilde{D}$, $\tilde{X}$), and $\ell(\tilde{X}) = E(\tilde{Y} \mid \tilde{X})$ and $m(\tilde{X}) = E(\tilde{D} \mid \tilde{X})$ are fitted by the learner on K folds of units (all of a unit's periods in one fold, so a unit's own errors never help predict it). With the out-of-fold residuals $U = \tilde{Y} - \hat{\ell}$ and $V = \tilde{D} - \hat{m}$, the partialling-out estimator is

$$\hat{\theta} = \sum VU / \sum V^2,$$

the solution of the Neyman-orthogonal score $(U - \theta V)V$ of Chernozhukov et al. (2018). Its variance is the cluster-robust sandwich over units,

$$\hat{V} = \frac{G}{G-1} \frac{\sum_g \left(\sum_{i \in g} V_i (U_i - \hat{\theta} V_i) \right)^2}{(\sum V^2)^2},$$

with t on $G - 1$ degrees of freedom. Folds are drawn with `seed`; `nrep` repeated splits are combined by the median of the estimates and of the variances plus the squared deviations from it (Chernozhukov et al. 2018, section 3.4).

Learners, all in base R: `"lasso"` (coordinate descent, penalty by 5-fold cross-validation), `"ridge"` (penalty by generalised cross-validation), `"ols"`, or a function returning predictions (for example a random forest from another package), called as `function(x_train, y_train, x_test)`. `basis = "poly2"` adds squares and pairwise products of the covariates, built from the covariates as observed and then demeaned, so a confounding that is a combination of them is removed exactly. The demeaned features are what the learners see: a covariate fixed within a unit is absorbed by the unit effect and carries no information. A learner given only the demeaned covariates (a random forest, say) approximates a nonlinear confounding less well; the poly2 basis, or a basis of the user's own as further covariates, is the remedy.

Value

A data frame of class `rmbl_dml`: `estimate`, `se`, `lower`, `upper`, `t`, `p_value`; attributes `n`, `clusters`, `learner`, `folds`, `repeats`, and `r2` (the out-of-fold R-squared of the two nuisance fits).

References

Chernozhukov V, Chetverikov D, Demirer M, Duflo E, Hansen C, Newey W, Robins J (2018). Double/debiased machine learning for treatment and structural parameters. The Econometrics Journal 21(1):C1-C68. DOI 10.1111/ectj.12097

Clarke PS, Polselli A (2025). Double machine learning for static panel models with fixed effects. The Econometrics Journal 29:69-86. DOI 10.1093/ectj/utaf011

See Also

plrcf (one cross-section), stgatt with covariates (doubly robust difference in differences).

Examples

```
p <- polex()
set.seed(9)
p$unemployment <- 6 + sin(p$month / 6) + stats::rnorm(nrow(p), 0, 0.5)
p$patrols <- 0.5 * p$unemployment + p$reformed + stats::rnorm(nrow(p))
plndml(p, "district", "month", "crmrate", "patrols",
          c("unemployment", "arrrate"), lrn = "ridge")

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# the by-law as the treatment, income as the confounder
plndml(own, "town", "year", "rate", "bylaw", "income", lrn = "ridge")
```

---

plnpois *A fixed-effects Poisson or negative binomial regression for counts*

Description

The effect of a policy on counts (stops, arrests, offences) with unit and period fixed effects, reported as an incidence rate ratio, with cluster-robust standard errors that do not need the counts to be Poisson.

Usage

```
plnpois(
  data,
  unit,
  time,
  y,
  trtd,
  covs = NULL,
  offset = NULL,
  cl = NULL,
  family = c("poisson", "negbin"),
  conflvl = 0.95
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`trtd`	Column name of the treatment indicator (0 or 1).
`covs`	Optional column names of time-varying controls.
`offset`	Optional column name of the log exposure (for example log population).
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.
`family`	`"poisson"` (default) or `"negbin"` (NB2, $Var(Y) = \mu + \mu^2/\theta$, with θ by maximum likelihood).
`conflvl`	Confidence level of the intervals reported (default 0.95).

Details

The model is $E(Y_{it}) = \exp(\beta T_{it} + X_{it}'\gamma + \delta_i + \mu_t)$ (times the exposure, with `offset` its log), fitted by maximum likelihood. Units whose counts are all zero carry no information about β and are dropped first, as are periods with no count anywhere. The Poisson likelihood with fixed effects is consistent for β whatever the dispersion (Wooldridge 1999); the variance is the cluster-robust sandwich, with the small-sample factors of `plndid`. The rate ratio is e^{β}, with interval $e^{\hat{\beta} \pm z\, se}$.

With `family = "negbin"` the variance is $\mu + \mu^2/\theta$: the fit alternates iteratively reweighted least squares at fixed θ with the maximum of the profile likelihood in θ, and the sandwich uses the negative binomial scores with θ held at its estimate (as `sandwich::vcovCL()` does for `MASS::glm.nb()`; `fixest::fenegbin()`, which treats θ as a parameter of the variance too, gives slightly larger errors). With unit dummies this is the unconditional fixed-effects negative binomial, which behaves well as the number of periods grows (Allison and Waterman 2002); with few periods per unit prefer the Poisson, which is consistent whatever the dispersion.

Value

A data frame of class `rmbl_fe_poisson`, one row per term: `term`, `estimate` (log scale), `se`, `rateratio`, `lower`, `upper` (of the rate ratio), `p_value`; attributes `n`, `drounits`, `clusters`, `family`, `theta` (`Inf` for the Poisson).

References

Wooldridge JM (1999). Distribution-free estimation of some nonlinear panel data models. Journal of Econometrics 90(1):77-97. DOI 10.1016/S0304-4076(98)00033-5

Santos Silva JMC, Tenreyro S (2006). The log of gravity. Review of Economics and Statistics 88(4):641-658. DOI 10.1162/rest.88.4.641

Allison PD, Waterman RP (2002). Fixed-effects negative binomial regression models. Sociological Methodology 32:247-265. DOI 10.1111/1467-9531.00117

Examples

```
p <- polex()
plnpois(p, "district", "month", "crimes", "reformed", offset = "logpop")

# these counts are not overdispersed, so the negative binomial collapses
# to the Poisson fit and says so
plnpois(p, "district", "month", "crimes", "reformed", offset = "logpop",
           family = "negbin")

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# counts with the population as exposure
own$crimes <- round(own$rate * own$pop / 1e4)
own$logpop <- log(own$pop)
plnpois(own, "town", "year", "crimes", "bylaw", offset = "logpop")
```

`plnpwr` *Power and minimum detectable effect of a difference-in-differences design*

Description

Simulates the design on real untreated data: assigns adoption at random to units (or clusters) as the study would, adds an effect of known size, estimates `plndid`, and counts rejections; so the serial correlation and the clustering of the real data are carried, as formula-based power calculations do not.

Usage

```
plnpwr(
  data,
  unit,
  time,
  y,
  ntrt,
  adopt,
  eff = NULL,
  cl = NULL,
  reps = 200L,
  alpha = 0.05,
  power = 0.8,
  seed = NULL
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`ntrt`	Units (or clusters) to treat in each replication.
`adopt`	Adoption periods to draw from.
`eff`	Effect sizes to try (default a grid from 0 to two standard deviations of the outcome).
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.
`reps`	Replications per effect size (default 200).
`alpha`	Test level (default 0.05).
`power`	For the minimum detectable effect (default 0.8).
`seed`	Optional seed for the power simulations.

Details

In each replication, `ntrt` units (or clusters, with `cl`) are drawn at random and given adoption dates drawn from `adopt`; `effect` is added to their outcomes from adoption on; the cluster-robust t test of `plndid` at level `alpha` is counted as a rejection. With `effect = 0` the rejection rate is the test's actual size on these data, which should be near `alpha`: a larger one means the errors are understated (too few clusters, say), and the power figures are then optimistic too. The minimum detectable effect is where the power curve crosses `power`, by linear interpolation (Burlig, Preonas and Woerman 2020).

Value

A data frame of class `rmbl_power`: `effect`, `power`; attributes `mde` and `size` (the rejection rate at effect 0).

References

Burlig F, Preonas L, Woerman M (2020). Panel data and experimental design. Journal of Development Economics 144:102458. DOI 10.1016/j.jdeveco.2020.102458

Examples

```
p <- polex()
pre <- p[p$month <= 12, ]
pw <- plnpwr(pre, "district", "month", "crmrate", ntrt = 10, adopt = 7:10,
                  eff = c(0, 1, 2, 3), reps = 30, seed = 1)
pw
plot(pw)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# the power of four towns adopting in 2017 or 2018, from the years before
# any by-law
plnpwr(own[own$year <= 2018, ], "town", "year", "rate", ntrt = 4,
       adopt = 2017:2018, eff = c(0, 2, 4), reps = 20, seed = 1)
```

`plot.rmbl_census_cwb` *Plot methods for the package's results*

Description

Every result the package returns draws itself with `plot()`: no arguments beyond the object are needed, and every method takes the same customisation (`main`, `col` or `palette`, `...` for the base-graphics call, the grid from the session option; see `plotopt`). Titles fit their panel: a title too wide for it is drawn a step smaller, and in a panel set one that still does not fit becomes a letter with the key printed under the panels (the returned data carry it as the `plnkey` attribute); the left margin is sized from the widest tick label, so an axis title never overlaps the numbers. (Session defaults: see `plotopt` for the session defaults). Each method returns the data it drew, invisibly, as a data frame.

Usage

```
## S3 method for class 'rmbl_census_cwb'
plot(
  x,
  ...,
  component = c("cwb", "income", "education", "housing", "labour"),
  main = NULL,
  col = NULL,
  palette = NULL
)

## S3 method for class 'rmbl_profile_share'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_hotspot'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_connection'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_healing_change'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_community_index'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_indigenous_gap'
plot(
  x,
  ...,
  what = c("difference", "ratio"),
  main = NULL,
  col = NULL,
```

```
  palette = NULL
)

## S3 method for class 'rmbl_dsr'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_prevalence'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_exposure_trend'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_trc_progress'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_ordinal_compare'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_wellness_profile'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_holistic_balance'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_dual_continuum'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_distinctions'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_case_flow'
plot(
  x,
  ...,
  what = c("flow", "clearance"),
  main = NULL,
  col = NULL,
  palette = NULL
)

## S3 method for class 'rmbl_life_table'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_disposition_mix'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_density'
plot(x, ..., points = TRUE, main = NULL,
```

```
  col = NULL, palette = NULL)

## S3 method for class 'rmbl_knox'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_clearance_decomposition'
plot(x, ..., what = NULL,
  maxparts = 20L, main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_synthdid'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_power'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_balance'
plot(
  x,
  ...,
  threshold = 0.1,
  after = NULL,
  main = NULL,
  col = NULL,
  palette = NULL
)

## S3 method for class 'rmbl_aipw'
plot(
  x,
  ...,
  type = c("overlap", "estimate"),
  main = NULL,
  col = NULL,
  palette = NULL
)

## S3 method for class 'rmbl_stock_flow'
plot(
  x,
  ...,
  what = c("adp", "alos", "people", "days"),
  main = NULL,
  col = NULL,
  palette = NULL,
  xlab = NULL
)

## S3 method for class 'rmbl_yoy'
```

```
plot(
  x,
  ...,
  what = c("pctchg", "value"),
  main = NULL,
  col = NULL,
  palette = NULL,
  xlab = NULL,
  gaps = c("break", "join")
)

## S3 method for class 'rmbl_rate'
plot(x, ..., period = NULL, main = NULL,
  col = NULL, palette = NULL, xlab = NULL)

## S3 method for class 'rmbl_share'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_rate_change'
plot(x, ..., main = NULL, col = NULL,
  palette = NULL, xlab = NULL)

## S3 method for class 'rmbl_band_sensitivity'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_drift_calibration'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'bricklayer_drift'
plot(
  x,
  ...,
  what = c("psi", "columns", "benford", "ecdf", "quantiles", "fingerprint"),
  cols = NULL,
  warn = NULL,
  fail = NULL,
  main = NULL,
  col = NULL,
  palette = NULL
)

## S3 method for class 'bricklayer_benford'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'bricklayer_power'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'bricklayer_freq'
```

```
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'bricklayer_missingness'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_region_coverage'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'bricklayer_outliers'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'bricklayer_correlations'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'bricklayer_cortable'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'bricklayer_falsification'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_bounds'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_crf'
plot(x, ..., exposure = NULL, main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_burden'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_equity'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_pollution_report'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_footprint'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_field'
plot(x, ..., contours = 8L, main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_particles'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_count_trend'
plot(x, ..., main = NULL, col = NULL,
  palette = NULL, xlab = NULL)
```

```
## S3 method for class 'bricklayer_analysis'
plot(x, ..., which = NULL, main = NULL)

## S3 method for class 'rmbl_event_study'
plot(
  x,
  ...,
  main = NULL,
  col = NULL,
  palette = NULL,
  style = c("dot", "stata"),
  shadepost = FALSE,
  refline = -0.5,
  xlab = NULL,
  gaps = c("break", "join")
)

## S3 method for class 'rmbl_staggered_att'
plot(
  x,
  ...,
  main = NULL,
  col = NULL,
  palette = NULL,
  style = c("dot", "stata"),
  shadepost = FALSE,
  refline = -0.5
)

## S3 method for class 'rmbl_synth'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_bacon'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_permutation'
plot(
  x,
  ...,
  main = NULL,
  col = NULL,
  palette = NULL,
  type = c("density", "histogram")
)

## S3 method for class 'rmbl_loo'
plot(x, ..., main = NULL, col = NULL, palette = NULL)
```

```
## S3 method for class 'rmbl_cohort_trends'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_ltrlrng'
plot(x, ..., equiv = FALSE, spacing = NULL,
  main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_srchpod'
plot(x, ..., models = NULL, m = NULL,
  cmax = NULL, main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_datummap'
plot(x, ...,
  view = c("map", "persp", "both", "cells"), contours = 8L, digits = 2,
  main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_opteffort'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_svy_estimate'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_svy_compare'
plot(x, ..., main = NULL, col = NULL, palette = NULL)

## S3 method for class 'rmbl_trc_gaps'
plot(x, ..., main = NULL, col = NULL, palette = NULL)
```

Arguments

x	An rmbl_stock_flow result.
...	Passed to the base-graphics call (plot, barplot or image). They are passed on to each panel's method for a bricklayer_analysis object.
component	For rmbl_census_cwb: "cwb" (the index, default) or one of its components "income", "education", "housing", "labour".
main	Title; each method has a default.
col	Colours, overriding the palette.
palette	A blpal name, overriding the session option.
what	Depends on the class: • rmbl_indigenous_gap: "difference" or "ratio"; • rmbl_stock_flow: the measures to draw (one panel each), any of "adp", "alos", "people", "days", "flowrate", "stkrate"; • rmbl_yoy: "pctchg" or "value"; • bricklayer_drift: "psi", "columns", "benford", "ecdf", "quantiles" or "fingerprint"; • rmbl_case_flow: "flow" and/or "clearance";

- rmbl_clearance_decomposition: any of "rates", "mix" and "effects" (the default fits the device).

The 0.5.x values "pct_change", "flow_rate" and "stock_rate" are still accepted, as is "rate_change" for which.

points — For rmbl_density, whether to draw the incidents over the surface.

maxparts — For rmbl_clearance_decomposition, the most parts drawn: past it, the parts that moved the aggregate most, and the rest summed into one row. The 0.5.x name max_parts is still accepted.

threshold — For rmbl_balance: the absolute standardised difference drawn as the line of acceptable balance (default 0.1).

after — For rmbl_balance: optionally a second baltab on the same covariates (after weighting or matching), drawn beside the first.

type — For a permutation test: "density" (a kernel density of the permuted statistics, their 5th and 95th percentiles dashed, the observed one in red; the default) or "histogram". For a cross-fitted AIPW result: "overlap" (the weighted distribution of the out-of-fold propensity in each arm with the clipping bounds; the default) or "estimate" (the effect with its confidence interval).

xlab — For the plots over time (rmbl_stock_flow, rmbl_yoy, rmbl_rate_change, rmbl_count_trend, rmbl_event_study, and rmbl_rate with period), the title of the period axis. The default NULL names the unit the period implies: "Date" for dates, "Time" for date-times, "Fiscal year" for fiscal-year labels such as "2022/23", "Periods since adoption" for an event study, otherwise "Period". A number or a date is drawn on its own continuous scale, so a missing period leaves its gap; any other period (text, a factor) is drawn at evenly spaced positions labelled with its text.

gaps — For rmbl_yoy (what = "value") and rmbl_event_study (style = "stata"): how a missing value shows in a line. "break" (the default) breaks the line there, so a "no data" period that yoy(complete = TRUE) inserted is a visible gap; "join" draws the line straight over it.

period — For rmbl_rate, a by column holding the period: the rate is drawn over time with its interval band, one panel per group of the other by columns.

cols — For bricklayer_drift, the columns to draw with what = "columns" or "benford"; NULL (the default) takes up to six, worst first.

warn, fail — For bricklayer_drift, the PSI bands: a column is warn above the first and fail above the second (or when its test rejects at alpha). NULL takes the object's bands.

exposure — For rmbl_crf and rmbl_burden, the exposure range to draw the curve over; the default runs from 0 to 1.5 times the exposure in the object.

contours — Number of contour lines over a field image (0 for none).

which — For bricklayer_analysis, the panels: any of "change", "rates", "ratechg"; NULL draws every panel the analysis has.

style — For an event study: "dot" (filled points with solid capped bars, the default) or "stata" (hollow points joined by a line, dashed capped bars, dashed reference lines).

shadepost	Shade the periods after adoption (default FALSE).
refline	Where the vertical reference line goes (default -0.5, between the reference period and adoption; 0 puts it at adoption).
equiv	Whether to draw the equivalent definite-range rectangle and the A and B areas (figure 4-7 of the book).
spacing	NULL, or the distance to a second track, drawn beside the first.
models	Model codes ("random", "definite", "invcube", "mbeta"), one curve each; NULL draws the fitted model.
m	M-Beta heights, one curve each when "mbeta" is among the models.
cmax	The largest coverage drawn.
view	"map" (the top view), "persp" (the surface), "both" (side by side, in the book's colour bands) or "cells" (the probability map as a grid of percentages).
digits	Decimal places of the percentages in the "cells" view.

Details

rmbl_stock_flow (stkflow) the average daily population, length of stay, people and person-days over the periods, one panel per measure in what; with an exposure, the flow and stock rates too.

rmbl_yoy (yoy) the period-over-period percent change with its interval over the periods, the period across and the change up, groups side by side, coloured by verdict; what = "value" draws the values instead.

rmbl_rate, rmbl_share (rtcalc, shrs) the rate or share per group with its interval.

rmbl_rate_change (ratechg) the percent change in the rate with its interval over the periods, groups side by side.

rmbl_band_sensitivity (bandsens) the statistic against the cap applied to the open band.

rmbl_drift_calibration, bricklayer_drift (drftcal, cpsdrft) the false-alarm rate per column against alpha; for a drift result, what = "psi" (the default) the population stability index per column, worst first, coloured by verdict, with the warn and fail bands as dashed lines; "columns" the reference and current distributions of up to six columns (cols, or the worst), each PSI bin's term written on it; "benford" the first digits of numeric columns against Benford's law with its 95% band, and the test's verdict; "ecdf" both ECDFs with the area between them shaded; "quantiles" the current quantiles against the reference's on the identity line; "fingerprint" one spoke per column, its PSI as the distance out, with the bands as rings.

bricklayer_benford (benfordtst) observed first-digit proportions against Benford's law.

bricklayer_power (cpspwr) the power curve: detection rate against injected effect size, with the 0.8 line.

bricklayer_freq, bricklayer_missingness, rmbl_region_coverage (freqtbl, misspat and rgncov) horizontal bars.

bricklayer_outliers (mahaoutl) the distance of every row, outliers marked.

bricklayer_correlations, bricklayer_cortable (topcors, cortbl) the correlation of every pair as a diverging bar.

`bricklayer_falsification` (`cpsfals`) the permutation null distribution with the observed statistic.

`rmbl_bounds` (`pubbnd`) each value inside its published-rounding envelope.

`rmbl_indigenous_gap`, `rmbl_dsr`, `rmbl_prevalence` each group's difference or ratio, rate, percentage, odds ratio or index with its interval, for these classes and for `rmbl_exposure_trend` and `rmbl_community_index` (`indgap`, `agestd`, `stdprev`, `exptrnd`, `commidx`); `what = "ratio"` draws the ratio of a gap.

`rmbl_trc_progress` (`trcprog`) the Calls to Action completed, and the pace projected with the band its interval gives.

`rmbl_ordinal_compare` (`ordcmp`) each group's ratings as one stacked bar, lowest rating on the left.

`rmbl_wellness_profile`, `rmbl_wellness_continuum` wellness indicators above, distress below, each group's estimate and interval (`welprof`, `mhcont`).

`rmbl_dual_continuum` (`dualcont`) each group on the two axes, wellness up and distress across.

`rmbl_holistic_balance` (`holbal`) the domains as spokes and each group's levels on them, with its balance.

`rmbl_distinctions` (`dstntst`) each group's estimate and the pooled figure, with I-squared.

`rmbl_connection` (`connassoc`) the places on the two indicators, with their intervals; `xlab` and `ylab` name the axes (default: the indicators' column names).

`rmbl_healing_change` (`healchg`) the change in each indicator between the periods, wellness first.

`rmbl_profile_share` (`profshr`) each group's share of a census characteristic with its interval.

`rmbl_crf`, `rmbl_burden`, `rmbl_equity` and `rmbl_pollution_report`: in turn, the concentration-response curve with the exposure marked; baseline against attributable cases; the concentration curve of exposure over income rank; and, for a report, all three on one page (`crfpm25`, `polbrdn`, `expconcidx`, `vrfpol`).

`rmbl_footprint` (`co2fp`) the CPU, memory and total emission in one unit chosen to fit, with the car-kilometre and tree-month equivalents as a caption; "Too short to measure" when the work took no measurable time.

`rmbl_field`, `rmbl_particles` (`plmfld`, `advdif2d`, `lagrprt`) a concentration field as an image with contours; particle positions, with the counting grid behind them when one was given.

`rmbl_count_trend` (`cnttrnd`) the counts with the fitted trend.

`bricklayer_analysis` (`anltab`) the change, rates and rate change of a published table, one panel each.

`rmbl_event_study`, `rmbl_staggered_att` (`evstudy`, `stgatt`) the effect at each period since adoption with its interval, the reference period at zero.

`rmbl_synth`, `rmbl_synthdid` (`synctl`, `syndid`) the treated unit (or mean) and its synthetic or weighted comparison over time, adoption marked.

`rmbl_power` (`plnpwr`) power against effect size, with the 0.8 line and the minimum detectable effect.

`rmbl_prevalence` (`stdprev`) each group's age-standardised prevalence with its interval.

`rmbl_census_cwb` (`cnscwb`) the Community Well-Being index, or one `component`, by place, one colour per group.

`rmbl_balance` (`baltab`) a love plot: each covariate's absolute standardised difference against the `threshold` line, with a second table (`after`, after weighting or matching) beside it.

`rmbl_aipw` (`aipwcf`) the overlap of the two groups' out-of-fold propensities (`type = "overlap"`) or the estimate with its interval (`type = "estimate"`).

`rmbl_event_study`, `rmbl_staggered_att` also take `style = "stata"` (hollow points joined by a line, dashed bars) and `shadepost`; `plotevgrid` puts several in one grid.

`rmbl_bacon` (`bacondcmp`) each two-group comparison's estimate against its weight, by type, with the two-way estimate.

`rmbl_permutation` (`perminf`) the permuted statistics as a density (5th and 95th percentiles dashed) or histogram, the observed one in red, one panel per outcome.

`rmbl_loo`, `rmbl_cohort_trends` (`loounit`, `cohtrnd`) the leave-one-out estimates with the full-sample one; each cohort's outcome by time since adoption.

`rmbl_case_flow` (`caseflow`) cases received and disposed, shaded where the inventory grew or fell, and the clearance rate with its interval and rolling value.

`rmbl_life_table` (`lifetbl`) the share of cases still open at each band edge with its interval, and the median.

`rmbl_disposition_mix` (`dispmix`) the shares of each outcome over the periods, stacked.

`rmbl_density` (`crmdens`) the density of incidents as a heat map, with the highest-density regions outlined.

`rmbl_knox` (`knoxtst`) the Knox ratio of each distance-by-time cell, coloured on a log scale around 1.

`rmbl_hotspot` (`hotpers`) the share of places holding half the incidents and the overlap of hot spots with the period before, over time, and a map of the cells coloured by how often they were hot.

`rmbl_svy_estimate`, `rmbl_svy_compare` (`svyest`, `svycmp`) each domain's estimate with its interval, flagged domains marked; each difference from the reference domain with its interval.

`rmbl_clearance_decomposition` (`clrdcmp`) the parts' rates in both comparisons with the aggregate on top, where the workload moved (coloured by whether a part clears above or below the average), and the change as a waterfall of rate and mix effects: the pictures that show a Simpson's-paradox reversal and why it happened.

Value

The data drawn, invisibly. For the plots over time the frame carries the x position of each point drawn: an `at` column (`rmbl_yoy`, `rmbl_rate_change`), or an `"at"` attribute (`rmbl_stock_flow`).

Examples

```
sf <- stkflow(
  days = c(13500, 14600, 15100),
  people = c(300, 320, 310),
  prd = 2021:2023
)
plot(sf)
plot(sf, what = "alos", col = "firebrick", main = "Length of stay")
y <- yoy(
  data.frame(year = 2019:2023, n = c(400, 420, 390, 460, 450)),
```

```
  value = "n",
  prd = "year"
)
plot(y)
# small counts: each percentage carries its exact interval
plot(yoy(
  data.frame(year = 2019:2023, n = c(10, 12, 11, 15, 14)),
  value = "n",
  prd = "year"
))
plot(benfordtst(c(1, 12, 123, 1234, 2, 23, 3, 31, 4, 45, 5, 6, 7, 8, 9)))
plot(crfno2(25))
plot(co2fp(
  for (i in 1:20) {
    sum(sqrt(seq_len(2e5)))
  },
  loc = "CA",
  cpupwrw = 45,
  memgb = 8
))
# the courts and crime data shipped with the package
ext <- function(f) {
  read.csv(system.file("extdata", f, package = "rmoriebricklayer"))
}
fl <- ext("ocj_flow_ontario.csv")
plot(caseflow(fl$received, fl$disposed, pend = fl$pending, prd = fl$month))
td <- ext("ocj_time_to_disposition.csv")
open <- unlist(td[nrow(td), 2:5]) * fl$disposed[nrow(fl)]
plot(lifetbl(c(0, 10, 15, 18, Inf), open))
dm <- ext("ocj_dispositions.csv")
plot(dispmix(dm[, -(1:2)], prd = dm$year))
og <- ext("ocj_offence_groups.csv")
og <- og[og$year %in% c(2014, 2022), ]
plot(clrdcmp(og$disposed, og$received, og$offence_group, og$year))
be <- ext("tps_break_enter_d14_2024.csv")
be <- be[!is.na(be$lon), ]
plot(crmdens(be$lon, be$lat, ngrid = 60))
plot(knoxtst(
  be$lon,
  be$lat,
  as.Date(be$occ_date),
  c(100, 200, 400),
  c(7, 14, 28),
  nsim = 99,
  seed = 1
))
plot(hotpers(be$lon, be$lat, quarters(as.Date(be$occ_date)), cell = 0.3))
# rates, shares and the change in a rate
rt <- data.frame(
  division = rep(c("n", "e", "w"), 2),
  year = rep(2022:2023, each = 3),
  stops = c(120, 90, 60, 150, 85, 70),
  residents = rep(c(1e4, 8e3, 5e3), 2)
```

```
)
r23 <- rt[rt$year == 2023, ]
plot(rtcalc(r23, count = "stops", pop = "residents", by = "division"))
plot(shrs(r23, count = "stops", by = "division"))
plot(ratechg(
  rt,
  count = "stops",
  pop = "residents",
  prd = "year",
  by = "division"
))
# a rate over time, one panel per group
plot(
  rtcalc(rt, count = "stops", pop = "residents", by = c("division", "year")),
  period = "year"
)
# complex survey estimates by domain, and the differences between domains
set.seed(3)
sv <- data.frame(
  region = rep(c("north", "south", "east"), length.out = 300),
  smoker = rbinom(300, 1, 0.25),
  w = runif(300, 50, 150)
)
sd <- svydes(sv, w = "w", strata = "region")
plot(svyest(
  sd,
  sv,
  "smoker",
  by = "region",
  stat = "proportion",
  lvls = "1"
))
plot(svycmp(
  sd,
  sv,
  "smoker",
  by = "region",
  stat = "proportion",
  lvls = "1"
))
# a published table in one call: change, rates and rate change
tab <- data.frame(
  year = rep(2020:2023, each = 2),
  g = rep(c("a", "b"), 4),
  n = c(10, 20, 12, 22, 11, 25, 15, 24),
  pop = rep(c(1000, 2000), 4)
)
plot(anltab(tab, value = "n", prd = "year", by = "g", pop = "pop"))
plot(cnttrnd(c(21000, 23000, 22000, 25000, 24000)))
# banded tables
plot(bandsens(
  prsbands(c("0-17", "18-24", "25-44", "45+")),
  c(10, 20, 30, 40),
```

```
  caps = c(60, 70, 80, 90)
))
plot(pubbnd(c(10, 20), rnd = 5))
plot(rgncov(c("a", "b", "c"), c(100, 200, 300), c(1L, 0L, 2L)))
# drift between two extracts, and how often the test cries wolf
set.seed(1)
dd <- data.frame(a = rnorm(400), b = sample(letters[1:3], 400, TRUE))
dr <- cpsdrft(dd, transform(dd, a = a + 3))
plot(dr)
plot(dr, what = "columns")
plot(dr, what = "benford", warn = 0.05, fail = 0.2)
plot(dr, what = "ecdf")
plot(dr, what = "quantiles")
plot(dr, what = "fingerprint")
plot(drftcal(dd, n = 20L))
# describing a table
plot(freqtbl(data.frame(g = c("a", "b", "b", "c", "c", "c")), "g"))
plot(misspat(data.frame(a = c(1, NA, 3, NA), b = c(NA, NA, 1, 2))))
hw <- data.frame(h = rnorm(100), w = rnorm(100))
hw$h[1] <- 15
plot(mahaoutl(hw))
sq <- data.frame(a = 1:20, b = (1:20)^2, c = rev(1:20))
plot(topcors(sq))
plot(cortbl(sq))
# power and falsification of a statistic
xy <- data.frame(x = rnorm(60), y = rnorm(60))
shift <- function(z, s) {
  z$y <- z$y + s * z$x
  z
}
plot(cpspwr(
  xy,
  function(z) cor(z$x, z$y),
  inject = shift,
  trt = "x",
  sizes = c(0, 0.25, 0.5),
  n = 19,
  reps = 3,
  seed = 1
))
plot(cpsfals(xy, function(z) cor(z$x, z$y), trt = "x", n = 29, seed = 1))
# air pollution: burden, equity, and the full report
plot(polbrdn(25, 1, 0.008, 1e6, poll = "NO2"))
plot(expconcidx(
  data.frame(exposure = c(4, 1, 8, 2), income = c(2, 1, 4, 3)),
  "exposure",
  "income"
))
plot(vrfpol("no2", demo = TRUE))
plot(lagrprt(
  200,
  0,
  0,
```

```
  1,
  0.5,
  1,
  1,
  1,
  20,
  grid = c(-10, 40, -15, 25, 10, 8)
))
plot(plmfld(
  q = 100,
  u = 5,
  h = 50,
  x = seq(100, 2000, by = 100),
  y = seq(-300, 300, by = 50)
))
```

`plot.rmbl_dstfit` *Plot fitted distributions*

Description

Four views of a `dstfit()` result: the histogram (or, for counts, the observed proportions) with each fitted density or mass over it; a QQ plot and a PP plot per family; and the empirical survival function on log-log axes with each fitted survival curve, where a power law is a straight line.

Usage

```
## S3 method for class 'rmbl_dstfit'
plot(
  x,
  ...,
  which = c("density", "qq", "pp", "tail"),
  fams = NULL,
  main = NULL,
  col = NULL,
  palette = NULL
)
```

Arguments

`x`	A `dstfit()` result.
`...`	Further graphical arguments to the first panel's `plot()` call.
`which`	One of `"density"`, `"qq"`, `"pp"` or `"tail"`.
`fams`	Families to draw; default the three best by AIC.
`main`, `col`, `palette`	The title, colours, and a `blpal()` palette name.

Value

The data drawn, invisibly: a data frame with `family`, `x` and `y` (and for `qq` and `pp` the empirical value in `y` against the fitted one in `x`).

Examples

```
set.seed(2)
f <- dstfit(stats::rlnorm(200), nboot = 0)
plot(f)
plot(f, which = "qq")
plot(f, which = "pp", fams = "lognormal")
plot(f, which = "tail")
try(plot(f, fams = "beta"))
```

`plotevgrid` *Event studies for several outcomes in one grid*

Description

Draws a list of `evstudy` (or `stgatt`) results in a grid of panels, one per outcome, with the same style, the way papers show an effect on several crimes or several groups at once.

Usage

```
plotevgrid(x, ncol = NULL, main = NULL, ...)
```

Arguments

`x`	A named list of `evstudy` or `stgatt` results; the names title the panels.
`ncol`	Panels per row (default up to 3).
`main`	Optional overall title.
`...`	Passed to each panel's `plot()` (for example `style = "stata"`, `shadepost = TRUE`, `refline = 0`).

Value

Invisibly, a data frame of every panel's estimates with a column `panel`.

See Also

`evstudy`, `plot-methods`.

Examples

```
p <- polex()
es <- list(
  "Crime rate" = evstudy(p, "district", "month", "crmrate", "refmonth",
                         leads = 3, lags = 4),
  "Arrest rate" = evstudy(p, "district", "month", "arrrate", "refmonth",
                          leads = 3, lags = 4))
plotevgrid(es, style = "stata")

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
plotevgrid(list(
  "Rate" = evstudy(own, "town", "year", "rate", "first", leads = 3,
                   lags = 3),
  "Income" = evstudy(own, "town", "year", "income", "first", leads = 3,
                     lags = 3)
))
```

`plothwk` *Plot a fitted Hawkes process*

Description

Three views of a `chwkfit()` result against its events:

`"intensity"` the conditional intensity $\lambda(t) = \nu(t) + \eta \sum_{t_j<t} g(t - t_j)$ over the window, the baseline dashed and the events as ticks along the axis.

`"qq"` the time-rescaling residuals of `chwkres()` against the uniform quantiles: a correct model puts them on the diagonal (the Kolmogorov-Smirnov p-value of the fit in the title).

`"tree"` the most probable branching structure: each event drawn at its time and generation, linked to the earlier event (or the baseline) most likely to have triggered it, the probabilities $\eta g(t_i - t_j)$ and $\nu(t_i)$ in proportion. At most 5,000 events.

Usage

```
plothwk(
  fit,
  times,
  what = c("intensity", "qq", "tree"),
  ngrid = 600L,
  ...,
  main = NULL,
  col = NULL,
  pal = NULL
)
```

Arguments

fit	A chwkfit() result.
times	The event times the fit was made on.
what	One of "intensity", "qq" or "tree".
ngrid	Whole number, default 600: points of the intensity curve.
...	Further arguments to the plot() call that opens the panel.
main, col, pal	The title, colours, and a blpal() name.

Value

The data drawn, invisibly: t and lambda (intensity), uniform and residual (a Q-Q plot), or t, parent (the index of the parent event, 0 for the baseline), prob and generation (tree).

Examples

```
set.seed(1)
times <- sort(stats::runif(200, 0, 100))
fit <- chwkfit(times, 100)
plothwk(fit, times)
plothwk(fit, times, "qq")
tr <- plothwk(fit, times, "tree")
table(tr$generation)
```

---

plotlrnz *Lorenz and funnel plots*

Description

Two plots for results that are plain data frames rather than classed objects. plotlrnz() draws the Lorenz curve of a vector (or of lorenzc's output) with the Gini coefficient in the title. plotfnl() draws every area's ratio of observed to expected against its expected count, inside the control limits of fnllim, the standard small-area funnel plot. Use the Lorenz plot to show how concentrated a total is across units, and the funnel plot to judge which areas' rates differ from the target by more than chance would allow given their size.

Usage

```
plotlrnz(x, ..., main = NULL, col = NULL, pal = NULL)

plotfnl(
  obs,
  expd,
  area = NULL,
  lvls = c(0.95, 0.998),
  target = 1,
  ...,
  main = NULL,
  col = NULL,
  pal = NULL
)
```

Arguments

x	For `plotlrnz()`, a numeric vector of non-negative values (one per unit), or the data frame `lorenzc()` returns (columns `population` and `value`).
main, col, pal, ...	As in `plot-methods`: the title, colours overriding the palette, a `blpal` name, and arguments passed to `plot()`.
obs, expd	Numeric vectors of observed and expected counts per area, the same length; `observed` non-negative, `expected` positive.
area	Optional vector of labels, one per area, drawn for the areas outside the limits. Default `NULL`.
lvls	Numeric vector of control-limit levels strictly between 0 and 1, as in `fnllim`; default `c(0.95, 0.998)` (about two and three standard deviations).
target	Numeric scalar, the ratio the limits are drawn around; default `1` (observed equals expected).

Details

`plotlrnz()`: from a numeric vector the curve is computed with `lorenzc`, and the title carries `ginicoef`; a data frame is used as given (sorted, with the origin added when missing) and the title has no Gini. The dashed diagonal is perfect equality.

`plotfnl()`: each area's ratio observed / expected is drawn against its expected count. The limits are exact Poisson quantiles around `target`, computed by `fnllim` at the expected counts present: the widest level is drawn solid, the others dashed. A point outside the widest limits is coloured and, with `area`, labelled.

Errors: `plotlrnz()` given a data frame whose columns are not cumulative shares in $[0, 1]$ that never decrease, or invalid values for `lorenzc`; `plotfnl()` given negative or non-finite counts, an expected count of zero or less, `observed` and `expected` of different lengths, `area` of another length, `levels` outside (0, 1), or a `target` that is not one positive number. Both draw on the current device.

Value

The data drawn, invisibly. `plotlrnz()`: a data frame with `population` and `value`, starting at the origin. `plotfnl()`: a data frame with `area`, `observed`, `expected`, `ratio` and `outside` (logical, outside the widest limits).

Examples

```
plotlrnz(c(1, 1, 2, 3, 5, 8, 13))
plotfnl(obs = c(4, 20, 50, 9), expd = c(5, 22, 40, 20),
            area = c("a", "b", "c", "d"))

# a funnel with one area well above expectation, around a target of 1
set.seed(4)
e <- c(5, 10, 20, 40, 80, 160)
o <- rpois(6, e)
o[5] <- 130
fd <- plotfnl(o, e, area = LETTERS[1:6], lvls = 0.95)
fd[fd$outside, ]

# a curve from lorenz(), styled
plotlrnz(lorenzc(c(rep(1, 50), rep(10, 5), 200)), col = "darkred",
            main = "Placements per person")

# a hand-made frame must be cumulative shares
try(plotlrnz(data.frame(population = c(0, 0.5, 1), value = c(0, 0.8, 0.5))))
```

`plotmore` *Many-plot helpers: ridgelines, ECDFs, violins, hexagonal bins and concentration curves*

Description

Base-graphics plots in the house style for a numeric variable split by a group, or two numeric variables:

`plotrdg()` a ridgeline: one kernel density per group, stacked with a small overlap.

`plotecdf()` the empirical distribution function of each group, overlaid.

`plotvln()` a violin per group: the mirrored density, the median as a point and the interquartile range as a bar.

`plothex()` the two-dimensional density of `x` and `y` in hexagonal bins, shaded by count.

`plotgain()` cumulative-gain (concentration) curves: the Lorenz curve of each group (`lorenzc()`) with its Gini coefficient in the key.

Missing and non-finite values are dropped. A group of fewer than two values draws no density.

Usage

```
plotrdg(x, grp = NULL, bw = "nrd0", ..., main = NULL, col = NULL, pal = NULL)

plotecdf(x, grp = NULL, ..., main = NULL, col = NULL, pal = NULL)

plotvln(x, grp = NULL, bw = "nrd0", ..., main = NULL, col = NULL, pal = NULL)

plothex(x, y, bins = 30L, ..., main = NULL, col = NULL, pal = NULL)

plotgain(x, grp = NULL, ..., main = NULL, col = NULL, pal = NULL)
```

Arguments

x	Numeric vector to plot, one value per observation.
grp	Optional vector of groups, one per value of x.
bw	Bandwidth passed to stats::density(): one positive number or the name of a rule ("nrd0", the default, "nrd", "ucv", "bcv", "sj", "sj-ste", "sj-dpi").
...	Further arguments to the plot() call that opens each frame.
main, col, pal	The title, colours overriding the palette, and a blpal() name.
y	Numeric vector, one per value of x (plothex()).
bins	Whole number, default 30: hexagons across the range of x.

Value

The data drawn, invisibly: for plotrdg() and plotvln() the densities (grp, x, y), for plotvln() with the per-group median, q1 and q3 as an attribute summary, for plotecdf() the steps (grp, x, y), for plothex() the hexagon centres (x, y) and count, for plotgain() the curves (grp, population, value) with the Gini coefficients as an attribute gini.

Examples

```
set.seed(3)
x <- c(stats::rnorm(200), stats::rnorm(200, 2, 0.6))
g <- rep(c("before", "after"), each = 200)
plotrdg(x, g)
plotecdf(x, g)
plotvln(x, g)
plothex(x, x + stats::rnorm(400), bins = 20)
plotgain(stats::rexp(400), g)
try(plotrdg("a"))
```

`plotseries` *Plots for many series over time*

Description

Base-graphics plots in the house style for a value observed by period, for one series or many:

`plotcal()` a calendar heatmap of a daily series: one tile per day, weeks across and weekdays down, one block per year.

`plothrz()` a horizon chart: each series in its own thin row, its deviation from its own median folded into `bands` shaded bands, above in blue and below in red, so many series fit in little height.

`plotsm()` small multiples: one panel per series on a shared scale.

`plotbump()` a bump chart: the rank of each series in each period (1 = largest), so changes of order stand out.

Usage

```
plotcal(x, y, ..., main = NULL, col = NULL, pal = NULL)

plothrz(
  y,
  prd,
  grp = NULL,
  bands = 3L,
  ...,
  main = NULL,
  col = NULL,
  pal = NULL
)

plotsm(y, prd, grp = NULL, ..., main = NULL, col = NULL, pal = NULL)

plotbump(y, prd, grp, desc = TRUE, ..., main = NULL, col = NULL, pal = NULL)
```

Arguments

`x`	Dates (a `Date` or text `as.Date()` reads) for `plotcal()`.
`y`	Numeric vector of values.
`...`	Further arguments to the `plot()` call that opens the frame.
`main`, `col`, `pal`	The title, colours overriding the palette, and a `blpal()` name.
`prd`	Vector of periods, one per value (numbers, dates or text in order of appearance).
`grp`	Vector of series names, one per value, or `NULL` for one series.
`bands`	Whole number, default 3: the bands of a horizon chart.
`desc`	Logical, default `TRUE`: rank 1 is the largest value (`FALSE`: the smallest).

Value

The data drawn, invisibly: for `plotcal()` the days (`date`, `year`, `week`, `wday`, `y`), for the others the series (`grp`, `prd`, `y`, and `rank` for `plotbump()`, `band` for `plothrz()`).

Examples

```
set.seed(5)
days <- seq(as.Date("2024-01-01"), as.Date("2024-12-31"), by = "day")
plotcal(days, stats::rpois(length(days), 5 + 3 * (format(days, "%u") > "5")))
d <- expand.grid(
  year = 2015:2024,
  area = c("north", "south", "east", "west")
)
d$n <- stats::rpois(
  nrow(d),
  50 + 10 * as.integer(d$area) + 2 * (d$year - 2015)
)
plotsm(d$n, d$year, d$area)
plothrz(d$n, d$year, d$area)
plotbump(d$n, d$year, d$area)
try(plotcal("not a date", 1))
```

---

plrcf *Cross-fitted partially linear regression (double machine learning)*

Description

Estimates the effect of a continuous exposure (or treatment) on an outcome after adjusting for confounders, by the double machine learning recipe: the part of the outcome and of the exposure that the confounders predict is removed with models fitted on other folds of the data, and the effect is the slope of what is left. It answers questions such as "how much does asthma incidence rise per microgram of NO2, holding income and age structure fixed?".

Usage

```
plrcf(data, y, trt, covs, kfol = 5L, seed = 1L)
```

Arguments

`data`	A data frame with one row per unit.
`y`, `trt`	Column names (single strings) of the outcome and of the continuous treatment or exposure; both numeric.
`covs`	Column names of the confounders (one or more strings), numeric. Code a categorical confounder as indicator columns first.
`kfol`	Number of cross-fitting folds, a whole number of at least 2 (default 5).
`seed`	Seed of the fold assignment (default 1), so the estimate is reproducible.

Details

The partially linear model $Y = \theta D + g(X) + U$, $D = m(X) + V$ of Chernozhukov et al. (2018, section 4.2), estimated by the DML2 cross-fitting recipe: the sample is split into `kfol` folds; for each fold the nuisances m and g are fitted on the other folds and predicted on it; $\hat{\theta}$ is the slope of the out-of-fold outcome residuals on the out-of-fold treatment residuals,

$$\hat{\theta} = \sum_i \hat{V}_i \hat{U}_i / \sum_i \hat{V}_i^2,$$

with the sandwich standard error $\sqrt{\sum_i \hat{V}_i^2 (\hat{U}_i - \hat{\theta}\hat{V}_i)^2} / \sum_i \hat{V}_i^2$. The nuisances here are ordinary least squares on the covariates, which is the right learner when the confounding is approximately linear and keeps the estimator fully deterministic given `seed`; rmorie's `morie_estimate_double_ml()` offers richer learners and can be passed to `exprspplr` through `estimator`.

Each fold's two least-squares fits are computed from the normal equations of the covariates centred and scaled by their full-sample moments (the same column space, so the same fitted values as `lm.fit`), solved by Cholesky; a fold whose cross-product is poorly conditioned (reciprocal condition number below 1e-5: collinear covariates or a constant column) uses `lm.fit` instead, whose pivoting drops the redundant columns (their coefficients count as 0). An n = 20000 fit takes a few milliseconds. A bar on stderr shows the folds when the work is slow enough to need one (see `blprog`).

Rows with a missing value in any of the named columns are dropped first (`n` reports the rows used). The folds are a random permutation of `rep_len(1:kfol, n)` drawn with `seed`; R's own random stream is restored afterwards. Refusals: a column name that is not a single string (several for `covs`) or is not in `data`; a non-numeric column (a factor is not a code); `kfol` that is not a whole number of at least 2; fewer than `2 * kfol` complete rows; and a treatment with no variation left after the covariates are partialled out.

Value

A list of class `c("rmbl_plr", "list")` (its print method shows it as a box) with `ate` (the estimate of θ: the change in the outcome per unit of the treatment), `se` (its sandwich standard error), `n` (complete rows used), `kfol` and `method` (a string naming the estimator).

References

Chernozhukov, V., Chetverikov, D., Demirer, M., Duflo, E., Hansen, C., Newey, W. and Robins, J. (2018). Double/debiased machine learning for treatment and structural parameters. The Econometrics Journal 21(1), C1-C68.

Examples

```
set.seed(1)
d <- data.frame(x1 = rnorm(200), x2 = rnorm(200))
d$exposure <- 20 + 2 * d$x1 + rnorm(200)
d$asthma <- 5 + 0.3 * d$exposure + d$x2 + rnorm(200)
fit <- plrcf(d, "asthma", "exposure", c("x1", "x2"))
fit$ate

# A 95 percent normal confidence interval from the standard error.
```

```
fit$ate + c(-1.96, 1.96) * fit$se

# The fold split changes the estimate a little; ten folds instead of five.
plrcf(d, "asthma", "exposure", c("x1", "x2"), kfol = 10, seed = 2)$ate

# A text column is refused rather than read as codes.
d$region <- sample(c("north", "south"), 200, replace = TRUE)
try(plrcf(d, "asthma", "exposure", c("x1", "region")))
```

`polbrdarea` *Per-area pollution burden*

Description

Applies `polbrdn` to every row of a table with one row per area (forward sortation area, census tract, neighbourhood, ...) and returns the rows sorted by attributable cases, worst first. Use it to see where in a city or region the pollution burden falls, for example to rank neighbourhoods for an intervention.

Usage

```
polbrdarea(
  areatab,
  areacol = "fsa",
  expcol = "exposure",
  popcol = "population",
  basratecol = "brate",
  poll = "NO2",
  y = "allcause",
  rare = FALSE,
  hirate = FALSE
)
```

Arguments

`areatab`	A data frame with one row per area: a label, the exposure in micrograms per cubic metre, the population in persons and the baseline rate per person-year (0.008 is 800 per 100,000).
`areacol`, `expcol`, `popcol`, `basratecol`	Column names (single strings) of the label, exposure, population and rate; defaults `"fsa"`, `"exposure"`, `"population"` and `"brate"`.
`poll`, `y`, `rare`, `hirate`	Passed to `polbrdn` for every area: the pollutant (default `"NO2"`), the outcome (default all-cause mortality), and the switches that admit a rate below 1e-5 or above 0.1 per person-year.

Details

Each area is treated as wholly exposed at its own concentration (`expoprev = 1`), with the pollutant's default reference concentration. Every area is checked as `polbrdn` checks a single population, and the first area that fails stops the call with that area's error. Refusals: a column name that is not a single string or not in the table, an empty table, and a non-numeric exposure, population or rate column. The area labels can be of any type.

Value

A data frame with one row per area, sorted by `attrcases`, largest first, and row names reset: the four input columns (exposure, population and rate as numbers) plus `rr` (relative risk), `paf` (attributable fraction), `attrcases` and `basecases` (per year).

See Also

`expconcidx` for how the burden is spread across income.

Examples

```
t <- data.frame(fsa = c("M6H", "M5V"), exposure = c(28, 18),
                population = c(40000, 60000), brate = 0.008)
polbrdarea(t)

# PM2.5 and different column names.
u <- data.frame(tract = c("A", "B", "C"), pm = c(7, 11, 9),
                pop = c(5000, 8000, 6500), rate = 0.0075)
polbrdarea(u, areacol = "tract", expcol = "pm",
                          popcol = "pop", basratecol = "rate",
                          poll = "PM2.5")

# A missing column is named in the error.
try(polbrdarea(t[, c("fsa", "exposure")]))
```

`polbrdn` *End-to-end pollution burden*

Description

Chains the concentration-response function, the attributable fraction and the baseline case count into annual attributable cases, as the GBD risk factor estimates do (GBD 2019 Risk Factors Collaborators 2020). Use it for a single population (a city, a region) when you know its mean exposure, its baseline rate and its size; for many areas at once see `polbrdarea`.

Usage

```
polbrdn(
  expomean,
  expoprev,
  brate,
  pop,
  poll = "NO2",
  y = "allcause",
  refconc = NULL,
  rare = FALSE,
  hirate = FALSE
)
```

Arguments

expomean	The mean exposure among the exposed, in micrograms per cubic metre. With expoprev = 1 that is the population mean.
expoprev	Proportion of the population at that level (1 for ambient air), in [0, 1].
brate	Cases per person-year in the unexposed scenario (a rate per person, not per 100,000: 0.008 is 800 per 100,000).
pop	At-risk population (persons; any non-negative number).
poll	"PM2.5" (also written "PM25") or "NO2", in either case.
y	Outcome key passed to the concentration-response function; extra$unit says whether the attributable count is deaths or incident cases.
refconc	Counterfactual concentration; NULL takes the pollutant's default (PM2.5 5.8, NO2 10; see crfpm25).
rare, hirate	TRUE to admit a baseline rate below 1e-5 (one per 100,000) or above 0.1 (10,000 per 100,000) per person-year, as vrfpol does: without them such a rate is refused, being also what a rate in the wrong unit looks like.

Details

The steps are: the relative risk at expomean from crfpm25 or crfno2 (with the reference concentration given or the pollutant's default); the attributable fraction from attrfrac with expoprev; the baseline cases $y_0 N$; and the attributable cases, their product. The result is a yearly count of deaths (or incident cases for "chiast").

Refusals: an unknown pollutant, an outcome the pollutant's CRF does not have, any input that is not a single finite non-negative number, and a baseline rate outside the band described in mortdisp unless rare or hirate admits it.

Value

A list of class rmbl_burden (plot() draws it) with: paf (the attributable fraction), attrcases and basecases (per year), population, brate (per person-year), expomean, refconc (the one used), pollutant ("PM2.5" or "NO2"), citation (a string), and extra, a list with rr, logrr, outcome and unit ("deaths" or "incident cases").

References

GBD 2019 Risk Factors Collaborators (2020). Global burden of 87 risk factors in 204 countries and territories, 1990-2019. The Lancet 396(10258), 1223-1249.

Examples

```
# Ambient NO2 of 25 micrograms per cubic metre over a million people.
b <- polbrdn(25, 1, 0.008, 1e6, poll = "NO2")
b$attrcases
b$paf

# PM2.5 and ischaemic heart disease deaths, against the WHO guideline
# of 5 as the counterfactual.
polbrdn(12, 1, 0.0012, 2.8e6, poll = "PM2.5", y = "ihd",
                 refconc = 5)$attrcases

# Childhood asthma counts incident cases, not deaths.
polbrdn(30, 1, 0.012, 5e5, y = "chiast")$extra$unit

# Only PM2.5 and NO2 have concentration-response functions here.
try(polbrdn(40, 1, 0.008, 1e6, poll = "O3"))
```

---

polex *A simulated policy panel for the examples*

Description

Twenty districts in four regions over 36 months; a reform adopted region by region in three stages (months 13, 19 and 25), with the fourth region never adopting it; crime and arrest rates and counts. The numbers are made up, with a known effect of the reform (a fall of 3 in the crime rate per 100,000), so the examples and tests can check that each estimator recovers it.

Usage

```
polex(seed = 20240101)
```

Arguments

seed	The seed (default 20240101).

Value

A data frame: `district`, `region`, `month`, `refmonth` (`NA` for never), `reformed` (0 or 1), `population`, `logpop`, `crimes`, `crmrate`, `arrrate`.

Examples

```
head(polex())
```

pqcbknd *Available post-quantum signature schemes*

Description

Reports the signature schemes this package implements. The list is fixed, not probed: all of them are implemented in the package's own C++ and none depends on a system library, so a scheme available on one machine is available on every machine.

Usage

```
pqcbknd()
```

Details

`"xmss-sha256"` is the stateful hash-based scheme of `pqckg()`. The rest are the NIST standards, taken with `fipskg()`: ML-DSA (FIPS 204) at all three parameter sets, and SLH-DSA (FIPS 205) at all twelve – six over SHAKE and six over SHA-2, which are different schemes and not merely different code paths. Use it to see which names `fipskg()`, `fipssz()` and `fipskey()` accept, or to choose a parameter set.

The names follow the standards: `"ML-DSA-44"`, `"ML-DSA-65"` and `"ML-DSA-87"` (NIST security categories 2, 3 and 5), then `"SLH-DSA-SHA2-"` and `"SLH-DSA-SHAKE-"` followed by `128`, `192` or `256` (the security level in bits) and `s` (small signatures, slow signing) or `f` (fast signing, larger signatures). The function takes no arguments, has no side effects and returns the same 16 names in the same order on every platform.

Value

A character vector of 16 scheme names: `"xmss-sha256"` first, then the three ML-DSA sets, the six SLH-DSA SHA-2 sets and the six SLH-DSA SHAKE sets.

See Also

`fipskg()` for the standardised schemes, `pqckg()` for the stateful one.

Examples

```
pqcbknd()

# Every scheme is present in every build.
all(c("xmss-sha256", "ML-DSA-65", "SLH-DSA-SHAKE-128s") %in%
    pqcbknd())

# The SLH-DSA sets with fast signing, and their signature sizes in bytes.
fast <- grep("^SLH-DSA-.*f$", pqcbknd(), value = TRUE)
vapply(fast, function(s) fipssz(s)[["signature"]], integer(1))
```

pqckg *Generate a post-quantum signing key for capsule provenance*

Description

Builds an XMSS key pair: a Merkle tree over `2^height` Winternitz one-time keys, all derived from two 32-byte seeds. Security rests on SHA-256 alone – no lattice assumption, no elliptic curve, nothing Shor's algorithm breaks. The key signs a fixed number of messages, `2^height`, and must be carried forward after each signature (see `xmssadv()`); for a key with no such state use `fipskg()`.

Usage

```
pqckg(height = 10L, skseed = NULL, pubseed = NULL, skprf = NULL)
```

Arguments

`height`	Tree height, 1 to 16 (default 10, i.e. 1024 signatures).
`skseed, pubseed, skprf`	64-character hex seeds (32 bytes each) – the three secrets RFC 8391's private key carries. `skseed` derives the WOTS+ chains, `pubseed` masks the hashes, and `skprf` keys the per-signature randomiser. Omit them and they are drawn from the operating system's CSPRNG via `rndbytes()`, which fails rather than falling back to R's reproducible generator. Supply them ONLY to reproduce a key deterministically in a test – a seed you can guess is a key you can forge.

Details

Key generation walks all `2^height` leaves, so cost doubles with each unit of height. The default 10 gives 1024 signatures and takes a few seconds; heights above about 14 are slow enough to be worth avoiding unless the key really must last that long. A spinner on stderr shows the work after 0.75 s when a person is watching (see `blprog()`), and Ctrl-C stops it.

The construction is single-tree XMSS with SHA-256, n = 32 and Winternitz parameter 16 (RFC 8391; at heights 10 and 16 these are its `XMSS-SHA2_10_256` and `XMSS-SHA2_16_256` parameter sets, and the other heights use the same construction with a smaller or larger tree). Its signatures are byte-identical to the XMSS reference implementation's for the same key material.

Side effect: a key generated from fresh randomness is entered in the package's used-index record (the file described under `xmssadv()`), so that a stale copy of it can never sign an index twice on this machine. A key built from given seeds is NOT entered, and signs only after `xmssregkey()` vouches that it never has.

Refusals: a `height` that is not a whole number from 1 to 16, and a seed that is not 64 hex characters.

Value

A list of class `bricklayer_signing_key`: `root` (the public verification value, 64 hex characters), `pubseed`, `skseed` (SECRET) and `skprf` (SECRET), each 64 hex characters, `height` (an integer), `next_index` (the next unused leaf, 0 for a new key), `capacity` (`2^height`, the number of signatures), and `scheme` (`"xmss-sha256"`).

A height-`h` key signs exactly `2^h` messages

Each signature consumes one leaf, and **signing two different messages with the same leaf index breaks the scheme outright** – between two signatures at one index an adversary can forge a third message. `cpssign()` therefore tracks `next_index` and refuses to reuse one. Do not hand-edit that field, and do not copy a key to two machines that sign independently.

The key and its state

RFC 10033 (Hash-Based Signatures: State and Backup Management) splits an XMSS key into the private key, which never changes, and the state, which records the indices used and must be updated, durably, before each signature is released (its sections 2 and 4). In an R session the state is `next_index` plus the used-index record described under `xmssadv()`. From the command line, `rmbl sign --scheme xmss-sha256 --key-file key.json` makes the key file itself the state: it rewrites the file atomically, with `next_index` advanced and a `used` field, before anything is signed, and signs from the indices it reserved. `--state-file` keeps the state in a file of its own, `--sectors` and `--sector-index` give each signer its own disjoint range of indices (section 5.3), `--reserve` reserves a block in one write (section 5.9), and a key file older than a signature it is seen to have made is refused as a stale copy. See the section XMSS keys of `blcli()`.

A key saved as a CI secret is a fixed copy that every job starts from, so every job signs the same indices: in CI use ML-DSA (`fipskg()`) unless the XMSS state is kept in a file that persists between jobs. NIST SP 800-208 approves XMSS only inside a hardware cryptographic module that never exports the key; this software key is not such a deployment.

Key format

A key made before the RFC 8391 conformance work carries no `skprf` and cannot sign; `cpssign()` raises rather than producing a signature no other implementation could read. Generate a new one.

Security

The implementation is hand-written. Its standardised schemes are checked byte for byte against OpenSSL 3.5 and NIST known-answer vectors in the test suite. Side channels are checked, not assumed: `inst/ctcheck` runs every operation that touches a secret under valgrind memcheck with the secret marked undefined (the ctgrind method), so a branch or a memory address that depends on it is a reported error, and afterwards scans the dead stack for copies of the secret. Both checks run in CI on every change, with GCC and with Clang. Timing is also measured on x86-64 and arm64 hardware (`inst/dudect`) and power leakage is assessed in simulation under the value and the transition models (`inst/tvla`) for the masked ML-KEM and ML-DSA kernels; this scheme is not masked. These are checks of this code on those compilers; the README's security section lists what each check does and does not cover.

See Also

`cpssign()`, `cpsvrf()`, `signpubkey()`

Examples

```
# keep this example's
op <- options(rmoriebricklayer.xmssidxfile = tempfile())
```

```
# one-time index record out of your configuration directory; leave the
# default in use A small key, to keep the example quick.
key <- pqckg(height = 3)
key$capacity # 8 signatures
key$next_index # none used yet

# The public half is what a verifier needs; it carries no secret.
pub <- signpubkey(key)
names(pub)

# Deterministic seeds reproduce the same key -- for tests only. A key built
# from given seeds is not entered in the used-index record (it may have
# signed elsewhere): it signs only after xmssregkey() vouches that it never
# has.
s1 <- paste(rep("11", 32), collapse = "")
s2 <- paste(rep("22", 32), collapse = "")
identical(pqckg(3, s1, s2)$root, pqckg(3, s1, s2)$root)

# The height is limited to 16 (65,536 signatures).
try(pqckg(height = 20))
options(op)
```

`print.bricklayer_attestation` *Printed reports for bricklayer objects*

Description

Human-readable renderings of the objects the package returns. Each leads with the verdict, then the evidence. `format()` returns the lines as a character vector so they can be logged or written to a file; `print()` sends them to the console and returns its argument invisibly.

Usage

```
## S3 method for class 'bricklayer_attestation'
print(x, ...)

## S3 method for class 'bricklayer_bundle'
print(x, ...)

## S3 method for class 'bricklayer_attestation_check'
print(x, ...)

## S3 method for class 'bricklayer_benford'
print(x, ...)

## S3 method for class 'bricklayer_bundle_check'
print(x, ...)
```

```
## S3 method for class 'bricklayer_certificate'
print(x, ...)

## S3 method for class 'bricklayer_certpath_check'
print(x, ...)

## S3 method for class 'bricklayer_chain'
print(x, ...)

## S3 method for class 'bricklayer_drbg'
print(x, ...)

## S3 method for class 'bricklayer_drift'
print(x, ...)

## S3 method for class 'bricklayer_env_diff'
print(x, ...)

## S3 method for class 'bricklayer_falsification'
print(x, ...)

## S3 method for class 'bricklayer_falsify_family'
print(x, ...)

## S3 method for class 'bricklayer_fips_key'
print(x, ...)

## S3 method for class 'bricklayer_fips_public_key'
print(x, ...)

## S3 method for class 'bricklayer_hqc_capsule'
print(x, ...)

## S3 method for class 'bricklayer_hqc_key'
print(x, ...)

## S3 method for class 'bricklayer_hqc_public_key'
print(x, ...)

## S3 method for class 'bricklayer_kem_capsule'
print(x, ...)

## S3 method for class 'bricklayer_kem_key'
print(x, ...)

## S3 method for class 'bricklayer_kem_public_key'
print(x, ...)
```

```
## S3 method for class 'bricklayer_mcar'
print(x, ...)

## S3 method for class 'bricklayer_oqs_key'
print(x, ...)

## S3 method for class 'bricklayer_oqs_public_key'
print(x, ...)

## S3 method for class 'bricklayer_power'
print(x, ...)

## S3 method for class 'bricklayer_prereg'
print(x, ...)

## S3 method for class 'bricklayer_prereg_check'
print(x, ...)

## S3 method for class 'bricklayer_public_key'
print(x, ...)

## S3 method for class 'bricklayer_recode_manifest'
print(x, ...)

## S3 method for class 'bricklayer_recompute'
print(x, ...)

## S3 method for class 'bricklayer_report'
print(x, ...)

## S3 method for class 'bricklayer_schema'
print(x, ...)

## S3 method for class 'bricklayer_signature'
print(x, ...)

## S3 method for class 'bricklayer_signing_key'
print(x, ...)

## S3 method for class 'bricklayer_timestamp'
print(x, ...)

## S3 method for class 'rmbl_bounds'
print(x, ...)

## S3 method for class 'rmbl_fetch'
print(x, ...)
```

```
## S3 method for class 'rmbl_margin_check'
print(x, ...)

## S3 method for class 'rmbl_or_check'
print(x, ...)

## S3 method for class 'rmbl_recode_manifest_check'
print(x, ...)

## S3 method for class 'rmbl_transfer_check'
print(x, ...)

## S3 method for class 'bricklayer_drift'
summary(object, ...)

## S3 method for class 'bricklayer_benford'
summary(object, ...)

## S3 method for class 'bricklayer_report'
summary(object, ...)
```

Arguments

x	The object to render.
...	Ignored, present for S3 consistency.
object	The object to summarise.

Details

Box-drawing characters are used only when the session's encoding can render them; otherwise the same layout is drawn in plain ASCII. Set the environment variable RMBL_ASCII_ONLY to force the ASCII form, which is what to do when capturing output into a fixed-width log.

Value

format() methods return a character vector; print() methods return x invisibly.

Examples

```
# keep this example's
op <- options(rmoriebricklayer.xmssidxfile = tempfile())
# one-time index record out of your configuration directory; leave the
# default in use
set.seed(7)
ref <- data.frame(v = stats::rnorm(200), g = sample(c("a", "b"), 200, TRUE))
cur <- data.frame(
  v = stats::rnorm(200, mean = 1),
  g = sample(c("a", "b"), 200, TRUE)
)
```

```
# The drift report leads with its verdict.
cpsdrft(ref, cur)

# format() gives the same lines for a log file.
head(format(cpsdrft(ref, cur)), 3)

# A Benford screen.
benfordtst(10^stats::runif(500, 0, 5))

# Keys and signatures print without ever showing the secret seed.
key <- pqckg(height = 2)
key
cpssign("a-manifest", key)
options(op)
```

`profcols` *Column report for a data frame*

Description

One row per column: type, missingness, and the summaries that suit the column's type – mean and standard deviation alongside their robust counterparts (median, MAD) for a numeric column, and the number of distinct levels plus the most common one for a categorical column.

Usage

```
profcols(data, qnts = c(0.25, 0.5, 0.75))
```

Arguments

`data`	A data frame whose columns are profiled.
`qnts`	Numeric vector of quantile probabilities to include for numeric columns, one column each named `q` and the percentage (`q25`, `q50`, ...); default the quartiles. `NULL` or an empty vector adds none.

Details

Reporting the classical and robust centres side by side is the point: where they disagree, the column has outliers or a heavy tail, and the mean is not describing it. That is visible in one glance here and in no single number.

How the numbers are computed. A column is numeric when `is.numeric()` is `TRUE`; every other column (character, factor, logical, `Date`, ...) is treated as categorical, and `top` is its most frequent non-missing value as text (ties broken by the order of `sort(table())`). For a numeric column, over its non-missing values: `sd` is the sample standard deviation ($n - 1$), `mad` the median absolute deviation scaled by 1.4826, `skewness` the population moment ratio $m_3/m_2^{3/2}$, `n_outliers` the count outside the Tukey fences (1.5 interquartile ranges beyond the quartiles), and the quantiles are

type 7. Infinite values are counted in `ninf` (and in `nneg`, `n_distinct`) and left out of every other numeric statistic, which are taken over the finite values only: a column with an infinite value keeps a finite mean and range, and a column with no finite value has `NA` statistics. A progress bar is shown for wide tables when `options(morie.progress)` allows it. Errors when `data` is not a data frame, has no columns, or has duplicate or empty column names.

Value

A data frame of class `bricklayer_profile`, one row per column: the type, the missing count and share, the number of distinct values, and for a numeric column the counts of zero, negative and infinite values, the classical and robust centre and spread, the range, the skewness, the count outside the Tukey fences, and an `inlhist()` sketch of its distribution. A categorical column reports its most common value in `top` instead. The columns are `column`, `type` (the first class), `n`, `n_missing`, `pct_missing`, `n_distinct`, `nzero`, `nneg`, `ninf`, `mean`, `sd`, `median`, `mad`, `min`, `max`, `skewness`, `n_outliers`, `hist`, `top` and one per quantile; a statistic that does not apply to the column's type is `NA`. The print method formats the numbers to four significant digits and shows the histogram last.

See Also

`cpsdrft()` to compare two of these, `cmom()` for the underlying kernel.

Examples

```
set.seed(9)
df <- data.frame(
  clean = stats::rnorm(100),
  skewed = c(stats::rnorm(99), 500),
  grade = sample(c("a", "b", "c"), 100, TRUE),
  gappy = c(rep(NA, 10), stats::runif(90))
)

profcols(df)

# The mean and the median agree on `clean` and disagree sharply on
# `skewed`, which is the outlier announcing itself.
p <- profcols(df)
p[p$column %in% c("clean", "skewed"), c("column", "mean", "median")]

# The histogram column shows shape no summary number carries.
p[, c("column", "hist")]

# Zeros, negatives and infinities are counted separately, because
# each breaks a different downstream computation (a log, a square
# root, an average).
p[, c("column", "nzero", "nneg", "ninf")]

# deciles instead of quartiles
names(profcols(df, qnts = c(0.1, 0.9)))

# a table with no columns has nothing to profile
```

```
try(profcols(df[, 0]))
```

`profshr` *The share of a population with a characteristic, from the census profile*

Description

The share of a group with a characteristic of the 2021 Census Indigenous Population Profile (foster children among children, speakers of an Indigenous language, households needing major repairs, ...), with an approximate confidence interval that carries the long-form sample and the random rounding of the counts.

Usage

```
profshr(
  x,
  num,
  den = NULL,
  smpfrac = 0.25,
  deff = 1,
  conflvl = 0.95,
  roubase = 5
)
```

Arguments

`x`	A `cnsprof`.
`num`	The characteristic counted: its `chaid`, or its name exactly.
`den`	The characteristic it is a share of (id or name); default its parent.
`smpfrac`	The long-form sampling fraction (default 0.25); full-count characteristics use 1 whatever is given.
`deff`	Design effect (default 1).
`conflvl`	Confidence level of the share's limits (default 0.95).
`roubase`	The base of the random rounding (default 5, Statistics Canada's); 1 for counts that are not rounded.

Details

The share is the numerator's count over the denominator's (by default the numerator's parent in the profile's hierarchy), for each geography, gender, age group and group. Statistics Canada does not publish standard errors for the profile, so the interval is an approximation built from what is known about how the counts were made:

- a long-form ("25% sample") count is estimated from a sample of about one household in four, so the share has variance $d(1-f)p(1-p)/(fN)$ for a sampling fraction f (`smpfrac`, 0.25) of a population of N (the denominator) and a design effect d (`deff`, 1 by default: the clustering of people in households raises it, and a value of 1.5 to 2 is prudent for characteristics shared within a household). On reserves (rows whose `residence` is `"On reserve"`) every household got the long form, so $f = 1$ there and the sampling term vanishes;
- every count is randomly rounded, up or down without bias, to a multiple of b (`roubase`, 5). Averaged over the remainders, that adds variance $c = (b^2 - 1)/6$ to each count (4 for $b = 5$), so $c(1 + p^2)/N^2$ to the share (the only error of a full-count share).

In all,

$$V = \frac{d(1-f)p(1-p)}{fN} + \frac{c(1+p^2)}{N^2}.$$

The interval is Wilson's at the effective sample size $n_e = \max(p(1-p), 1/N)/V$ (the floor $1/N$ keeps a share of 0 or 100% from an infinite n_e), so it stays inside 0 and 100. It does not carry non-response (see `tnrlf`) or the edit and imputation error, so it is narrower than the full error; treat it as a lower bound on the uncertainty.

Value

A data frame of class `rmbl_profile_share` with `geo`, `dguid`, `residence`, `gender`, `age`, `group`, `numerator`, `denominator`, `estimate` (percent), `lower`, `upper`, `quality`; ready for `indgap` and `dstntst`. `plot()` draws it.

References

Statistics Canada (2022). Guide to the Census of Population, 2021, chapter 9 (data quality) and chapter 10 (confidentiality and random rounding). Catalogue no. 98-304-X.

Wilson EB (1927). Probable inference, the law of succession, and statistical inference. Journal of the American Statistical Association 22(158):209-212. DOI 10.1080/01621459.1927.10502953

See Also

`cnsprof`.

Examples

```
p <- cnsprof(indiex("censuspop"))
# children who are foster children, by province, First Nations, Metis, Inuit
# and non-Indigenous
fc <- profshr(p, "Foster children", "Total - Children in census families")
fc <- fc[
  fc$age == "0 to 14 years" &
    fc$group %in%
      c("First Nations", "M\u00e9tis", "Inuit", "Non-Indigenous"),
]
head(fc)
plot(fc[fc$geo == "Canada", ])
```

prrgdecl *Declare an analysis before running it*

Description

prrgdecl() records, before the analysis is run, the statistics it intends to report and the claim each one is meant to test, with a UTC timestamp. prrgchk() compares that declaration with what was actually reported and lists the departures in both directions. Seal the declaration with cpsatst() and it acquires a date it cannot be moved off. Use the pair to make a pre-analysis plan checkable.

Usage

```
prrgdecl(hyps, note = NULL)

prrgchk(prereg, rptd)
```

Arguments

hyps	Named character vector: one claim per statistic, named by the statistic's name as it will be recorded.
note	Optional character string, default NULL: free text – the design, the data source, what would count as a refutation. Only the first element is kept.
prereg	A declaration from prrgdecl().
rptd	Character vector of the statistic names actually reported, or a manifest (any list with a named results element, such as a capsule manifest), whose result names are used.

Details

The comparison prrgchk() makes is asymmetric on purpose, because the two ways of departing from a plan are different failures:

- a declared statistic that was NOT reported is outcome switching – the analysis was run and its result dropped;
- a reported statistic that was NOT declared is an addition, and twenty of them is why a nominal p of 0.05 means nothing.

Neither is misconduct on its own and both are invisible without a declaration made in advance.

Statistics are matched by exact name. The timestamp is the computer's clock at the time of the call, so on its own it proves nothing; the attestation is what fixes the date. Nothing is written to disk.

Errors: hyps not a non-empty character vector with non-empty, unique names; prereg not from prrgdecl().

Value

prrgdecl() returns a list of class bricklayer_prereg with statistics (the names), hypotheses (the claims), note and decutc (ISO 8601 time).

prrgchk() returns a list of class bricklayer_prereg_check with ok (logical: no departures), the character vectors decnotrep and repnotdec, hypotheses (a data frame of statistic, hypothesis and logical reported) and decutc. Both print a short summary.

See Also

cpsatst() to seal a declaration, cpsfals() for the controls themselves, falsfam() for the correction that additions make necessary.

Examples

```
plan <- prrgdecl(c(
  ate = "use of force is higher in the exposed division",
  nrows = "the extract has the row count the source publishes"),
  note = "OTIS 2019-2024, division-level, pre-specified")

# afterwards, against what was reported
prrgchk(plan, c("ate", "nrows"))$ok

# dropping a declared outcome is outcome switching
prrgchk(plan, "nrows")$decnotrep

# and adding undeclared ones is what makes a nominal p meaningless
prrgchk(plan, c("ate", "nrows", "atebyyear",
                    "atebyprecinct"))$repnotdec

# the full check prints the departures
prrgchk(plan, c("ate", "atebyyear"))

try(prrgdecl(c("an unnamed claim")))
```

---

prsbands *Parse banded category labels into numeric bounds*

Description

Published administrative tables often report intervals instead of values: “18 to 24”, “2 to 5”, "50+", “Greater than 15 days”. This function reads such labels into a lower and an upper bound per band, and flags the bands that are open at the top or the bottom. It recognises the wordings open-data publishers actually use, and leaves a label it cannot read as NA rather than guessing. Use it as the first step before bandvals, expbands or bandsens.

Usage

```
prsbands(x, clsup = TRUE, intsca = TRUE)
```

Arguments

x	Character vector (or factor) of band labels.
clsup	Logical, default TRUE. For the inclusive open-upper wordings ("+", "and over", "or more" and the like), whether the stated number belongs to the band: TRUE reads "50+" as 50 and above, FALSE as above 50 (lower bound 51 on an integer scale). The exclusive wordings ("greater than 15") always exclude the number.
intsca	Logical, default TRUE. Whether the quantity is integer-valued (counts, days, whole years). It decides whether "greater than 15" starts at 16 (TRUE) or at 15 as an infimum (FALSE). A label written with decimals is always treated as continuous.

Details

Labels are lower-cased and trimmed; the Unicode dash family (U+2010 to U+2015 and the minus sign U+2212) is read as a hyphen, and correctly grouped thousands separators are removed ("1,000 to 2,499"). Then, in order:

- **Closed range**: two numbers joined by to, - or through, optionally preceded by one word and followed by a unit ("ages 18-24", "0 to 5 overnight stays"). The two numbers must be the only numbers in the label, the lower must not exceed the upper, and no open-band cue word (over, under, more, less, plus, ...) or + may appear; otherwise the label is NA.
- **Open upper band, inclusive**: "50+", "65 and over", "and above", "or more", "or over", "plus". The lower bound is the number when clsup = TRUE, and the number plus one step otherwise.
- **Open upper band, exclusive**: "greater than", "more than", "over", "above", "gt". The lower bound is the number plus one step.
- **Open lower band, inclusive**: "17 and under", "or less", "or fewer", "and below", "or under". The upper bound is the number.
- **Open lower band, exclusive**: "less than", "fewer than", "under", "below", "<", "lt". The upper bound is the number minus one step.
- **Bare number**: a band of width zero. A bare negative number ("-5") is read as a suppressed cell and stays NA.

The step is 1 when intsca = TRUE and the label has no decimal point, and 0 otherwise. A label with more than one number that is not a range, an empty label and NA all give NA bounds. Anything other than a character or factor vector is an error.

Value

A data frame with one row per label and columns label (character, as given), lower and upper (numeric bounds, NA for the open side of an open band or for an unparsed label), and openlo, openup (logical).

References

The forms recognised here are taken from the categories the Ontario Ministry of the Solicitor General publishes in its inmate datasets (the segregation and restrictive-confinement releases), where the placement-count and age categories are banded and the top band is open.

See Also

`bandvals`, `bandsens`, `expbands`

Examples

```
prsbands(c("1", "2 to 5", "6 to 10", "Greater than 10"))

prsbands(c("18 to 24", "25 to 49", "50+", "under 18"))

# "50+" excluding 50, and a continuous scale
prsbands(c("50+", "over 15"), clsup = FALSE)
prsbands(c("over 15", "under 0.5"), intsca = FALSE)

# An unrecognised or self-contradictory label is NA, not guessed at.
prsbands(c("3", "unknown", "24 to 18", "15 to 19 and over"))

try(prsbands(1:3))
```

---

pubbnd *Bounds implied by rounding and suppression in a published table*

Description

Administrative releases rarely publish the count that was observed. Statistics Canada rounds to a base (random rounding to base 5 moves a cell by up to 4); provincial tables round to the nearest 5 or 10; cells below a disclosure limit are printed as `"x"`, `"<5"` or `".."`. Every figure computed downstream inherits that uncertainty, and it is not sampling uncertainty: no confidence interval covers it. This function turns each published cell into the interval of observed counts that could have produced it, so `chgenv()` and `yoybnd()` can carry the interval through a difference, a percent change or a rate.

Usage

```
pubbnd(
  x,
  rnd = NULL,
  rndkind = c("nearest", "random"),
  suplim = NULL,
  supmar = c("x", "X", "s", "..", "F", "suppressed", "n/a")
)
```

Arguments

x	Published values: a numeric vector, or a character vector (or factor) in which suppressed cells appear as one of supmar or as "<k" for a limit k.
rnd	Numeric scalar, the base the release rounds to (5, 10, ...), or NULL (the default) when the counts are exact.
rndkind	"nearest" (the printed value is the observed value rounded to the nearest multiple of rnd, so the observed value lies within half a base) or "random" (Statistics Canada's random rounding, where the observed value lies within rounding - 1).
suplim	Numeric scalar or NULL (the default). For a cell printed as suppressed with no explicit limit, the smallest count that would have been published: the cell then lies in [0, suplim - 1].
supmar	Character vector of the strings that mark a suppressed cell, matched exactly after trimming. Default c("x", "X", "s", "..", "F", "suppressed", "n/a").

Details

For a published value v rounded to base b: with rndkind = "nearest" the observed count lies in $[\max(0, v - b/2), v + b/2]$; with "random" (unbiased random rounding, where a count is moved to one of the two neighbouring multiples) it lies in $[\max(0, v - (b-1)), v + (b-1)]$. The bounds are not rounded to whole numbers. A suppressed cell lies in $[0, k-1]$, with k from "<k" or from suplim; rounding is not applied to it. Without rnd, a published number is taken as exact (lower = upper = value).

Text cells are trimmed, and thousands separators (commas) are removed before they are read as numbers. A missing value (NA) that is not a suppression mark gets status "missing" and NA bounds.

Errors: rnd not one positive finite number; a negative or infinite count in x; an unknown rndkind; a text cell that is neither a number nor a suppression mark (the message lists the cells); a suppressed cell with no limit (neither "<k" nor suplim).

Value

A data frame of class c("rmbl_bounds", "data.frame"), one row per cell, with value (numeric, the published number, NA for a suppressed cell), lower and upper (numeric, the interval of observed counts) and status (character: "exact", "rounded", "suppressed", or "missing" for an NA that carried no suppression mark). Attribute "bounds" records rounding, rndkind and suplim. plot() draws each value inside its interval.

See Also

chgenv(), yoybnd()

Examples

```
pubbnd(c(120, 35, 0), rnd = 5)
pubbnd(c("120", "x", "<5"), rnd = 5,
                suplim = 5)
# random rounding to base 5 is wider than rounding to the nearest 5
```

```
pubbnd(15, rnd = 5, rndkind = "random")

# thousands separators and a custom suppression mark
pubbnd(c("1,200", "--", "<10"), suplim = 10,
                 supmar = "--")

# a suppressed cell needs a limit
try(pubbnd(c("120", "x")))
```

`qltyrcd` *Recode another agency's quality flags to the package's*

Description

The comparisons of this family read Statistics Canada's quality flags: `E` (use with caution), and `F`, `x`, `..`, `...` (no value: too unreliable, suppressed, not available, not applicable). Other agencies flag otherwise: the Australian Bureau of Statistics `np` (not published) and `*` or `**` (relative standard error over 25% or 50%), Stats NZ `S` (suppressed), the US Census Bureau `(X)`. This maps them to the package's vocabulary, so suppressed cells are never compared and uncertain ones are marked.

Usage

```
qltyrcd(x, suppr = c("F", "x", "..", "..."), caution = "E")
```

Arguments

`x`	The flags, one per row.
`suppr`	Flags meaning the value must not be used (it becomes `"x"`).
`caution`	Flags meaning the value may be used with caution (it becomes `"E"`).

Value

A character vector of `"ok"`, `"E"` and `"x"`; empty and missing flags are `"ok"`. `x` must be character (or a factor); a flag in neither `suppr` nor `caution` (other than `"ok"`) becomes `"ok"` and is named in a warning.

See Also

`limfrom`, `indgap`.

Examples

```
# flags as an Australian table might give them
qltyrcd(c("", "np", "*", "**", NA), suppr = c("np", "**"), caution = "*")
```

`ralloc` *Make a Randomisation List with a Sealed Commitment*

Description

Allocates participants to arms by simple, permuted-block or stratified block randomisation, from a seed through the package's NIST SP 800-90A generator (`drbgnew()`), so the list can be regenerated exactly. It returns the list and one SHA-256 commitment to it: sign and timestamp the commitment before the first participant is enrolled, keep the list and the seed with the unblinded statistician, and at unblinding `rallocvrf()` shows that the allocation used is the one committed to.

Usage

```
ralloc(
  n = NULL,
  arms = c("A", "B"),
  ratio = rep(1, length(arms)),
  block = 2 * sum(ratio) * c(1, 1.5),
  strata = NULL,
  seed = NULL,
  codes = FALSE,
  prefix = "P"
)

## S3 method for class 'rmbl_alloc'
plot(x, ...)
```

Arguments

`n`	How many participants, a positive whole number. With `strata`, it may be left out (it is their total).
`arms`	The arms' names, default `c("A", "B")`.
`ratio`	Whole numbers, one per arm: the allocation ratio, default equal.
`block`	The block sizes to choose from at random, default `2 * sum(ratio)` and `3 * sum(ratio)` (4 and 6 for two equal arms); `NULL` for simple randomisation.
`strata`	`NULL`, or the strata and their numbers of places: a named vector of counts, or a data frame with the stratum columns and `n`.
`seed`	`NULL` (default, 48 random bytes), 48 raw bytes, their 96 hexadecimal characters, or a number or word that is expanded to 48 bytes.
`codes`	Give every participant a kit code, default `FALSE`.
`prefix`	The start of the participant ids, default `"P"`.
`x`	A list from ralloc(), for `plot()`.
`...`	Passed to the base plot.

Details

Every draw is exact: an arm or a block size is chosen with rejection sampling on 32-bit draws (no modulo bias), and a block is shuffled by Fisher-Yates. With `block = NULL` each participant is allocated on their own (simple randomisation), with probabilities in the ratio `ratio`. Otherwise blocks are drawn one after another, each of a size chosen at random from `block` (every size a multiple of `sum(ratio)`), each holding the arms in the ratio; random block sizes keep the next allocation hard to guess at a site. The list stops at `n` participants, so its last block may be incomplete; the print-out says how far the arms then differ.

With `strata`, each stratum has its own sequence of blocks: name the strata and how many places each needs, as a named vector (`c(site1 = 40, site2 = 40)`) or a data frame with the stratum columns and a column `n` (one row per combination, such as site by sex).

The commitment is the SHA-256 of the settings, every row of the list and the SHA-256 of the seed, so it binds both the list and the seed: a list edited afterwards, or a seed chosen afterwards to give a wanted list, does not match it. Each field enters it with its length, so settings that differ (a stratum named `"x,y"` against strata `"x"` and `"y"`) never share a commitment; lists made by 1.0.3's first release keep their commitment and still verify. The print-out shows how many participants each arm has, never who is in which arm, and not the seed.

Keep `seed = NULL` for a trial: the seed is then 48 bytes from the operating system's generator (`rndbytes()`), which nobody can guess before the list is opened. A number or a word as `seed` gives a list anyone can recompute, which suits teaching and testing, not concealment.

With `codes = TRUE` every participant also gets a kit code (six characters, no look-alike letters or digits): sites and pharmacies are given `id` and `kit` only, and the arm stays with the statistician.

Errors: fewer than two arms, repeated or empty arm names; a `ratio` that is not one positive whole number per arm; a block size that is not a positive multiple of `sum(ratio)` up to 1000000; `n` that is not a whole number from 1 to 10000000, or that disagrees with the strata's total; a `seed` that is not one value, or a raw `seed` that is not 48 bytes.

Value

An object of class `rmbl_alloc`: a data frame with `id`, `stratum` (with `strata`), `block` (the block's number in its stratum; `NA` for simple randomisation), `arm` and, with `codes`, `kit`, in enrolment order. Attribute `"alloc"` holds the settings, the seed (as hexadecimal) and `digest`, the commitment. It prints as a box of counts.

See Also

`rallocvrf()` at unblinding; `cpssign()` and `idcrd()` to sign the commitment; `tsstmp()` to timestamp it and `tsvrf()` to check the timestamp; `prrgdecl()` to declare the outcomes in the same signed record.

Examples

```
# 1:1 in blocks of 4 or 6, for 40 participants
a <- ralloc(40, seed = "teaching example")
a
attr(a, "alloc")$digest            # sign and timestamp this first
head(as.data.frame(a))
```

```
# 2:1, stratified by site, with kit codes for the pharmacy
s <- ralloc(arms = c("drug", "placebo"), ratio = c(2, 1),
            strata = c(Toronto = 30, Ottawa = 24), codes = TRUE,
            seed = 7)
s
head(as.data.frame(s)[c("id", "kit")])  # what the sites are given

# simple randomisation: how many each arm has, against the ratio
plot(ralloc(60, block = NULL, seed = 1))

# at unblinding: the list is the one committed to
rallocvrf(a, digest = attr(a, "alloc")$digest)

# a block size the ratio does not divide
try(ralloc(30, block = 5))
```

`rallocvrf` *Check a Randomisation List at Unblinding*

Description

Checks that a list from `ralloc()` is the one committed to before enrolment, and that the stated procedure and seed produce exactly it.

Usage

```
rallocvrf(x, digest = NULL, seed = NULL)
```

Arguments

`x`	A list from `ralloc()` (kept with `saveRDS()`).
`digest`	The commitment published before enrolment, or `NULL` to check against the one in `x`.
`seed`	The seed revealed at unblinding, or `NULL` for the one in `x`.

Details

Three checks: the list's own commitment, recomputed from its rows, settings and seed, equals `digest` (the value signed and timestamped before enrolment; without it, the commitment stored in the list); the list regenerates row for row from the seed and the settings; and every participant id is there once. A list edited after the commitment fails the first; a list that does not come from its seed (one chosen among many, or written by hand) fails the second.

Errors: `x` that does not come from `ralloc()`.

Value

An object of class `rmbl_alloc_check`: a list with `ok` (all checks passed) and `checks` (a data frame of `check`, `ok` and `detail`). It prints as a box that opens with VERIFIED or NOT VERIFIED.

See Also

`ralloc()`

Examples

```
a <- ralloc(20, seed = "teaching example")
commitment <- attr(a, "alloc")$digest
rallocvrf(a, digest = commitment)

# one allocation changed after the commitment
b <- a
b$arm[3] <- if (b$arm[3] == "A") "B" else "A"
rallocvrf(b, digest = commitment)

# a different seed revealed at unblinding
rallocvrf(a, digest = commitment, seed = "another seed")
```

---

raplok *Is CPU energy measurable here?*

Description

`TRUE` when the operating system exposes readable RAPL (Running Average Power Limit) energy counters for at least one CPU package: Linux with the `intel_rapl` powercap driver (Intel and, since kernel 5.8, AMD), with read permission on `energy_uj`. On most systems since Linux 5.10 that file is readable by root only, so `FALSE` is the common answer for an ordinary user, and `co2fp` then falls back to its model.

Usage

```
raplok(rapldir = "/sys/class/powercap")
```

Arguments

`rapldir`	Character scalar, the powercap directory; default `"/sys/class/powercap"`, the Linux location. Another path is useful for testing against a copy of the directory tree.

Details

A domain counts when it is a directory named `intel-rapl:N` (the `intel-rapl:N:M` subdomains are parts of a package and would count the same energy twice) whose `name` file starts with `"package"` and whose `energy_uj` file holds a number. Every such domain must be readable: if any is not, the answer is `FALSE`. Only files under `rapldir` are read; nothing is written. On macOS and Windows there is no such directory and the answer is `FALSE`. Errors when `rapldir` is not a single non-missing string.

Value

A logical scalar: `TRUE` when every CPU-package energy counter under `rapldir` is readable, `FALSE` otherwise (including when the directory does not exist).

See Also

co2fp

Examples

```
raplok()

# a fake powercap tree with one readable package domain (Linux names the
# domain "intel-rapl:0"; a colon is not allowed in a Windows file name)
if (.Platform$OS.type != "windows") {
  d <- file.path(tempfile(), "intel-rapl:0")
  dir.create(d, recursive = TRUE)
  writeLines("package-0", file.path(d, "name"))
  writeLines("123456789", file.path(d, "energy_uj"))
  raplok(dirname(d))
}

# a folder that does not exist is simply FALSE; a non-path is an error
raplok(file.path(tempdir(), "no-such-dir"))
try(raplok(NA_character_))
```

---

ratechg *Change in a rate between periods*

Description

Compares each period's rate with the rate `lag` periods earlier (for example this year's stops per 100,000 residents with last year's), and gives the rate ratio, the percent change and an exact confidence interval for the change. Use it instead of the percent change of two published rates: both the counts and the populations move between periods, and the interval here accounts for the randomness in the counts.

Usage

```
ratechg(x, ...)

## Default S3 method:
ratechg(x, ...)

## S3 method for class 'data.frame'
ratechg(
  x,
  count,
  pop,
  prd,
  by = NULL,
  lag = 1L,
  per = 1000,
  conflvl = 0.95,
  mincnt = 0,
  prdord = NULL,
  ...
)
```

Arguments

x	A data frame.
...	Passed to methods.
count	Column of counts (unquoted or a string): non-negative whole numbers.
pop	Column of exposure (unquoted or a string): positive.
prd	Column of periods (unquoted or a string). Sorted within each group; see Details for how the earlier period is found.
by	Character vector of grouping columns, default NULL.
lag	Whole number, default 1: how many periods back to compare against.
per	The rate denominator: a positive number, or "1k", "10k", "100k", "1m". Default 1000.
conflvl	Number in (0, 1), default 0.95: the confidence level of the interval.
mincnt	Number, default 0. Comparisons where the earlier count is at or below this are flagged as resting on a small count; the percent change and its exact interval are still reported, as yoy does.
prdord	Optional character vector: the period labels in order, as for yoy(). A label it lists that the data lacks is a gap, so the change across it is not reported as one step.

Details

This is not the percent change of two rates treated as measured numbers. Both the counts and the denominators move between periods, and an interval that ignores the denominators understates

the uncertainty of the change. The construction here conditions on the total of the two counts and corrects for the ratio of the two exposures, which is yoy's exact conditional-binomial interval generalised to unequal denominators: with equal populations it reduces to exactly that.

With counts a (now) and b (earlier) and exposures t_a, t_b, conditional on $a + b$ the count a is binomial with success probability $p = RR\,t_a/(RR\,t_a + t_b)$. The Clopper-Pearson limits for p, $p_L = B^{-1}(\alpha/2;\, a, b+1)$ and $p_U = B^{-1}(1-\alpha/2;\, a+1, b)$ (0 when $a = 0$, 1 when $b = 0$), give $RR = (t_b/t_a)\,p/(1-p)$ and the percent change $100(RR - 1)$. When $b = 0$ the upper limit is `Inf` and the percent change itself is `NA`; when both counts are 0 there is no interval.

Counts and populations are summed within each period (and by group). Matching to the earlier period: for a numeric period column the comparison is the period whose value is exactly `lag` less, so a missing year gives no comparison instead of a silent comparison against the wrong year. For any other period type (character, factor, Date) the earlier period is `lag` steps back along the periods observed in the group, so a gap in such a column is not detected.

Errors: `x` not a data frame; columns not found; `lag` not a positive whole number; `conflvl` outside (0, 1); a population of zero or below.

Value

A data frame of class `c("rmbl_rate_change", "data.frame")` with one row per group and period: the grouping columns, the period column, `count`, `population`, `rate`, `prerate`, `precnt`, `prepop`, `rateratio`, `pctchg`, `pctlo`, `pctup` (numeric) and the character `flag`. The flag is `"no comparison period"`, `"earlier period is zero"`, a small-count note, or `NA`. The `ratechg` attribute records the settings.

See Also

`rtcalc`, `yoy` for change in a count or a measured quantity.

Examples

```
d <- data.frame(
  year = rep(2021:2023, each = 2),
  division = rep(c("North", "South"), 3),
  stops = c(400, 70, 430, 66, 455, 61),
  residents = c(120000, 41000, 122000, 41500, 125000, 42000))

# North's count rose while its population rose too: the rate change
# is smaller than the count change, which is the reason to use it
ratechg(d, stops, residents, year, by = "division", per = "100k")

# a missing year gives no comparison rather than a wrong one
gap <- data.frame(year = c(2019, 2020, 2022), n = c(10, 14, 12),
                  pop = c(1000, 1000, 1100))
ratechg(gap, n, pop, year)$flag

# two-year change, flagging small earlier counts
ratechg(d, stops, residents, year, by = "division", lag = 2,
            mincnt = 65)

try(ratechg(1:3))
```

`rategap` *The gap between two standardised rates*

Description

The ratio and the difference of two directly standardised rates from `agestd`, with confidence intervals: log-normal for the ratio (variance $v_1/y_1^2 + v_2/y_2^2$) and normal for the difference, the two rates independent. Both measures are reported, as the relative gap alone hides how large the absolute one is (Anderson et al. 2016).

Usage

```
rategap(x, grp, ref)
```

Arguments

`x`	Output of `agestd` (two or more groups).
`grp`, `ref`	The labels of the group and of the reference group.

Value

A one-row data frame: `group`, `reference`, `ratio` with `ratiolo`/`ratioup`, and `difference` (per `per` people) with `diflo`/`difup`.

References

Anderson I, Robson B, Connolly M, et al. (2016). Indigenous and tribal peoples' health (The Lancet-Lowitja Institute Global Collaboration): a population study. The Lancet 388(10040):131-157. DOI 10.1016/S0140-6736(16)00345-7

Examples

```
# made-up counts, to show the computation
std <- c(30000, 25000, 20000, 15000, 10000)
r <- agestd(events = c(4, 9, 11, 6, 3, 2, 4, 6, 5, 4),
                     pop = c(9000, 8000, 6000, 4000, 2000,
                                   20000, 19000, 18000, 15000, 12000),
                     std = std, grp = rep(c("Inuit", "Non-Indigenous"),
                                          each = 5))
plot(r)
rategap(r, "Inuit", "Non-Indigenous")
```

rcdmnf *Record a recode as a manifest that can be signed and verified later*

Description

Builds a record of a recode that a later reader can check: the mapping and its SHA-256, the counts per label before and after, the results of the before/after cross-tabulation check and (when given) of the match against published counts, a UTC timestamp, and, with a key, a signature over the whole record. Write it to disk with `wrrcdmnf` and re-check it with `vrfrcdmnf`, so anyone can confirm which labels went where and that nothing was changed afterwards.

Usage

```
rcdmnf(orig, recd, map, pub = NULL, key = NULL, ctx = NULL)
```

Arguments

`orig`, `recd`	Vectors of the same length: the values before and after the recode.
`map`	Named character vector, old = new: the mapping that was applied.
`pub`	Optional named numeric vector of published counts per recoded label, default `NULL` (no check).
`key`	Optional signing key, default `NULL` (unsigned): a key from `pqckg` or `fipskg`. When given the manifest carries a signature and the public key.
`ctx`	Optional character string, default `NULL`: free text such as the dataset, table and analyst. Only the first element is kept.

Details

The function first runs `vrfrcd`, as `vrfrcd(original, recoded, mapping)`. When `pub` is given it then runs `vrfmrg`, as `vrfmrg(recoded, published)`, with the default `strict = TRUE`. Either check failing is an error, so a manifest is only produced for a recode that passed. For that reason `checks$matches_published` is `TRUE` or `NA` (no published counts), never `FALSE`.

`body_sha256` is `csha256` of the canonical JSON of every other field. With a `key`, `cpssign` signs that hash and the manifest also stores the public key: a `bricklayer_signing_key` from `pqckg` (XMSS, which consumes one one-time leaf per signature) or a key from `fipskg` (ML-DSA or SLH-DSA). Nothing is written to disk.

Value

A list of class `bricklayer_recode_manifest` with these elements:

- `context`;
- `created` (ISO 8601 UTC time);
- `mapping` (list) and `mapping_sha256`;
- `n` (number of rows) and `n_missing`;

- counts_before and counts_after (named lists of counts);
- checks, a list: crosstab_is_function, missingness_unchanged and matches_published;
- published (list or NULL);
- body_sha256;
- with a key, signature and public_key.

Examples

```
x <- c("W", "B", "W", "I")
map <- c(W = "White", B = "Black", I = "Indigenous")
y <- grdrcd(x, map)
m <- rcdmnf(x, y, map,
  pub = c(White = 2, Black = 1, Indigenous = 1),
  ctx = "example survey, race item"
)
m$checks
m$counts_after

# signed with an ML-DSA key
k <- fipskg("ML-DSA-44", seed = as.raw(1:32))
ms <- rcdmnf(x, y, map, key = k)
names(ms)

# a recode that does not match the published counts is refused
try(rcdmnf(x, y, map, pub = c(White = 1, Black = 2, Indigenous = 1)))
```

---

recevid *Record Correctness Evidence in a Manifest*

Description

Adds the outcome of a correctness check to a manifest, so the capsule carries not only the reported numbers but the checks they passed. Each planted effect from cpsplant() becomes a record (recovered against planted, within the tolerance); a cpsspec() check records the share of specifications agreeing in sign (which should be 1); a cpsfals() result records each control that ran (1 for passed).

Usage

```
recevid(man, x, grp = "evidence")
```

Arguments

man	A manifest from mkmnf().
x	A result of cpsplant(), cpsspec() or cpsfals().
grp	The records' group.

Value

The manifest, with the new records.

See Also

`recrd()`, `cpsbndl()`.

Examples

```
set.seed(1)
d <- data.frame(x = rbinom(100, 1, 0.5))
d$y <- d$x + rnorm(100)
dm <- function(df) mean(df$y[df$x == 1]) - mean(df$y[df$x == 0])
man <- mkmnf(list(project = "demo"), env = FALSE)
man <- recevid(man, cpsplant(d, dm, "y", "x", n = 50, seed = 1))
man
```

---

recrd *Record a Cross-Check Result in a Manifest*

Description

Appends one named cross-check to a manifest: a number your analysis produced (`obs`) next to the number it should reproduce (`expd`, for example a published table value). Each entry is classified `PASS`, `DIFFER` or `INFO`, a one-line report is shown as a message, and the updated manifest is returned so calls can be chained. The tallies appear in `wrsmrytxt()` and the entries in `wrmnfjs()`.

Usage

```
recrd(man, name, obs, expd, tol = 1e-04, grp = "general", syn = FALSE)
```

Arguments

`man`	A manifest as returned by `mkmnf()`.
`name`	Character scalar, the name of this cross-check. It is the key of the entry in `manifest$results`, so it should be unique.
`obs`	The value the analysis produced: a numeric scalar for a tolerance check, or any single value to be stored as information.
`expd`	The value to compare against, of the same kind.
`tol`	Numeric scalar, the absolute tolerance: a numeric pair at most `tol` apart is `PASS`. Default `0.0001`.
`grp`	Character scalar, a grouping label stored with the entry (for example the table the number comes from). Default `"general"`.
`syn`	Logical scalar; `TRUE` marks this entry `INFO` because a comparison on synthetic data is not meaningful. Default `FALSE`.

Details

The status is decided in this order: `INFO` when `synthetic = TRUE` or the manifest's `meta$synthetic` is `TRUE` (a comparison on generated data is not meaningful); `PASS` when both values are numeric and $|observed - expected| \leq tol$; `DIFFER` when both are numeric and further apart; `INFO` otherwise (for example a text or logical value, or an `NA`). The tolerance is absolute, not relative: scale it to the size of the quantity.

`obs` and `expd` must be single values: a vector of another length (including an empty one) is an error naming the argument; record a vector's elements under separate names, or a summary of it. Recording under an existing `name` replaces that entry. The message line is a side effect; wrap the call in `suppressMessages()` to silence it.

Value

The manifest with `results[[name]]` set to a list with `group`, `observed`, `expected`, `diff` (the absolute difference, `NA` when the pair is not numeric), `status` (`"PASS"`, `"DIFFER"` or `"INFO"`), `tol` and `note` (a sentence for synthetic entries, else `NULL`).

See Also

`mkmnf()`, `wrsmrytxt()`.

Examples

```
man <- mkmnf(list(project = "demo"), env = FALSE)

# Within tolerance -> PASS. (Every status below is INFO instead in a
# session that has already generated synthetic data, because mkmnf()
# then marks the manifest synthetic; see mkmnf().)
man <- recrd(man, "meanmatches",
  obs = 1.0001, expd = 1,
  tol = 0.001
)
man$results$meanmatches$status # "PASS"

# Outside tolerance -> DIFFER.
man <- recrd(man, "sdmatches", obs = 2.5, expd = 2.0, tol = 0.01)
man$results$sdmatches$status # "DIFFER"

# Synthetic data -> INFO (comparison not meaningful).
man <- recrd(man, "syntheticrow",
  obs = 5, expd = 5,
  syn = TRUE
)
man$results$syntheticrow$status # "INFO"

# a non-numeric value is stored as information, silently
man <- suppressMessages(recrd(man, "country", obs = "CA", expd = "CA",
                              grp = "metadata"))
man$results$country$status # "INFO"

# Calls chain: record() returns the mutated manifest.
```

```
length(man$results) # 4
```

`relbl` *Relabel a categorical variable by name, never by position*

Description

Gives a factor or character vector new labels with a mapping written by name, old label = new label, and returns a factor. Assigning a vector of labels by position (the `levels(x) <- ...` idiom, or `factor(x, labels = ...)` with labels in a different order from the codes) is how the four race categories were rotated in the analysis that the Ontario Human Rights Commission corrected on 26 January 2023; this function refuses unnamed mappings outright, and refuses levels that the mapping does not cover.

Usage

```
relbl(x, map, keep = character())
```

Arguments

`x`	A factor or character vector.
`map`	Named character vector, old label = new label. Every level of `x` must appear as a name unless listed in `keep`. Several old labels may map to one new label.
`keep`	Character vector, default empty: levels allowed to pass through unchanged.

Details

The old levels are `levels(x)` for a factor and the sorted distinct values otherwise; all of them must be names of `map` or members of `keep`, so for a factor even an unused level must be mapped. The recode itself is `grdrcd`. The levels of the result are the distinct values of `map` in the order given, followed by the `keep` labels that occur in the data.

Errors: `map` without names (including a plain vector of new labels); a level with no mapping.

Value

A factor with the new labels, levels in the order of `mapping`, carrying a `recode_audit` attribute (see `grdrcd`).

See Also

`grdrcd`, `rlblfrns`, `declbld`

Examples

```
f <- factor(c("W", "B", "O", "W"))
relbl(f, c(W = "White", B = "Black", O = "Other"))

# the level order follows the mapping, not the alphabet
levels(relbl(f, c(W = "White", O = "Other", B = "Black")))

# a kept label passes through
relbl(c("W", "B", "X"), c(W = "White", B = "Black"), keep = "X")

# positional labels are refused
try(relbl(f, c("Black", "Other", "White")))
```

---

rgncov *Population of the regions that contain a unit, and of those that do not*

Description

Summarises where a set of point-located units (jails, courthouses, hospitals, police stations) sits relative to the regions of a statistical geography: how many regions hold at least one unit, and what share of the population lives in them. Use it to describe the footprint of a set of facilities, not to build a per-capita rate (see Details).

Usage

```
rgncov(reg, pop, units)
```

Arguments

`reg`	Region identifiers, one per region, not repeated (character, or anything that converts to it; surrounding spaces are trimmed).
`pop`	Population of each region, same length and order, non-negative numbers.
`units`	Number of units located in each region, same length and order, non-negative whole numbers. Zero is the expected value for most regions in most geographies.

Details

The share this returns is CONTEXT, not a denominator, and the distinction is the reason the function exists rather than a bare `tapply()`.

A region holding no unit is not an unserved population. Units serve catchments, and a catchment is an administrative fact about where people are sent from; a point location does not state it and cannot imply it. Some geographies were never meant to have one unit each.

So a rate built by summing the populations of unit-holding regions pairs a denominator covering part of the territory with a numerator drawn from all of it. Every such rate is inflated, and inflated unevenly: a dense region holding one unit and a sparse region holding seven distort it in opposite

directions. Where numerator and denominator must cover the same population, the defensible figure is the whole-territory one.

The print method says this each time, because the covered share is precisely the number a reader is tempted to divide by. With a total population of zero the shares are NA.

Errors: empty, missing or repeated reg values; arguments of different lengths; missing or negative pop; missing, negative or fractional units.

Value

A data frame of class c("rmbl_region_coverage", "data.frame"), one row per region, with the columns region (character), population (numeric), units (numeric), hasunit (logical) and popshr (numeric, percent of the total population), ordered by units then population, both descending. The totals are carried on the coverage attribute, a list of regions, withunit, witunit, units, totalpop, covpop, uncpop and covshr (percent), and printed by the print method (which shows the first n = 10 regions).

See Also

rgnmapintg(), rgnmapcmp(), rgnmapalt()

Examples

```
# Four regions, two of which hold a facility.
cov <- rgncov(
  reg = c("A", "B", "C", "D"),
  pop = c(1200000, 800000, 450000, 90000),
  units = c(3, 0, 1, 0)
)
cov

# The covered share is reported, and is not a rate denominator.
attr(cov, "coverage")$covshr
attr(cov, "coverage")[c("withunit", "uncpop")]

# counting units per region from a facility list first
fac <- c("C", "A", "A", "A")
regions <- c("A", "B", "C", "D")
units <- as.vector(table(factor(fac, levels = regions)))
rgncov(regions, c(1200000, 800000, 450000, 90000), units)$hasunit

# a region listed twice is refused
try(rgncov(c("A", "A"), c(10, 20), c(1, 0)))
```

---

rgnmapalt *Check a region map against an independently derived assignment*

Description

Compares the region each unit was assigned with the region a DIFFERENT method assigns it, and marks the disagreements that are already known and explained. Use it to catch errors in the method that built the map, which re-running that same method can never find.

Usage

```
rgnmapalt(map, unit, reg, route, known = character())
```

Arguments

`map`	Data frame, one row per unit.
`unit`	Name of the unit identifier column (a string).
`reg`	Name of the assigned region column (a string).
`route`	Named character vector (names are units, values regions), or a data frame with the same two column names as `map`, giving the second method's assignment. Units it does not cover, and entries that are `NA` or blank, are skipped rather than counted as disagreements.
`known`	Character vector of units whose disagreement is expected and documented – the cases the second route is known to get wrong. Default none.

Details

This is the only check here that can catch an error in the original method, because it does not use that method. Recomputing point in polygon against the same boundary file proves the pipeline is deterministic; deriving the region a second way – from a name, a postal geography, an administrative lookup – can disagree, and a disagreement is information either way round.

The `known` argument exists because a second route usually has understood weaknesses: a place name that is a community rather than a municipality, an amalgamated city, a township absorbed into a neighbour. Listing them keeps the check sharp instead of loosening the tolerance until everything passes, and listing them by NAME means an unexpected disagreement cannot hide inside an allowance.

Units and regions are compared as text with spaces trimmed. Errors: `map` not a data frame or without the named columns; a `route` vector without names; a `route` data frame without the two columns.

Value

A data frame of disagreements, sorted with unexplained ones first, with the columns `unit`, `primary` (the map's region), `second` (the route's region), all character, and `known` (logical). `sum(!x$known)` is the number of unexplained disagreements and zero is the passing value; `nrow(x)` should equal the number of documented cases, because a documented case that stops disagreeing means the second route changed underneath the documentation.

See Also

`rgnmapcmp()`, `rgnmapintg()`

Examples

```
cw <- data.frame(
  inst = c("North Jail", "South Jail", "Hill Jail"),
  cd = c("3557", "3520", "3506"),
  stringsAsFactors = FALSE
)

# a name-based route that is known to mis-place one unit
route <- c(
  "North Jail" = "3557",
  "South Jail" = "3520",
  "Hill Jail" = "3519"
)
rgnmapalt(cw, "inst", "cd", route, known = "Hill Jail")

# an undocumented disagreement is what the check is for
route["South Jail"] <- "3521"
d <- rgnmapalt(cw, "inst", "cd", route, known = "Hill Jail")
sum(!d$known)

# the route as a data frame with the same column names
routedf <- data.frame(
  inst = c("North Jail", "Hill Jail"),
  cd = c("3557", "3506")
)
nrow(rgnmapalt(cw, "inst", "cd", routedf))

# an unnamed vector cannot be matched to units
try(rgnmapalt(cw, "inst", "cd", c("3557", "3520")))
```

---

rgnmapcmp *Compare a recomputed region map against a published one*

Description

Matches two region maps on the unit identifier and compares the named columns cell by cell, reporting for each column how many cells disagree and the first disagreement. Use it to confirm that a region map rebuilt from the boundary file (for example with `rgnmappts()`) reproduces the one that was published or pinned.

Usage

```
rgnmapcmp(pub, obs, unit, cols = NULL)
```

Arguments

pub	The region map as published (a data frame).
obs	The region map as recomputed (a data frame).

`unit`	Name of the unit identifier column, present in both.
`cols`	Character vector of columns to compare. `NULL` (the default) compares every column the two share apart from `unit`.

Details

Units are matched by their identifier as text with spaces trimmed; when an identifier repeats, its first row is used. Numeric columns are compared with `all.equal()` at its default tolerance (relative 1.5×10^{-8}), so a coordinate that survived a round trip through text is not reported as a change; everything else is compared exactly as text after trimming whitespace. Two missing values agree; a missing value against a present one disagrees.

What this establishes is that the recomputation reproduced the published assignment. It does NOT establish that the assignment is right: run the same method against the same boundary file and a definitional error reproduces perfectly. That is what `rgnmapalt()` is for.

Errors: either input not a data frame or without `unit`; a column in `cols` that is not in both.

Value

A data frame, one row per compared column plus a first `"rows"` row for units present on one side only, with columns `column` (character), `cells` (integer: for `"rows"` the number of distinct units on either side, otherwise the number of shared units compared), `mismatched` (integer) and `first` (character, the first disagreement written out as text, `""` when none). Zero `mismatched` throughout is the passing result.

See Also

`rgnmapalt()`, `rgnmapintg()`

Examples

```
pub <- data.frame(
  inst = c("North Jail", "South Jail"),
  cd = c("3557", "3520"),
  stringsAsFactors = FALSE
)
obs <- pub
rgnmapcmp(pub, obs, "inst")

# a changed assignment is reported with the unit that moved
obs$cd[2] <- "3521"
rgnmapcmp(pub, obs, "inst")

# a unit on one side only, and a coordinate that only moved by rounding noise
pub$lat <- c(44.39, 43.65)
obs2 <- data.frame(
  inst = c("North Jail", "Hill Jail"),
  cd = c("3557", "3506"),
  lat = c(44.39 + 1e-12, 45.1)
)
rgnmapcmp(pub, obs2, "inst", cols = c("cd", "lat"))
```

```
# a compared column must be in both maps
try(rgnmapcmp(pub, obs, "inst", cols = "lat"))
```

`rgnmapintg` *Internal soundness of a region map*

Description

Checks that a region map (a crosswalk from units, such as facilities, to the statistical regions that contain them) lists each unit once, gives every unit a region, and names only regions the reference geography knows about. Run it on any crosswalk before tables are built on top of it.

Usage

```
rgnmapintg(map, unit, reg, regs = NULL)
```

Arguments

`map`	Data frame, one row per unit.
`unit`	Name of the column holding the unit identifier (a string).
`reg`	Name of the column holding the region identifier (a string).
`regs`	Optional character vector of every valid region identifier, typically the identifier column of the population table. When supplied, region codes outside it are counted as failures; when `NULL` (the default) that check is not run and its row is absent.

Details

These are the failures that a comparison against published output cannot see, because they would be present on both sides: a unit matched into two regions, a unit matched into none, a region code that is a typo or belongs to a neighbouring province. None of them require the geometry, so they run with nothing installed.

The checks as computed: "units assigned more than one region" counts rows whose (non-blank) unit identifier repeats an earlier row's, whether or not the region differs, so an exact duplicate row also fails; "units assigned no region" counts rows whose region is `NA` or blank; "region codes not in the reference geography" counts non-blank region codes not in `regs`, comparing the codes as text after trimming spaces (so `"3506"` and `3506` match, but `"03506"` does not). Errors: `map` not a data frame, or without the named columns.

Value

A data frame with one row per check: `check` (character), `observed` (numeric count), `expected` (numeric, always 0) and `pass` (logical). Every check is stated so that zero is the passing value, which is what makes the frame safe to feed straight into a manifest.

See Also

rgnmapcmp(), rgnmapalt(), rgncov()

Examples

```
cw <- data.frame(
  inst = c("North Jail", "South Jail", "East Jail"),
  cd = c("3557", "3520", "3506"),
  stringsAsFactors = FALSE
)
rgnmapintg(cw, "inst", "cd", regs = c("3557", "3520", "3506"))

# a region code the geography does not know fails the third check
cw$cd[3] <- "2406"
rgnmapintg(cw, "inst", "cd", regs = c("3557", "3520", "3506"))

# a unit listed twice and a unit with no region, without a reference list
cw2 <- rbind(
  cw,
  data.frame(inst = c("North Jail", "West Jail"), cd = c("3560", ""))
)
rgnmapintg(cw2, "inst", "cd")

# the columns must exist
try(rgnmapintg(cw, "institution", "cd"))
```

---

rgnmappts *Recompute a region map by point in polygon*

Description

Assigns each point (a facility's longitude and latitude) to the polygon of a boundary file that contains it, and carries the polygon's identifier and name columns onto the result. Requires the `sf` package and a boundary file; returns `NULL` when either is absent, so a verification script can record the check as unavailable instead of failing for a missing optional dependency.

Usage

```
rgnmappts(x, y, unit, bounds, fields, crs = 4326)
```

Arguments

`x`, `y`	Longitude and latitude of each point (numeric vectors, no `NA`), in the coordinate reference system `crs`.
`unit`	Identifier for each point, same length (character, no missing or empty values).
`bounds`	Path to a boundary file `sf` can read (a shapefile, GeoPackage, GeoJSON, ...).
`fields`	Character vector of columns of the boundary file to carry onto the result.

`crs` Coordinate reference system the points are in, anything `sf::st_crs()` accepts. Default `4326`, that is WGS84, which is what published latitude and longitude columns almost always are.

Details

The points are projected onto the boundary file's own coordinate reference system before matching, never the other way round: a cartographic boundary file is published in a projection chosen for the country it covers, and reprojecting the polygons to compare them against unprojected points moves the edges. Containment is `sf::st_within()`, so a point exactly on a shared edge is in neither polygon. The whole boundary file is read into memory with `sf::st_read()`; national files can take some seconds.

Point in polygon is preferred to matching place names against subdivision names whenever both are available. Names fail on exactly the cases that matter and fail quietly: a facility in a community rather than an incorporated municipality, a city amalgamated into a larger one, a township absorbed by a neighbour. The geometry has no opinion about any of that.

`rgnmapalt()` checks this result against the name route, with those cases named, and `rgnmapcmp()` compares it with a published map.

Errors (when `sf` and the file are present): missing or empty `unit`; `x`, `y` and `unit` of different lengths; missing coordinates; a field the boundary file does not have.

Value

A data frame: `unit`, the requested `fields` (factors converted to character; `NA` for a point in no polygon, the first polygon's values for a point in several), and `nreg` (integer), the number of polygons that contained the point. Any value of `nreg` other than 1 is a failure – zero means the point fell outside the geography, more than one means the boundaries overlap – so it is returned rather than silently resolved. `NULL` if `sf` is not installed or `boundaries` does not exist.

See Also

`rgnmapcmp()`, `rgnmapalt()`

Examples

```
# Two square regions as a small GeoJSON boundary file.
gj <- tempfile(fileext = ".geojson")
writeLines(
  paste0(
    '{"type":"FeatureCollection","features":[',
    '{"type":"Feature","properties":{"CDUID":"A","CDNAME":"West"},"geo',
    'metry":',
    '{"type":"Polygon","coordinates":[[[-80,43],[-79,43],[-79,44],[-80',
    ',44],[-80,43]]]}},',
    '{"type":"Feature","properties":{"CDUID":"B","CDNAME":"East"},"geo',
    'metry":',
    '{"type":"Polygon","coordinates":[[[-79,43],[-78,43],[-78,44],[-79',
    ',44],[-79,43]]]}}]}'
  ),
  gj
```

```
)

if (requireNamespace("sf", quietly = TRUE)) {
  obs <- rgnmappts(
    x = c(-79.5, -78.5, -70),
    y = c(43.5, 43.5, 43.5),
    unit = c("North Jail", "South Jail", "Far Jail"),
    bounds = gj,
    fields = c("CDUID", "CDNAME")
  )
  print(obs)
  # the point outside both regions is reported, not dropped
  obs$unit[obs$nreg != 1]

  # a field the file does not have is an error
  print(try(rgnmappts(-79.5, 43.5, "North Jail", gj, fields = "PRUID")))
}

# no boundary file: NULL, so a script can record the check as unavailable
rgnmappts(
  -79.5,
  43.5,
  "North Jail",
  bounds = file.path(tempdir(), "missing.shp"),
  fields = "CDUID"
)
unlink(gj)
```

`rlblfrns` *Name the mechanical step that reproduces a label permutation*

Description

When categories come out permuted (each true group reported under another group's label), this function asks which deterministic step, applied to the code book, produces exactly that permutation. It tries the known mechanisms: labels sorted alphabetically (also reversed or case-insensitive) and assigned by code position, labels reversed, every rotation, codes sorted as strings, and, when counts are given, labels ordered by frequency. A match is a reconstruction, not proof of intent; but a file transfer has no reason to pick the sort order of the labels, so a match on a sort-based mechanism points to a positional relabel in the analysis rather than to the software.

Usage

```
rlblfrns(vlabs, obs, cnts = NULL)
```

Arguments

`vlabs`	Named character vector, code = true label, in code order (the source program's code book).

obs	Named character vector, true label = label it was seen under, covering every label once.
cnts	Optional named numeric vector of frequencies per true label, default NULL. Enables the two frequency-order mechanisms.

Details

With labels $L_1, \ldots, L_k$ in code order, each mechanism produces a sequence $M_1, \ldots, M_k$ meaning "true group L_i was seen as M_i". It matches when $M_i =$ observed[L_i] for all i. There are five fixed mechanisms, $k - 1$ rotations, and two frequency orders with cnts. When obs is the identity no mechanism is reported as matching.

Errors: either argument unnamed; obs not a permutation of the labels in vlabs.

Value

A data frame of class c("bricklayer_relabel_forensics", "data.frame") with one row per mechanism tried and columns mechanism (character), permutation (character, "A -> B, ...") and matches (logical). The verdict attribute is a character summary. The print method shows only the matching mechanisms and the verdict.

See Also

vrfmrg, relbl.

Examples

```
# the OHRC 2023 case: alphabetical labels assigned to codes 1-4
vl <- c("1" = "White", "2" = "Black", "3" = "Other", "4" = "Unknown")
obs <- c(White = "Black", Black = "Other", Other = "Unknown",
         Unknown = "White")
rlblfrns(vl, obs)

# a frequency-ordered relabel needs the counts
obs2 <- c(White = "White", Black = "Other", Other = "Black",
          Unknown = "Unknown")
rf <- rlblfrns(vl, obs2, cnts = c(White = 50, Black = 10,
                                  Other = 30, Unknown = 5))
rf$mechanism[rf$matches]

# observed must be a permutation of the labels
try(rlblfrns(vl, c(White = "Black")))
```

`rmblapp` *Open the analysis dashboard*

Description

A local web app over the package's analyses: every tab opens on a demonstration dataset (`rmbldemo()`) with its first analysis already run, and you can load a CSV file, the pinned data of a capsule, or a table from data.rmorie.com (after `rmbl login`) instead. Everything runs on this machine in this R session; nothing is sent anywhere except the data.rmorie.com requests you make.

Usage

```
rmblapp(data = NULL, capsule = NULL, ...)
```

Arguments

`data`	A data frame, or the path of a CSV file, to start with; `NULL` for none.
`capsule`	The directory of a capsule whose pinned data to start with; `NULL` for none.
`...`	Options passed to `shiny::shinyApp()`, such as `port` and `launch.browser`.

Details

Each analysis page shows the headline numbers (estimate, interval, n and a verdict) in value boxes, the package's plot in a card that opens full screen and downloads as PNG, SVG or CSV, a diagnostics card (the checks and secondary tables the function returns, residual and goodness-of-fit plots where the method has them, effect sizes with a line on how to read them), the table, and the R call it ran, which runs exactly that call, so a result can be reproduced in a script by copying the code. The method card states the method in the words of its help page, its assumptions and what the numbers mean. A refusal is shown as the package words it, with `explerr()`'s explanation of where it comes from beside it, and as a toast. The resampling analyses (bootstraps, permutations, simulations and the Hawkes fits) run in a forked background process on Unix-alikes, so the page stays live while they work. "Share this view" in the navbar gives a link that reopens the page with the same tab, data and form.

The menus and their tabs:

Data Describe: `profcols()`, `freqtbl()`, `cortbl()`, `mahaoutl()`, `missgrid()` and the distribution plots (`plotrdg()`, `plotecdf()`, `plotvln()`, `plothex()`, `plotgain()`); click a column of the profile to see its distribution. Distributions: `dstfit()` with its density, QQ, PP and tail plots, and a synthetic sample from a fit through `mksyncol()`.

Over time Year over year: `yoy()` (hover a period of its plot for the period's rows) and the `yoyhtml()` page to download. Rates: `rtcalc()`, `ratechg()` and `shrs()`. Trends: `trndtst()` (Mann-Kendall with the Theil-Sen slope), `stepchg()`, `cnttrnd()`, and `plotcal()`, `plothrz()`, `plotsm()` and `plotbump()`.

Causal and survey Survey: `svydes()` with weights, strata, clusters or replicate weights, then `svyest()` or `svycmp()`. Policy: `plndid()`, `evstudy()`, `stgatt()`, `bacondcmp()`, `swtdid()`, `perminf()`, `syndid()`, `mtchdid()`, `wcb()`, `loounit()`, `cohtrnd()`, `plnpwr()`, `baltab()` and `tse()`.

Capsules Drift: `cpsdrft()` on two snapshots, with a density whose brushed range gives each snapshot's share of rows, and the app of `drftapp()` for two CSV files. Capsule: the provenance of a capsule (source, pinned digest, manifest, attestation, bundle files) and its verification rows, with buttons that run `vrfcps()` and `cpschk()`, or `cpsrfrsh()`. Proof ledger: a ledger (`ldgrread()`) from the demo, a file, a capsule or an installed package, its theorems with their status and covered functions, and `ldgrchk()`.

Domains First Nations, Metis and Inuit: the bundled Statistics Canada tables (`indiex()`) or a table in the same layout: `indgap()`, `dstntst()`, `ordcmp()`, `welprof()` with `dualcont()`, `healchg()`, `mhcont()`, `stdprev()` and `trcprog()`. Each distinction is reported on its own, suppressed cells stay suppressed with their quality flags, and the page states the OCAP and CARE principles and the source tables. Courts and corrections: `caseflow()`, `stkflow()`, `clrdcmp()`, `dispmix()`, `lifetbl()`, `crmdens()`, `knoxtst()`, `rgnmapintg()`, and the SIU report readers `siuprs()` and `siuso()` on pasted text. Air pollution: `plmfld()` and `gplume()` for a Gaussian plume, the monitors of the demo series (`plotsm()`), `crfpm25()`, `crfno2()`, `polbrdn()` and `vrfpol()`. Hawkes: `chwkfit()` with `plothwk()` (intensity, time-rescaling QQ, branching tree), `chwksim()` and `chwkjit()`. Search and rescue: the search theory functions behind one form, `ltrlrng()`, `srchpod()`, `pswppod()`, `datumpmap()`, `opteffort()` and `optcrcsrch()`, with the POD and POS numbers and four plots, and any figure of Frost's The Theory of Search through `srchfig()`.

Report the results added from the tabs, as one self-contained HTML page (plots as inline SVG, year-over-year tables as `yoyhtml()` draws them), as Markdown, or as a GitHub comment to copy.

Settings light, dark or the system's theme (also the switch in the navbar), digits, and the seed of the seeded analyses.

The data key of data.rmorie.com is never held by the app: `dload()` reads it from where `rmbl login` stored it.

The dashboard needs the shiny package (Suggests), and with it bslib, which shiny installs; without shiny `rmblapp()` refuses.

Value

A shiny app object; printing it (or `shiny::runApp()`) starts it.

See Also

`rmbl app` on the command line (`blcli()`); `rmbldemo()` for the datasets the tabs open on.

Examples

```
rmblapp()
rmblapp(indiex("health"))
```

`rmbldemo` *Demonstration datasets*

Description

Seeded synthetic datasets, made up but shaped like the data each family of the package reads, so that every tab of `rmblapp()` (and any example) has something realistic to run on without a download. Each call with the same `name` returns the same data, and the caller's random number stream is left as it was.

Usage

```
rmbldemo(name = NULL)
```

Arguments

`name`	The dataset, one of the names `rmbldemo()` lists; `NULL` (the default) lists them.

Details

Every dataset is generated in R from a fixed seed, with `mksyncol()` for the categorical and count columns and base R's random number generators for the rest. None of it describes real people, courts or places.

`court` 576 rows: 16 courts by 36 months (2022 to 2024). Columns `court`, `region`, `month` (1 to 36), `date`, `received`, `disposed`, `pending` (the backlog carried forward), `clearance` (disposed per 100 received), `caseload` (cases received per judge), `delay` (median days to disposition), `reformed` (1 from the month a court adopted the reform) and `refmonth` (the month it did, `NA` for the eight that never did). Eight courts adopt a case-management reform in months 13, 19, 25 or 31; it lowers `delay` by 12 days, and the courts share a seasonal cycle and a common trend, so the parallel-trends assumption holds.

`offences` 1,440 rows: monthly offence counts for 10 areas, 2013 to 2024. Columns `area`, `year`, `month`, `date`, `n` (Poisson counts with a summer peak of about 15 percent) and `pop` (growing 1 percent a year). From April 2020 every area's level steps down by 28 percent and stays there.

`offences_annual` 120 rows: `offences` summed by area and year (`n`), with the mean population (`pop`).

`survey` 1,200 respondents of a stratified two-stage sample: 4 strata (`stratum`), 10 primary sampling units in each (`psu`), design weights (`w`), and 20 bootstrap replicate weights `rw01` to `rw20` (Rao-Wu: primary units drawn with replacement within each stratum). Variables `age`, `sex`, `smoker` (0/1), `drinks` per week, `victim` (0/1, victimized in the past year), `trust` (1 to 5) and `income` (with 8 percent missing and a few very large values).

`fnmi` 128 rows in the layout of `indiex()`: `period` (2015/2018, 2019/2022), `group` (First Nations, Metis, Inuit, Non-Indigenous), `age`, `indicator` (four health indicators), `estimate`, `lower`, `upper` (95 percent) and `quality` (A to E by coefficient of variation; F, with the estimate and its interval suppressed, above 33.3 percent).

`otis` 144 rows: admissions to 12 institutions in 2013 to 2024, as a stock and flow table. Columns `inst`, `cd` (census division), `region`, `year`, `people` (admissions), `days` (days in custody of those admitted) and `releases`. Lengths of stay are log-normal and drift up after 2019.

`air` 2,016 rows: hourly NO2 at 6 monitors for 14 days around one stack (10 g/s, effective height 50 m). Columns `monitor`, `x`, `y`, `z` (metres from the stack), `hour`, `wind_from` (degrees), `wind_speed` (m/s), `stability` (Pasquill-Gifford class) and `no2` (micrograms per cubic metre): a diurnal background plus the stack's `gplume()` concentration at the monitor for that hour's wind.

`hawkes` About 700 event times (`t`, in days, over 365 days) from a self-exciting process with a baseline of 0.9 events a day, branching ratio 0.55 and an exponential kernel of rate 0.8 a day, simulated by thinning; `date` and `district` are added.

`drift` 3,000 rows: a reference and a current snapshot (`snapshot`) of 1,500 rows each, with `age`, `income`, `region`, `visits` and `satisfied`. In the current snapshot age is 4 years older on average, income 16 percent higher, the North region larger and satisfaction lower; visits do not change.

`waits` 1,000 court cases: `case_id`, `court_type`, `days` to disposition (a gamma body with a heavy log-normal tail) and `appearances` (zero-inflated Poisson).

`ledger` A proof ledger of 120 theorems (proved, sorry or axiom) covering the package's functions, written to a temporary JSON file and read back with `ldgrread()`; four theorems carry checks that `ldgrchk()` runs.

Value

A data frame; for `"ledger"` the ledger as `ldgrread()` returns it. With `name = NULL`, a data frame of `name`, `rows` and `description`.

See Also

`rmblapp()`, whose tabs open on these datasets; `mksyncol()`.

Examples

```
rmbldemo()
head(rmbldemo("court"))
identical(rmbldemo("survey"), rmbldemo("survey"))
```

---

rndbytes *Cryptographically strong random bytes*

Description

Returns `n` random bytes that are suitable for keys, salts and nonces. Use it whenever a value must be unguessable; use `set.seed()` and `runif()` for simulations, where reproducibility is the point.

Usage

```
rndbytes(n)
```

Arguments

n	Whole number from 1 to 1048576: the number of bytes.

Details

Reads the operating system's own random source rather than R's Mersenne Twister: `/dev/urandom` on Unix and macOS, `RtlGenRandom` on Windows.

This distinction matters for anything that becomes a key. `set.seed()` makes R's generator reproducible BY DESIGN, and its state can be recovered from its output; a key drawn from it is guessable. Reading the OS source also leaves R's own random stream untouched, so generating a key does not perturb a reproducible analysis.

If no OS source can be read the function FAILS rather than falling back to a weaker generator, because a silent downgrade in a key is worse than an error.

Errors: `n` not a single whole number from 1 to 1,048,576 (1 MiB); no readable operating-system source.

Value

A raw vector of length `n`. Print it as hex with `paste(format(x), collapse = "")`.

See Also

`drvkey()` to stretch a passphrase instead, `pqckg()` which uses this for its seeds.

Examples

```
rndbytes(8)

# Independent between calls, unlike a seeded generator.
identical(rndbytes(16), rndbytes(16))

# R's own stream is not consumed, so a seeded analysis is unaffected.
set.seed(1)
a <- stats::runif(1)
set.seed(1)
invisible(rndbytes(32))
identical(stats::runif(1), a)

# As hex, for a seed argument.
paste(format(rndbytes(4)), collapse = "")

try(rndbytes(0))
```

`rptanl` *Write an analysis as a Markdown or HTML report*

Description

Renders an `anltab()` result as a report: the change table with its intervals and adjusted p-values, the rates, the trends, the drift screen and the column profile, each with one sentence saying how to read it. The Markdown is plain enough to paste into a notebook or a pull request; the HTML is a single self-contained file with no external resources, suitable for e-mailing to a reviewer.

Usage

```
rptanl(
  x,
  path = NULL,
  format = NULL,
  title = "Analysis of a published table",
  digits = 1L
)
```

Arguments

`x`	A `bricklayer_analysis` from `anltab()`.
`path`	Where to write, a file path. With `NULL` (the default) the text is returned invisibly and nothing is written.
`format`	`"markdown"` or `"html"`. With `NULL` (the default), a `path` ending in `.html` or `.htm` gives HTML and anything else, or no `path`, gives Markdown.
`title`	Report title, a string; the top heading and, for HTML, the page title.
`digits`	Number of decimals for percentages and rates, default 1.

Details

Sections appear in this order: a header line (value, period, groups, rows, period range and the time of the analysis), “Change between periods”, “Rates per ...” (when there were rates), “Trend over the series”, “Drift against the prior capsule” (when a prior was given) and “What is in the table” (with “Missing values” when any are missing). Tables use pipe syntax; numbers are rounded to `digits` decimals, whole numbers of 10,000 and above get thousands separators, logicals are written yes/no and `NA` is blank. When the values are percentages the change is reported in percentage points. The HTML is produced by a small built-in converter (headings, paragraphs, tables and inline code, with the text escaped). The file is written as UTF-8 and overwritten if it exists.

Errors: `x` not from `anltab()`; an invalid `format`.

Value

The report text (a length-1 character string, Markdown or a full HTML document), invisibly.

Examples

```
path <- system.file("extdata", "otis_a01_individuals.csv",
                    package = "rmoriebricklayer")
a <- anltab(read.csv(path), value = "individuals",
                   prd = "year", by = c("table", "group"))
md <- rptanl(a)
cat(substr(md, 1, 400))

# HTML, chosen from the file extension
out <- tempfile(fileext = ".html")
rptanl(a, out, title = "OTIS A01: individuals in custody", digits = 2)
readLines(out, n = 3)

# only an anltab() result can be reported
try(rptanl(read.csv(path)))
unlink(out)
```

`rptmd` *Write a capsule report as Markdown*

Description

Renders a `cpsrpt()` as Markdown, so the assessment can travel with the capsule instead of living in a console someone has since closed. The document renders on GitHub, in R Markdown and Quarto, and in any Markdown viewer, and reads well as plain text.

Usage

```
rptmd(report, path = NULL, title = "Capsule report")
```

Arguments

`report`	A `bricklayer_report` from `cpsrpt()`.
`path`	Optional character scalar, the file to write. Without one (`NULL`, the default) the lines are returned.
`title`	Character scalar, the heading for the document; default `"Capsule report"`.

Details

The document has a level-one heading (`title`), the verdict in one sentence, a two-column table of rows, columns, missing cells, complete rows and the full data digest (with the Merkle root and the signature result when the report has them), a `"Findings"` table of severity, check, subject and detail (left out when there is none), and a closing caveat that no finding is proof of correctness. The text of the findings is written as it is: a `|` in a column name would break the table. An existing file at `path` is overwritten. Errors when `report` does not come from `cpsrpt()`.

Value

A character vector of Markdown lines; returned visibly when `path` is `NULL`, invisibly when written to a file.

See Also

`cpsrpt()`

Examples

```
set.seed(1)
df <- data.frame(v = stats::rnorm(100), g = rep("x", 100),
                 stringsAsFactors = FALSE)
r <- cpsrpt(df)

md <- rptmd(r)
cat(head(md, 8), sep = "\n")

# Written beside the capsule it describes.
p <- tempfile(fileext = ".md")
rptmd(r, p)
file.exists(p)
unlink(p)

# a report with findings, under its own title
r2 <- cpsrpt(transform(df, v = ifelse(seq_along(v) <= 60, NA, v)))
md2 <- rptmd(r2, title = "Extract of 2024-03-01")
md2[1]
md2[grep("^## Findings", md2) + 0:4]

# only a cpsrpt() result is accepted
try(rptmd(list(verdict = "clean")))
```

`rslvarc` *Resolve a Query URL via ArcGIS FeatureServer Metadata*

Description

Checks that an ArcGIS FeatureServer layer still exists and returns a query URL that downloads every feature of the layer as GeoJSON. FeatureServer layers back the Toronto Police Service open-data portal and many municipal portals used across the MORIE family.

Usage

```
rslvarc(prov)
```

Arguments

prov	A provenance list as returned by loadprov(), or NULL. It must have the field dataset$arcgis_layer_url, a FeatureServer layer root. An example is https://services.arcgis.com/.../ followed by Assault_Open_Data/FeatureServer/0.

Details

Trailing slashes are dropped from dataset$arcgis_layer_url, the URL is checked (public https only), and one request is made to <layer>?f=json. If the answer is JSON with a name and no error element, the function returns <layer>/query?where=1%3D1&outFields=*&f=geojson: every row (where=1=1), every field, as GeoJSON. ArcGIS servers return at most their maxRecordCount features per request, so a large layer needs paging (resultOffset) by the caller; the URL is not fetched here.

NULL is returned when prov is NULL, when the field is missing, when the request fails or when the metadata reports an error. Errors: a prov that is not a list; a layer URL that is not public https.

Value

The layer query URL as a length-1 character string, or NULL if the field is missing, the request fails, or the layer metadata reports an error.

Examples

```
# Missing fields return NULL rather than erroring:
rslvarc(list())

# only https layer URLs are accepted
try(rslvarc(list(dataset = list(
  arcgis_layer_url = "ftp://example.org/FeatureServer/0"))))
try(rslvarc("not a list"))


# needs the network
prov <- list(dataset = list(arcgis_layer_url = paste0(
  "https://services.arcgis.com/S9th0jAJ7bqgIRjw/arcgis/rest/services/",
  "Neighbourhood_Crime_Rates_Open_Data/FeatureServer/0"
)))
rslvarc(prov)
```

---

rslvckan *Resolve a Download URL via CKAN package_show*

Description

Asks a CKAN open-data portal for the current list of files in a dataset and returns the URL of the first file whose name matches the pattern pinned in the provenance record. CKAN powers data.ontario.ca, data.gov.uk, data.gov and most government open-data portals; when a portal replaces a file (a new resource UUID, a new URL) the dataset page stays the same, so this recovers the current download URL without editing the project.

Usage

```
rslvckan(prov)
```

Arguments

`prov`	A provenance list as returned by `loadprov()`, or `NULL`. A lookup is made only when it has both of these fields: • `dataset$ckan_api_endpoint`; • `resource$name_match_pattern`.

Details

The function reads `dataset$ckan_api_endpoint`, the full URL, which ends in `/api/3/action/` and `package_show?id=<slug>`. It also reads `resource$name_match_pattern`, a regular expression. It fetches the endpoint as JSON, and walks `result$resources` in the order the portal lists them. The first resource whose `name` matches the pattern (case-insensitively, with `grepl()`) wins. If the slug itself has changed, `rslvckns()` is the fallback.

`NULL` is returned, without an error, when `prov` is `NULL`, when either field is missing or empty, when the request fails (offline, HTTP error, invalid JSON), when CKAN answers `"success": false`, or when no resource name matches. Errors are raised for a `prov` that is not a list, for an endpoint that is not a public `https` URL (plain `http` only when the option `rmoriebricklayer.allowhttp` is `TRUE`; loopback, private and link-local addresses are refused), and for a matched resource URL that fails the same check. One network request; nothing is written.

Value

The matched resource URL as a length-1 character string, or `NULL` as described in Details.

Examples

```
# Missing fields return NULL rather than erroring:
rslvckan(list())
rslvckan(NULL)

# the endpoint must be public https, checked before any request
prov <- list(
  dataset = list(
    ckan_api_endpoint = "http://localhost/api/3/action/package_show?id=x"
  ),
  resource = list(name_match_pattern = "2014")
)
```

```
try(rslvckan(prov))


# needs the network: asks data.ontario.ca for the current file URL
prov <- list(
  dataset = list(ckan_api_endpoint = paste0(
    "https://data.ontario.ca/api/3/action/package_show",
    "?id=ontario-public-library-statistics"
  )),
  resource = list(name_match_pattern = "2014")
)
rslvckan(prov)
```

`rslvckns` *Resolve a Download URL via CKAN package_search*

Description

The fallback for `rslvckan()` when a dataset's slug has changed and `package_show` no longer finds it. It runs a full-text `package_search` on the same CKAN portal and returns the URL of the first resource, across all datasets returned, whose name matches the pinned pattern.

Usage

```
rslvckns(prov)
```

Arguments

`prov`	A provenance list as returned by `loadprov()`, or `NULL`. The fields read are `dataset$ckan_api_endpoint`, `resource$name_match_pattern` (a regular expression, required for a match), and, both optional, `resource$search_query` (search words) and `resource$format` (a format name such as `"XLSX"`).

Details

The portal base is the part of `dataset$ckan_api_endpoint` before `/api/3/`; the request goes to `<base>/api/3/action/` followed by `package_search?q=<query>`.

The query is `resource$search_query` when given, otherwise the `resource$name_match_pattern` with every character other than a letter, digit or space replaced by a space. Datasets are walked in the order CKAN ranks them, and their resources in listed order. A resource is taken when its `name` matches `resource$name_match_pattern` (case-insensitively) and either its format is CSV, or `resource$format` is unset, or its format equals `resource$format` (case-insensitively). So a CSV resource is accepted even when `resource$format` asks for another format.

`NULL` is returned when `prov` is `NULL`, when no query can be formed, when there is no endpoint to derive the base from, when the request fails, or when nothing matches. Errors: a `prov` that is not a list; a search URL or matched resource URL that is not public `https`. One network request.

Value

The matched resource URL as a length-1 character string, or NULL.

Examples

```
# Missing fields return NULL rather than erroring:
rslvckns(list())

# no endpoint to derive the portal from: NULL, no request made
rslvckns(list(resource = list(name_match_pattern = "2014")))

# a provenance must be a list
try(rslvckns("data.ontario.ca"))


# needs the network: searches data.ontario.ca
prov <- list(
  dataset = list(ckan_api_endpoint = paste0(
    "https://data.ontario.ca/api/3/action/package_show",
    "?id=ontario-public-library-statistics"
  )),
  resource = list(
    name_match_pattern = "2014",
    search_query = "public library statistics"
  )
)
rslvckns(prov)
```

---

rslvsoc *Resolve a Download URL via the Socrata Metadata API*

Description

Checks that a dataset still exists on a Socrata open-data portal and returns its full CSV export URL. Socrata powers the Calgary, Chicago and New York City portals used across the MORIE family; a dataset is named by its portal domain and a four-by-four id such as `ijzp-q8t2`.

Usage

```
rslvsoc(prov)
```

Arguments

prov	A provenance list as returned by loadprov(), or NULL. It must have two fields. The first is dataset$socrata_domain, a host name without scheme such as "data.cityofchicago.org". The second is dataset$socrata_id, the four-by-four dataset id such as "ijzp-q8t2".

Details

One request is made, to `https://<domain>/api/views/<id>.json`. If it answers with JSON that has an `id` field the dataset is taken to exist. The function then returns the portal's export of every row, the URL `https://<domain>/api/views/<id>/` followed by `rows.csv?accessType=DOWNLOAD`. The id is not checked further, and the export URL is not fetched.

`NULL` is returned when `prov` is `NULL`, when the domain or id is missing, when the request fails (offline, HTTP error, invalid JSON) or when the metadata has no `id`. Errors: a `prov` that is not a list; a domain that is not a public host (`localhost`, a private address, a single-label name).

Value

The CSV export URL as a length-1 character string, or `NULL` if the fields are missing, the request fails, or the metadata reports an error.

Examples

```
# Missing fields return NULL rather than erroring:
rslvsoc(list())
rslvsoc(list(dataset = list(socrata_domain = "data.cityofchicago.org")))

# a local host is refused before any request
try(rslvsoc(list(dataset = list(
  socrata_domain = "localhost", socrata_id = "abcd-1234"))))


# needs the network
prov <- list(dataset = list(
  socrata_domain = "data.cityofchicago.org",
  socrata_id = "ijzp-q8t2"
))
rslvsoc(prov)
```

---

rsv *Uniform sample of a stream in one pass*

Description

Vitter's Algorithm R: draws `k` items from a stream, each item equally likely to be retained, in a single pass and using memory proportional to `k` rather than to the stream's length. Use it to take a fair sample of a capsule member you cannot hold, or whose length you do not know in advance.

Usage

```
rsvidx(n, k, seed = 42L)

rsvsmp(x, k, seed = 42L)
```

Arguments

n	Length of the stream, one non-negative number.
k	Sample size, one non-negative number. Values above the stream length give the whole stream; 0 gives an empty result.
seed	Seed for the core's generator, a whole number from 0 to below 2^{53} (default 42).
x	Vector to sample from.

Details

`rsvidx()` returns the retained positions, so the same sample can be applied to a file read line by line. `rsvsmp()` applies it to a vector in memory.

The first `k` items fill the reservoir; item $i > k$ then replaces a uniformly chosen slot with probability k/i, so after n items every item is in the sample with probability k/n (Vitter 1985). The positions are returned sorted.

The stream is sampled with the core's own generator seeded by `seed`, not R's, so the sample is reproducible and R's random stream is left alone – a capsule whose sample changed between runs would not be reproducible. The generator is C++ `std::mt19937_64`, whose output sequence the C++ standard fixes, and each draw is Lemire's unbiased multiply-and-reject on its 64-bit outputs (Lemire 2019), so the same seed gives the same sample on every platform and C++ library. Versions up to 0.5.12 drew with `std::uniform_int_distribution`, whose algorithm differs between C++ libraries, so their samples for a seed differ from these.

Errors: `n` or `k` not one non-negative whole number; `n` above 2^{53}; `k` above 10^8; `seed` not a whole number from 0 to below 2^{53} (`NA` included).

Value

`rsvidx()` a sorted numeric (double) vector of 1-based positions, of length `min(k, n)`; `rsvsmp()` the corresponding elements of `x`, in their original order.

References

Vitter JS (1985). Random sampling with a reservoir. *ACM Transactions on Mathematical Software* 11(1), 37–57. DOI 10.1145/3147.3165

Lemire D (2019). Fast random integer generation in an interval. *ACM Transactions on Modeling and Computer Simulation* 29(1), 3. DOI 10.1145/3230636

Examples

```
# A reproducible sample of 5 from 1000.
rsvidx(1000, 5, seed = 1)
rsvsmp(letters, 4, seed = 2)

# Reproducible on every platform, and independent of R's own RNG.
identical(rsvidx(1000, 5, seed = 1),
          rsvidx(1000, 5, seed = 1))

# Asking for more than the stream holds returns the whole stream.
```

```
rsvidx(3, 10)

# Every item is equally likely: over many seeds the retained positions
# are spread uniformly rather than favouring the start or the end.
hits <- unlist(lapply(1:400, function(s) rsvidx(50, 5, s)))
round(mean(hits))          # near the midpoint, 25.5

# sample rows of a data frame
df <- data.frame(id = 1:20, v = (1:20)^2)
df[rsvidx(nrow(df), 3, seed = 7), ]

# a negative size is refused
try(rsvidx(10, -1))
```

`rtcalc` *Event rates per unit of population*

Description

Turns counts of events (arrests, stops, admissions, deaths) into rates per unit of exposure, such as per 1,000 or per 100,000 residents, with an exact Poisson confidence interval for each rate. A count is not comparable across areas of different size or years of different population; a rate is. The interval says how much of a rate is signal: three events in a small area give a rate whose interval is several times its own width.

Usage

```
rtcalc(x, ...)

## S3 method for class 'data.frame'
rtcalc(x, count, pop, by = NULL, per = 1000,
  conflvl = 0.95, mincnt = 0, ...)

## Default S3 method:
rtcalc(x, pop, per = 1000, conflvl = 0.95, mincnt = 0, ...)
```

Arguments

`x`	A data frame, or a numeric vector of counts.
`...`	Passed to methods.
`count`	Column of counts, given unquoted or as a string: non-negative whole numbers.
`pop`	For the data-frame method, the column of exposure, unquoted or as a string; for the default method, a numeric vector the length of `x` (or one number, recycled). Positive. Person-years, stops, residents – whatever the events were at risk of happening to.

by	Character vector of grouping columns, default NULL. The rate is computed within each group.
per	The rate denominator: a positive number, or "1k", "10k", "100k", "1m". Default 1000.
conflvl	Number in (0, 1), default 0.95: the confidence level of the interval.
mincnt	Number, default 0. Counts at or below this are flagged as too small to report, with the rate still computed. The default flags nothing; published guidance often uses 5 or 10.

Details

For count y and exposure t, the rate is $\text{per} \times y/t$. The interval is the exact Poisson interval for y, computed through the gamma relation,

$$\left[G^{-1}(\alpha/2;\, y),\; G^{-1}(1-\alpha/2;\, y+1)\right] \times \text{per}/t,$$

with $G^{-1}(p; k)$ the gamma quantile with shape k; a count of zero has a lower limit of exactly zero. This is the interval of `stats::poisson.test()`. It treats the exposure as known without error.

With a data frame, counts and exposures are summed within each combination of the by columns (missing values are ignored in the sums), and the groups come out in sorted order. Groups with no rows are not created. Without by the whole table is one group. With a vector, each element is its own row.

per takes a number, or one of "1k", "10k", "100k", "1m" (case, spaces, commas and a leading "per" are ignored), because a denominator mistyped by a factor of ten is invisible once it reaches a table.

A rate and a share are different quantities: use shrs when the denominator is the total of the same events, not a population.

Errors: count or population columns that are not numeric or not found; negative or fractional counts; a population of zero or below; conflvl outside (0, 1); an unknown per shorthand or a non-positive per.

Value

A data frame of class c("rmbl_rate", "data.frame") with the grouping columns (or label, for a named vector), count, population, rate, lower, upper (numeric, rates per per) and flag (character, NA when not flagged), plus a rate attribute: a list of count, population, by, per, perlbl (for example "per 100,000"), conflvl and mincnt. The print method titles the table with the denominator and the interval.

See Also

shrs for a percentage of a total, ratechg for the change in a rate between periods, stdinc for a rate compared against an expected count rather than a population.

Examples

```
stops <- data.frame(
  division = c("North", "South", "East"),
  stops = c(412, 77, 3),
  residents = c(120000, 41000, 9500))

# per 1,000 residents, the default
rtcalc(stops, stops, residents, by = "division")

# the denominator published guidance usually asks for
rtcalc(stops, stops, residents, by = "division", per = "100k")

# East's three events: the interval is wider than the estimate, and
# flagging it is the point of mincnt
rtcalc(stops, stops, residents, by = "division", per = "100k",
     mincnt = 5)

# vectors work too, for a single figure; a zero count has a lower limit of 0
rtcalc(c(3, 0), 9500, per = "100k")

# a fractional count has no Poisson interval
try(rtcalc(2.5, 1000))
```

`rul` *Declare a validation rule*

Description

Builds a rule for `valrules()`: a predicate over one column, or over the whole data frame, with a severity and a message. Use rules for the checks that are particular to a data set and that a schema cannot express: an age must be non-negative, an identifier must be unique, a release date must not precede an admission date. Rules live alongside a schema in a provenance record, which keeps the project-specific checks in data rather than scattered through code.

Usage

```
rul(name, expr, col = NULL, sev = c("warning", "fatal"), msg = NULL)
```

Arguments

`name`	A short identifier for the rule, a string. It names the issue in the result of `valrules()`, so give each rule a different name.
`expr`	A function, as described above.
`col`	Name of the column the rule applies to, or `NULL` (the default) for a table-level rule.
`sev`	`"warning"` (default) or `"fatal"`. A fatal failure stops `schval()`.
`msg`	Human-readable description of what a failure means, a string. Defaults to `"Rule '<name>' failed"`.

Details

`expr` is a function. Given `col`, it receives that column and must return a logical vector the same length (`TRUE` = the row passes) or a single logical for a whole-column property. Given no `col`, it receives the whole data frame and must return a single logical. How the result is read is described in `valrules()`.

Only the first element of `name`, `col` and `msg` is kept. The rule is not run here. Errors: an empty or `NA` `name`; `expr` not a function; an empty or `NA` `col`; a `sev` other than `"warning"` or `"fatal"`.

Value

A list of class `c("bricklayer_rule", "list")` with `name`, `expr`, `column` (`NULL` for a table rule), `severity` and `message`; it prints as one line.

See Also

`valrules()`, `valsch()` for the structural checks that need no rules.

Examples

```
# A column predicate, applied row-wise.
rul("agenonnegative", function(v) v >= 0, col = "age")

# A whole-column property.
rul("idunique", function(v) !anyDuplicated(v), col = "id",
     sev = "fatal")

# A table-level rule spanning two columns.
rul("datesordered", function(df) all(df$start <= df$end))

# with a message that says what a failure means
r <- rul("releaseafteradmission",
          function(d) all(d$release >= d$admission, na.rm = TRUE),
          msg = "a release date precedes its admission date")
r$message

# the predicate must be a function, not an expression
try(rul("agenonnegative", "age >= 0", col = "age"))
```

`rulelib` *Ready-made validation rules*

Description

Constructors for the checks most schemas need, each returning a `rul()` that `valrules()` evaluates. Use `rul()` directly for anything not covered here. Writing the common checks once, here, avoids the off-by-one or the forgotten `NA` that makes a hand-written rule pass silently, and gives every failure a readable message and the rows that failed.

Usage

```
ruleinset(col = NULL, set, napass = TRUE, sev = c("warning", "fatal"))

rulebtw(col = NULL, lo, hi, napass = TRUE, sev = c("warning", "fatal"))

rulenull(col = NULL, sev = c("warning", "fatal"))

ruleuniq(col = NULL, sev = c("warning", "fatal"))

rulerx(col = NULL, pat, napass = TRUE, sev = c("warning", "fatal"))

ruleincr(col = NULL, strict = FALSE, sev = c("warning", "fatal"))

rulemad(col = NULL, n = 3, napass = TRUE, sev = c("warning", "fatal"))

rulecmpl(sev = c("warning", "fatal"))

ruledstn(cols = NULL, sev = c("warning", "fatal"))

rulecolcnt(n, sev = c("warning", "fatal"))
```

Arguments

col	Character scalar, the name of the column the rule applies to.
set	Vector of allowed values (ruleinset()); compared as text.
napass	Logical scalar, default TRUE: treat NA as passing (see above); FALSE makes a missing value fail.
sev	"warning" (the default) or "fatal": how valrules() and the capsule gate treat a failure.
lo, hi	Numeric scalars, the inclusive bounds (rulebtw()); lo must not exceed hi.
pat	Character scalar, the regular expression the values must match (rulerx()); only the first element is used.
strict	Logical scalar, default FALSE: require a strict increase rather than non-decreasing (ruleincr()).
n	Numeric scalar: the multiplier for rulemad() (default 3), or the expected number of columns for rulecolcnt() (a whole number, no default).
cols	Character vector of columns that jointly must be unique (ruledstn()), or NULL (the default) for all of them.

Details

What each rule checks:

- ruleinset(): each value, compared as text, is one of set (so 1 and "1" match). Name "<column>_in_set".
- rulebtw(): each value lies in [lo, hi], ends included. Name "<column>_between".

- `rulenull()`: no value is `NA`. Name `"<column>_not_null"`.
- `ruleuniq()`: no value repeats an earlier one; the rows reported are the repeats, not the first occurrence. Two `NA`s count as a repeat. Name `"<column>_unique"`.
- `rulerx()`: each value, as text, matches `pattern` (an extended regular expression, `grepl()`). Name `"<column>_regex"`.
- `ruleincr()`: the column never decreases from one row to the next (or always increases, with `strictly = TRUE`); missing steps are skipped. A whole-column check: a failure names no rows. Name `"<column>_increasing"`.
- `rulemad()`: each value lies within `n` median absolute deviations (scaled by 1.4826, so comparable to standard deviations for normal data) of the column's median. When the MAD is zero (more than half the values equal) any value other than the median fails. The column must be numeric when the rule is evaluated. Name `"<column>_within_mads"`.
- `rulecmpl()`: no row has a missing value. Name `"complete_rows"`.
- `ruledstn()`: no row repeats an earlier one on `cols` (columns not in the data are ignored; when none is present the rule passes). Name `"distinct_rows"`.
- `rulecolcnt()`: the data have exactly `n` columns. Name `"col_count"`.

The name is the key of the rule's entry in the list `valrules()` returns, which holds only the rules that failed (with `sev`, `message`, and for row-wise rules `n_failed` and `rows`). A rule that cannot be evaluated (a missing column, text in a numeric rule) is reported as failed, with the reason in its message.

Errors, at construction: `col` not a single non-empty string; `sev` other than `"warning"` or `"fatal"`; `rulebtw()` with `lo > hi` or a missing bound; `rulemad()` with `n` not positive; `rulecolcnt()` with `n` not a non-negative whole number.

`NA` handling is explicit and per-rule, because the right answer differs. `rulenull()` exists precisely to fail on `NA`. The value rules (`ruleinset`, `rulebtw`, `rulerx`, `rulemad`) treat `NA` as PASSING by default, so that a column's missingness is reported once by `rulenull()` or `max_missing_fraction` rather than again by every other rule; set `napass = FALSE` to make them fail on it instead.

Value

A `rul()` object: a list of class `c("bricklayer_rule", "list")` with `name`, `expr` (the predicate function), `column` (`NULL` for whole-table rules), `severity` and `message`.

See Also

`rul()` for an arbitrary predicate, `valrules()` to evaluate them, `infsch()` for the structural checks that need no rules at all.

Examples

```
df <- data.frame(
  id = c(1, 2, 2),
  grade = c("a", "b", "z"),
  score = c(5, 200, 7),
  email = c("a@b.com", "nope", "c@d.org"),
  day = c(3, 1, 2),
```

```
  stringsAsFactors = FALSE
)

rules <- list(
  ruleuniq("id", sev = "fatal"),
  ruleinset("grade", c("a", "b", "c")),
  rulebtw("score", 0, 100),
  rulerx("email", "^[^@]+@[^@]+\\\\.[a-z]+$"),
  ruleincr("day")
)
names(valrules(df, rules))

# Each names the rows that failed.
valrules(df, rules)$grade_in_set$rows

# Clean data passes every one of them.
ok <- data.frame(
  id = 1:3,
  grade = c("a", "b", "c"),
  score = c(5, 50, 7),
  email = c("a@b.com", "c@d.org", "e@f.net"),
  day = 1:3,
  stringsAsFactors = FALSE
)
length(valrules(ok, rules))

# A robust outlier rule: MADs from the median, not standard
# deviations from the mean, so one wild value cannot hide the others.
valrules(
  data.frame(v = c(1, 2, 3, 2, 1, 900)),
  rulemad("v", 5)
)$v_within_mads$rows

# Whole-table rules.
valrules(df, ruledstn())
valrules(df, rulecolcnt(5))
valrules(df, rulecmpl())

# NA fails only when asked to
g <- data.frame(grade = c("a", NA, "b"))
length(valrules(g, ruleinset("grade", c("a", "b"))))
valrules(
  g,
  ruleinset("grade", c("a", "b"), napass = FALSE)
)$grade_in_set$rows

# bounds are checked when the rule is made
try(rulebtw("score", 100, 0))
```

`rvkfetch` *Fetch revocation data for a certificate path*

Description

Checks over the network whether any certificate in a chain has been revoked by its issuer. It downloads the certificate revocation lists (CRLs) that the certificates point to, and asks each OCSP responder they name about the certificate. It is called by `certchnvrf()` when `revocation = "fetch"`; you would call it directly only to inspect the raw revocation data. Nothing in the package does this unless asked, because a verifier that goes online stops working offline, can give different answers on different days, and tells the responder which certificates are being checked.

Usage

```
rvkfetch(path, tmout = 10)
```

Arguments

`path`	A certificate path, leaf first, as `certchnvrf()` builds it.
`tmout`	Number, default 10: seconds to allow each request.

Details

For each certificate, every http(s) URL in its CRL distribution points extension (2.5.29.31) is fetched; a PEM CRL is converted to DER. For each certificate except the last (which has no issuer in the path), every OCSP URL in its authority information access extension is asked about it. The OCSP request is sent by POST first, which RFC 6960 requires responders to accept, and if that gets no answer, by GET with the DER request base64-encoded into the URL, as RFC 6960 appendix A.1.1 allows. A responder that answers neither is reported as unreachable rather than treated as a pass.

A responder's answer is only believed when its signature verifies under a certificate in the path or one it carries that the path issued. An unsigned or unverifiable answer is reported as such – treating it as `"good"` would make revocation checking worse than skipping it, since it would look like it had happened.

The fetched CRLs are returned, not checked: `certchnvrf()` checks the certificates against them. Network: plain http is allowed here, because CRLs and OCSP answers are signed and RFC 5280 and RFC 6960 distribute them over http, but URLs that point at local or private addresses are refused (and reported as not retrieved). A certificate with no CRL or OCSP URL produces no request at all.

Errors: `path` not a non-empty list of parsed certificates (each a list with a raw `der` element); `tmout` not positive. Network failures are not errors; they are reported in `ok` and `notes`.

Value

A list with `crls` (a list of raw DER vectors fetched), `ok` (a logical vector, one element per CRL fetch or OCSP query, `TRUE` for a success) and `notes` (a named list of character details, named like `"crl fetch [1]"` or `"ocsp [1]"` by the certificate's position in the path; a later attempt for the same certificate overwrites an earlier note of the same name).

See Also

certchnvrf(), tsvrf().

Examples

```
# Offline: a path entry that names no CRL or OCSP URL (here, bytes that
# are not a certificate at all) makes no request and returns empty results.
str(rvkfetch(list(list(der = as.raw(1:3)))))

# the input checks
try(rvkfetch(list()))
try(rvkfetch(list(list(der = as.raw(1:3))), tmout = 0))

# Reaches the network, so it is not run here.


res <- certchnvrf("leaf.crt", trust = "ca.crt",
                          rvk = "fetch")
res$checks
```

---

scanadj *Adjust a scan of many comparisons for multiple testing*

Description

A change table over fifty institutions is fifty tests. Reporting the raw p-values invites reading the largest of fifty noise draws as a finding. This adds padj (Benjamini-Hochberg by default, which controls the false discovery rate over the scan) and significant at level alpha after adjustment. Exact p-values are computed where the object does not carry them: the conditional binomial test for yoy() counts and for ratechg() (with the exposures as the binomial weights).

Usage

```
scanadj(x, meth = "BH", alpha = 0.05)
```

Arguments

x	An rmbl_yoy or rmbl_rate_change object, or any data frame with a p_value column.
meth	Character, default "BH". It can be any method of p.adjust ("BH", "holm", "bonferroni", "BY", "hochberg", "hommel", "fdr", "none").
alpha	Number, default 0.05: the level at which significant is decided, applied to the adjusted p-values.

Details

For a `ratechg()` table, conditional on the two counts' total the current count is binomial with success probability $t_a/(t_a + t_b)$, the share of the combined exposure in the current period, and the p-value is the two-sided exact binomial test of that. Rows without a p-value (no comparison period, both counts zero) are left out of the adjustment, so the number of tests is the number of rows with a p-value; it is recorded in the `scanadj` attribute.

The adjustment is `stats::p.adjust(p, method)`: `"BH"` controls the expected proportion of false discoveries among the rows declared significant, which suits a screen; `"holm"` and `"bonferroni"` control the chance of any false discovery, which a claim about one specific row needs. A row is `significant` when its adjusted p-value is strictly below `alpha`.

Errors: `x` not a data frame; no `p_value` column and not a `yoy()` or `ratechg()` result; a `yoy()` result without count units; a p-value outside $[0, 1]$; a method unknown to `p.adjust()`.

Value

`x` with `p_value` (if it was missing), `padj` (numeric, `NA` where `p_value` is) and `significant` (logical) columns; the class and attributes of `x` are kept, and a `scanadj` attribute lists `method`, `alpha` and `tests`.

See Also

`yoypv`, `falsfam` for a family of falsification results, `drftcal`.

Examples

```
seg <- data.frame(year = rep(2022:2023, each = 3),
                  site = rep(c("A", "B", "C"), 2),
                  n = c(30, 50, 80, 45, 52, 79))
y <- yoy(seg, value = "n", prd = "year", by = "site")
scanadj(y)[, c("site", "pctchg", "p_value", "padj")]

# family-wise control instead of the false discovery rate
scanadj(y, meth = "holm")$padj

# any table with a p_value column
scanadj(data.frame(test = c("a", "b", "c"), p_value = c(0.001, 0.02, 0.4)))

try(scanadj(data.frame(p_value = 1.5)))
```

`scests` *The estimates of a Statistics Canada table, with their confidence limits*

Description

Statistics Canada's tables give each estimate on several rows: the estimate itself (a percent, a rate, a count, a life expectancy), its low and high confidence limits, or its coefficient of variation or standard error, all in one statistics column (`Statistics`, `Characteristics`, `Statistics (3)`, ...). This puts them side by side, one row per cell of the table and measure, ready for `indgap`, `ordcmp`, `stdprev` and the rest. Every table reads: one without a statistics column (a census count) gives each value as its own estimate, with no limits.

Usage

```
scests(x)
```

Arguments

x	A table read by `sctidy`, a data frame as downloaded, or a table key.

Details

Each label of the statistics column is read as an estimate or as one of its measures of precision, in every form the tables use:

- `"Low 95% confidence interval"`;
- `"High 95% confidence interval, age-standardized rate"`;
- `"Confidence interval lower bound of count"`;
- `"95% confidence interval (upper limit)"`;
- `"95% confidence interval lower bound, 2020"`;
- `"Coefficient of variation for percent"`;
- `"Standard error"`.

A limit that names its measure is matched to it (ignoring case and spacing); one that names none, or a name that fits several measures (`"potential years of life lost"` in a table with a number, a crude rate and an age-standardised rate of them), is matched to the measure with the same unit (`UOM`) whose estimates it contains in every cell, and left unmatched (said in the attribute `unmatched`) if no single measure fits. A coefficient of variation c (in percent) gives log-normal limits $e\exp(\mp z\sqrt{\log(1+(c/100)^2)})$, which stay positive however large c is; a standard error s the limits $e \mp zs$; the column `cisou` says which (`"published"`, `"cv"` or `"se"`). Published limits that do not contain their own estimate (it happens: a 100% with limits 85.5 to 98.6 in table 41-10-0012) are kept as published and marked `"inconsistent"`; every comparison of the family warns and leaves such a row out. A limit is matched to a measure when it fits 95% of the cells. Labels are read in English: use the English-language tables.

Every other column but the metadata (`VECTOR`, `COORDINATE`, `UOM`, ...) identifies the cell, `DGUID` among them (one place name can name several geographies), with the spaces some labels end in trimmed. The quality flag is the estimate's: an `F` or `x` estimate is `NA` (see `sctidy`), and so are its limits.

Value

A data frame, one row per cell and measure: the identifying columns, `measure`, `estimate`, `lower`, `upper`, `cisou` and `quality`; the attribute `conflvl` is the level of the limits, and `unmatched` names any limits that could not be matched to one measure.

References

Statistics Canada (2023). Self-perceived mental health of First Nations people living off reserve, Metis and Inuit by age group and gender. Table 41-10-0067-01. DOI 10.25318/4110006701-eng

See Also

`sctidy`, `indgap`, `ordcmp`.

Examples

```
# the layout of a downloaded table: percent and number of persons, with
# limits for the percent
d <- data.frame(
  REF_DATE = "2022",
  GEO = "Canada",
  `Indigenous identity` = rep(c("First Nations", "Metis"), each = 4),
  Statistics = rep(
    c(
      "Number of persons",
      "Percent",
      "Low 95% confidence interval",
      "High 95% confidence interval"
    ),
    2
  ),
  UOM = rep(c("Persons", "Percent", "Percent", "Percent"), 2),
  VALUE = c(415000, 47.1, 45.2, 49.0, 320000, 50.3, 48.1, 52.5),
  STATUS = "",
  check.names = FALSE
)
scests(d)
# a rate with its coefficient of variation instead of limits
r <- data.frame(
  GEO = "Canada",
  Characteristics = c("Rate", "Coefficient of variation for rate"),
  VALUE = c(12.5, 8),
  STATUS = ""
)
scests(r)
```

schemapath *Path of a JSON Schema shipped with the package*

Description

The capsule files are described by JSON Schema (draft 2020-12) documents in the package's `schemas` folder, the same files as the `spec` folder of the rmbl GitHub Action: `capsule` for `data_provenance.json`, `bundle` for `capsule_bundle.json` and `manifest` for a results manifest. `rmbl verify` checks each capsule file against its schema before the package's own checks.

Usage

```
schemapath(name)
```

Arguments

`name`	Character scalar: `"capsule"`, `"bundle"` or `"manifest"`, with or without the `.schema.json` ending.

Value

The path of the installed schema file, a character scalar. Error: any other name.

See Also

`blcli` (the verify verb), `vrfcps`.

Examples

```
schemapath("capsule")
s <- fromjs(paste(readLines(schemapath("bundle")), collapse = "\n"))
s$required

# only the four schemas exist
try(schemapath("nope"))
```

schval *Apply Schema Validation, Stopping on Fatal Issues*

Description

Runs `valsch()` (or, for a list of rules, `valrules()`) and acts on the result: the first fatal issue stops with an error, and every warning-severity issue is raised as an R warning. Put it at the top of an analysis script so that a re-released file with a missing column stops the run before any result is computed.

Usage

```
schval(dfraw, prov)
```

Arguments

dfraw	The data frame to validate.
prov	A provenance list as returned by loadprov() (with a schema element), or one rule or a list of rules from rul(), run with valrules() under the same stop-on-fatal contract.

Details

Issues are handled in the order the validator returns them; a fatal issue stops at once, so warnings after it are not raised. With a list of rules the same contract applies to each rule's severity.

Unlike valsch(), this function refuses to check against nothing, because a validation that checks nothing must not report success. Errors: dfraw not a data frame; prov a data frame; prov NULL or an empty list; a list that mixes rules with other elements; an unnamed list that is not all rules; something other than a list; a provenance list with no schema element (the message names the elements it has, so a typo such as scheme shows); an empty schema (list(schema = list())).

Value

Invisibly, TRUE if no issues were found and FALSE if only warning-severity issues were found (each raised as a warning). Errors if any fatal issue is present.

Examples

```
prov <- list(schema = list(expected_columns = "id"))
schval(data.frame(id = 1), prov)
# A missing required column is fatal:
try(schval(data.frame(x = 1), prov))

# a warning-level issue is raised as a warning and the result is FALSE
ranged <- list(schema = list(numeric_ranges = list(v = c(min = 0, max = 1))))
ok <- withCallingHandlers(
  schval(data.frame(v = c(0.5, 3)), ranged),
  warning = function(w) {
    message("warning: ", conditionMessage(w))
    invokeRestart("muffleWarning")
  }
)
ok

# rules work the same way
rules <- list(rul(
  "id_positive",
  function(v) v > 0,
  col = "id",
  sev = "fatal"
))
```

```
try(schval(data.frame(id = c(1, -2)), rules))

# a provenance without a schema is refused, not passed
try(schval(data.frame(id = 1), list(scheme = list())))
```

`sctidy` *Read a Statistics Canada table as published*

Description

Tidies a Statistics Canada full-table download (the "CSV for database loading" form, as the data service mirrors it under `statcan_indigenous/tbl_<pid>`): `VALUE` as a number, and the quality flags kept beside it in `quality`.

Usage

```
sctidy(x)
```

Arguments

`x`	A data frame read from a Statistics Canada table (English-language full-table CSV), or a key of the data service (`"statcan_indigenous/tbl_13100924"`, or the eight-digit table id alone). A key needs a data service key: sign in once with `dlogin` (see `dload`); a data frame you read yourself needs none.

Details

Statistics Canada flags every estimate: `E` use with caution, `F` too unreliable to be published, `x` suppressed to meet the confidentiality requirements of the Statistics Act, `..` not available, `...` not applicable. An `F`, `x`, `..` or `...` cell has no value here (`NA`), whatever the file holds, and nothing in the package imputes it or works it back from totals and margins: a suppressed cell is suppressed to protect people, and small First Nations, Metis and Inuit communities most of all. `E` values are kept and marked.

The census data tables (98-10-xxxx) come in a wide layout: one column per member of a dimension (`"Indigenous identity (9):Inuk (Inuit)[6]"`), each followed by its `Symbol` column, and no `VALUE`. They are made long first: the dimension becomes a column of member labels (`"Indigenous identity (9)"`), the values `VALUE` and the symbols `STATUS`, so every Statistics Canada table reads the same way.

A label that names two members of one dimension (the census tables repeat labels under different parents, and the file drops the hierarchy) gets its member number from the coordinate: `"Without children [3]"` and `"Without children [7]"`. Other labels are left as they are.

Value

The data frame with `VALUE` numeric and a factor column `quality` with levels `"ok"`, `"E"`, `"F"`, `"x"`, `".."`, `"..."`, of class `c("rmbl_statcan", "data.frame")`.

References

Statistics Canada. Symbol legend. statcan.gc.ca/en/concepts/definitions/guide-symbol

See Also

indgap, indtabs.

Examples

```
# the layout of a downloaded table, with one cell suppressed
d <- data.frame(REF_DATE = "2019/2022", GEO = "Canada",
                `Indigenous identity` = c("First Nations", "Inuk (Inuit)",
                                          "Non-Indigenous"),
                VALUE = c("45.8", "", "61.5"), STATUS = c("", "x", ""),
                check.names = FALSE)
sctidy(d)


# needs a data service key: sign in once with dlogin()
t924 <- sctidy("13100924")
```

---

sevidx *A severity-weighted crime index*

Description

Offence rates combined into one index, unweighted (the sum of rates) and weighted by each offence's severity (for example its mean statutory sentence), per unit and period.

Usage

```
sevidx(data, unit, time, offence, count, w, pop = NULL, per = 1e+05)
```

Arguments

data	A data frame, one row per unit, period and offence.
unit, time, offence, count	Column names.
w	Named weights, one per offence counted.
pop	Optional column name of the population; without it count is taken as a rate already.
per	Rate base (default 100000).

Details

With r_k the rate of offence k per `per` residents and w_k its weight, the indices are $\sum_k r_k$ and $\sum_k w_k r_k$. Mean statutory sentence lengths are one choice of weight (as in Acosta, Mejia and Zorro Medina 2023); sentencing guidelines are another (the Cambridge Crime Harm Index, Sherman, Neyroud and Neyroud 2016). The weights are the analyst's to state and defend: the index records them. A unit-period missing an offence that has a weight gets `NA`, never a partial sum.

Value

A data frame of class `rmbl_severity`: `unit`, `time`, `unweighted`, `weighted`; attribute `weights`.

References

Sherman L, Neyroud PW, Neyroud E (2016). The Cambridge Crime Harm Index: measuring total harm from crime based on sentencing guidelines. Policing 10(3):171-183. DOI 10.1093/police/paw003

Applied with weights from the Colombian Penal Code: Acosta, Mejia and Zorro Medina (2023), see `plndid`.

Examples

```
d <- data.frame(
  city = rep(c("A", "B"), each = 3),
  year = 2024,
  offence = rep(c("homicide", "robbery", "burglary"), 2),
  n = c(5, 120, 300, 2, 90, 410),
  pop = rep(c(250000, 180000), each = 3)
)
sevidx(
  d,
  "city",
  "year",
  "offence",
  "n",
  w = c(homicide = 25, robbery = 8, burglary = 4),
  pop = "pop"
)
```

`sha256file` *Compute a File's SHA256 Digest*

Description

Returns the SHA-256 digest of a file as a lowercase hex string. A digest is a 64-character fingerprint of the file's exact bytes: any change to the file, even one byte, gives a different digest, so recording it when data are downloaded and checking it later shows whether the file used in an analysis is the one that was recorded. The package uses it to record and verify data provenance in capsules and manifests.

Usage

```
sha256file(path)
```

Arguments

path	A single string: the path of the file to hash (~ is expanded as usual by R's file functions).

Details

The digest is SHA-256 as specified in FIPS 180-4 (National Institute of Standards and Technology, 2015), computed by the package's own compiled core; it is identical to `digest::digest(file = path, algo = "sha256")` and to the `sha256sum` command-line tool. The file is read in binary mode, 8 MB at a time, so memory use stays at one chunk whatever the file's size, and line endings or encodings are never altered. A file larger than one chunk shows a byte-progress bar on stderr after 0.75 s when a person is watching (an interactive session, or `options(morie.progress = TRUE)`); `options(morie.quiet = TRUE)` silences it. Hashing runs at roughly 100 MB a second or faster.

When this file is sourced on its own inside a capsule bundle (outside the package), the same digest comes from base R's `tools::sha256sum()` (R 4.5.0 and later) or else from the bundle's pure-R SHA-256 implementation, which reads the file 1 MB at a time (slow on a large file, but in constant memory).

Refusals: anything other than a single string naming an existing regular file – a missing path, a directory, `NA`, a number or a vector of several paths – stops with `"sha256file: <path> is not a file"`.

Value

A single string of 64 lowercase hexadecimal characters, the SHA-256 digest of the file's bytes.

References

National Institute of Standards and Technology (2015). Secure Hash Standard (SHS). FIPS PUB 180-4. DOI 10.6028/NIST.FIPS.180-4

See Also

`csha256()` for the digest of a string or raw vector held in memory.

Examples

```
f <- tempfile()
writeLines("hello capsule", f)
sha256file(f)

# Deterministic: the same bytes always yield the same digest.
identical(sha256file(f), sha256file(f))

# Any change to the file changes the digest (tamper-evidence).
before <- sha256file(f)
```

```
writeLines("hello capsule (edited)", f)
after <- sha256file(f)
before == after      # FALSE

# Provenance pin: record a digest, verify it later.
pinned <- sha256file(f)
stopifnot(sha256file(f) == pinned)

# An empty file has the well-known empty-input digest.
e <- tempfile()
file.create(e)
sha256file(e)

# The file's bytes and the same bytes in memory hash alike.
identical(sha256file(f), csha256(readBin(f, "raw", file.size(f))))

# A directory or a missing file is refused.
try(sha256file(tempdir()))
```

---

shrs *Share of a total, in percent*

Description

Expresses each count as a percentage of a total (for example each division's share of all stops), with a Wilson confidence interval. A share's denominator is the total of the same events, so shares over a complete grouping sum to 100. Use `rtcalc` instead when the denominator is a population; "40% of stops" and "40 stops per 1,000 residents" are different claims.

Usage

```
shrs(x, ...)

## S3 method for class 'data.frame'
shrs(x, count = NULL, by = NULL, total = NULL, conflvl = 0.95, ...)

## Default S3 method:
shrs(x, total = NULL, conflvl = 0.95, ...)
```

Arguments

`x`	A data frame, or a numeric vector of counts.
`...`	Passed to methods.
`count`	Column of counts (required), unquoted or as a string: non-negative whole numbers.
`by`	Character vector of grouping columns. Shares are computed over the groups, so they sum to 100 unless `total` says otherwise.

`total`	The denominator. `NULL` (default) sums `count`, which makes the shares sum to 100. A number uses that number; a column name (data-frame method) uses the sum of that column – for the case where some of the total is not in the table.
`conflvl`	Number in (0, 1), default 0.95: the confidence level of the interval.

Details

For count y out of total n, the share is $100\,y/n$ and the interval is Wilson's score interval for $\hat{p} = y/n$,

$$\frac{\hat{p} + \frac{z^2}{2n} \pm z\sqrt{\frac{\hat{p}(1-\hat{p})}{n} + \frac{z^2}{4n^2}}}{1 + z^2/n},$$

with z the standard normal quantile for `conflvl`, reported in percent and clipped to $[0, 100]$.

The interval is Wilson's, not the textbook normal approximation. The normal interval on a proportion is wrong in exactly the cases people reach for it – small counts and shares near 0 or 100, where it runs past the ends of the scale and reports a negative percentage. Wilson's stays inside the scale and is accurate at those counts. It treats each count as a binomial draw from the total, which is an approximation when the total is itself the sum of the counts.

With a data frame, counts are summed within each combination of the `by` columns (missing values ignored), groups sorted. The default method treats each element of `x` as a row; when `x` is named, its names become a leading `label` column.

Errors: a count column that is not numeric or not found; negative counts; a total of zero or below; a count larger than the total; `conflvl` outside (0, 1).

Value

A data frame of class `c("rmbl_share", "data.frame")` with the grouping columns (or `label`, for a named vector), `count`, `total`, `share`, `lower` and `upper` (percent), plus a `share` attribute: a list of `count`, `by`, `total` and `conflvl`.

See Also

`rtcalc` for events per population, `yoy` for change between periods.

Examples

```
stops <- data.frame(
  division = c("North", "South", "East"),
  stops = c(412, 77, 3))

# shares of the table's own total, summing to 100
shrs(stops, stops, by = "division")

# East is 0.6% of stops, and the interval does not run below zero
# the way a normal approximation would
shrs(stops, stops, by = "division")$lower

# a denominator from outside the table
shrs(stops, stops, by = "division", total = 10000)
```

```
# a plain vector
shrs(c(3, 7))

# a count larger than its total is not a share
try(shrs(c(5, 2), total = 4))
```

---

signpubkey *Public half of a signing key*

Description

Strips the secret seeds from an XMSS signing key, leaving only what a verifier needs. Publish this, or hand it to whoever must check your signatures; never the object returned by pqckg(), which lets its holder sign in your name. The public key does not change as the signing key is used, so it is made once.

Usage

```
signpubkey(key)
```

Arguments

key	A bricklayer_signing_key from pqckg().

Details

The public key is the Merkle root and the public seed that masks the hashes (RFC 8391's root || SEED), with the tree height and scheme name. skseed, skprf and next_index are dropped. For a standardised key from fipskg() use fipspubkey() instead: this function refuses any key that is not a bricklayer_signing_key ("key must come from pqckg()").

Value

A list of class bricklayer_public_key: root and pubseed (64 hex characters each), height and scheme.

See Also

cpsvrf(), which takes it.

Examples

```
# keep this example's
op <- options(rmoriebricklayer.xmssidxfile = tempfile())
# one-time index record out of your configuration directory; leave the
# default in use
key <- pqckg(height = 2)
pub <- signpubkey(key)

# The secret seed is gone.
is.null(pub$skseed)

# And verification works from the public half alone.
sig <- cpssign("manifest-digest", key)
cpsvrf("manifest-digest", sig, pub)

# A standardised key has its own function.
try(signpubkey(fipskg("ML-DSA-44")))
options(op)
```

`siudate` *Convert a human-readable SIU report date to ISO format*

Description

Turns the written-out dates found in SIU reports, in English or French, into ISO `YYYY-MM-DD` strings that sort and convert with `as.Date()`. `"January 5, 2023"`, `"January 5 2023"`, `"5 January 2023"` and `"5 janvier 2023"` all become `"2023-01-05"`.

Usage

```
siudate(x)
```

Arguments

`x`	A character vector of human-readable dates. A vector that is all `NA` (of any type) is accepted.

Details

Each element is parsed by the native core. Month names may be English or French (accents optional), full or abbreviated, before or after the day; ordinals such as `"1er"` are accepted, and an ISO date is passed through. A date that does not exist (`"February 30, 2023"`) or text that is not a date gives `""`. `NA` stays `NA`.

Error: `x` that is neither character nor all `NA` ("x must be a character vector of dates").

Value

A character vector the same length as `x`: `YYYY-MM-DD` strings, `""` for unparseable input and `NA` for `NA`. Unnamed.

Examples

```
siudate(c("January 5, 2023", "not a date"))

# French, day-first and impossible dates
siudate(c("5 janvier 2023", "5 January 2023", "February 30, 2023", NA))
as.Date(siudate("April 28, 2023"))

# numbers are refused
try(siudate(20230105))
```

`siufetch` *Fetch an Ontario SIU director's report by drid*

Description

Downloads the web page of one director's report of the Ontario Special Investigations Unit (SIU), the agency that investigates police interactions that end in death, serious injury or sexual-assault allegations, and saves its HTML to `dest`. Reports are identified by their `drid` number. If the live site does not answer, a copy from the Wayback Machine is used. This is the fetch step of the open SIU corpus pipeline: pair it with the SIU parser in this package (`siufprs` fetches and parses in one call; the compiled core under `src/siu_*.cpp` is the canonical implementation, and **rmorie** links to the same core) to rebuild the director's-report corpus yourself.

Usage

```
siufetch(drid, dest, lang = c("en", "fr"), wb = "", tmout = 120L)
```

Arguments

`drid`	Director's-report id: the `drid=` query parameter, one positive whole number (integer or integer-like, such as `648` or `"648"`).
`dest`	Character string: the file to write the fetched HTML to.
`lang`	`"en"` (default) or `"fr"`: the language of the report page.
`wb`	Optional explicit Wayback snapshot URL (passed through to `blfetch`); `""` (the default) lets it discover one.
`tmout`	Request timeout in seconds. Default 120.

Details

The URL fetched is

`https://www.siu.on.ca/<lang>/directors_report_details.php?drid=<drid>`

through `blfetch`, which tries the live page first and falls back to a Wayback Machine snapshot (the one given in `wb`, or one it looks up).

Network: one or more HTTPS requests to www.siu.on.ca and, on failure, web.archive.org. Side effect: writes `dest`.

Errors, before any request: `drid` not one positive whole number (`1.5` is refused rather than read as report 1); `lang` not `"en"` or `"fr"`; `dest` not a single non-empty string. Then any error from `blfetch()`, for example when neither the live page nor an archived copy can be reached.

Value

`"live"` or `"wayback"` (invisibly), as `blfetch`: which source the HTML came from.

See Also

`blfetch`, `siufprs`

Examples

```
# the report number is checked before anything is sent
try(siufetch(1.5, tempfile(fileext = ".html")))
try(siufetch(648, tempfile(fileext = ".html"), lang = "de"))

# a real fetch needs the network (the SIU site or the Wayback Machine)


dest <- tempfile(fileext = ".html")
siufetch(648, dest)
siufetch(648, tempfile(fileext = ".html"), lang = "fr")
```

---

siufprs *Fetch and parse one SIU director's report*

Description

In one call, downloads one director's report from the SIU web site with `siufetch()` and parses it with `siuprs()`. It is meant for loops over many report ids: a report that cannot be retrieved gives `NULL` and a message instead of stopping the loop.

Usage

```
siufprs(drid, lang = c("en", "fr"))
```

Arguments

drid	The director's report id: one positive whole number (or a string of one), the drid= query parameter of the report URL on the SIU site.
lang	Report language: "en" (default, the English page) or "fr" (the French page). Partial matching is allowed.

Details

The arguments are checked before any network use, so a bad id or language fails at once. The report is downloaded to a temporary file, which is deleted on exit. Any error from the download (network failure, an HTTP error, no report at that id) is caught and turned into a message of the form "bricklayer: could not fetch SIU report <drid>: <reason>" and a NULL return. Errors from parsing are not caught.

Errors: drid that is not one positive whole number; lang other than "en" or "fr".

Value

A named character vector of parsed fields, as from siuprs(), or NULL when the fetch fails.

Examples

```
# argument errors come before any network use
try(siufprs("abc"))
try(siufprs(648, lang = "de"))


# needs the network: downloads report 648 from the SIU site
f <- siufprs(648)
if (!is.null(f)) f[["police_service"]]
fr <- siufprs(648, lang = "fr")
```

---

siuprs *Parse an SIU director's report into the schema fields*

Description

Extracts every siusch() field from the HTML of an Ontario SIU director's report: the police service, the incident, notification and decision dates, the investigator and witness counts, the affected person's age and sex, the injuries, the legislation considered and the director's name. The extraction is rule-based, deterministic and offline, so the same report always gives the same answer; use it to turn a folder of saved reports into a data set.

Usage

```
siuprs(html)
```

Arguments

`html`	A character vector. Each element is either the raw HTML of one report or the path to a saved report file (for example one written by `siufetch()`). With more than one element, each is parsed and the results are bound into a data frame.

Details

The text is first converted as in `siutxt()`, then the report language is detected from section headings (English or French; the SIU publishes both) and each field is read by its own rule set in the native core. Dates are converted with the same rules as `siudate()`. The subject-official count uses the rules of `siuso()`. Fields the report does not state come back as `""`; a page that is not a report gives all fields `""` and `_language` `"unknown"`.

A single string with no < that is shorter than 1000 bytes and names an existing file is read as a path. A string that looks like a path (ends in `.htm`/`.html` or contains a slash) but names no file is an error, not an empty report.

Errors: `html` not character, empty or `NA`; a missing file ("no such file"); input over 2 MiB.

Value

For one input, a named character vector of 17 elements: the 16 schema fields plus `_language` (`"en"`, `"fr"` or `"unknown"`), in alphabetical order of name. Every value is a string, counts included (`"2"`); convert with `as.integer()` as needed. For several inputs, a data.frame with one row per input and one character column per field (`NA` where a field is absent from a row).

Limits

The input is capped at 2 MiB (a report page is a few hundred KB). Before the text is read, a line longer than 2000 characters is split at the sentence end nearest the cap (else at a space), and a warning says how many lines were split: a field that spans a split may be incomplete.

Examples

```
f <- siuprs(system.file(
  "extdata",
  "siu_synthetic_report.html",
  package = "rmoriebricklayer"
))
f[["number_of_subject_officials"]]
f[c(
  "police_service",
  "date_of_incident_iso",
  "date_of_director_decision_iso"
)]

# several reports at once give a data frame, one row each
path <- system.file(
  "extdata",
  "siu_synthetic_report.html",
  package = "rmoriebricklayer"
)
```

```
html <- paste(readLines(path), collapse = "\n")
df <- siuprs(c(html, "<p>not a report</p>"))
df[, c("_language", "police_service", "number_of_witness_officials")]

# a mistyped path is an error, not an empty report
try(siuprs("no-such-report.html"))
```

`siusch` *The panel-reviewed SIU report field schema*

Description

Returns the sixteen fields that the package extracts from every Ontario Special Investigations Unit (SIU) director's report, with a one-line description of each. Use it to see what `siuprs()` returns, to build an empty coding sheet for hand review, or to check which fields are counts before summing them across reports.

Usage

```
siusch()
```

Details

The schema is held in the native core (src/siu_schema.h) and is the same list that `siuprs()` fills. Three fields are dates in ISO `YYYY-MM-DD` form, five are counts and the rest are free text. Count-type fields (`count = TRUE`) count distinct entities, and zero is a real answer: a witness-official-only investigation has zero subject officials. The function takes no arguments, reads no files and uses no network.

Value

A data.frame with 16 rows and three columns: `name` (character, the field name used by `siuprs()`), `count` (logical, `TRUE` for the five count fields) and `description` (character, what the field records).

Examples

```
s <- siusch()
s$name

# only the count fields
s[s$count, c("name", "description")]

# an empty hand-coding sheet with one column per field
sheet <- as.data.frame(setNames(
  replicate(nrow(s), character(0), simplify = FALSE),
  s$name
))
ncol(sheet)
```

`siuso` *Resolve the subject-official count from SIU report text*

Description

Counts the subject officials (SO), the officers whose conduct the SIU investigated, from the plain text of one report, and says which rule gave the answer. Use it for reports where a model panel or a human coder is unsure, so that the number is reproducible and its evidence can be checked.

Usage

```
siuso(text)
```

Arguments

`text`	A length-1 character vector of plain report text (see `siutxt()`), not HTML.

Details

The standard SIU privacy paragraph and the witness-officer glossary note are stripped first (both mention "subject officer" without naming one). The rules then run most-specific first and the first that applies wins:

1. the roster under the "Subject Officials"/"Subject Officers" heading: the larger of the highest `SO #N` ordinal and the number of entries;
2. the highest `SO #N` ordinal in the whole document, trusted only when `SO #1` is also present;
3. a spelled-out or numeric plural ("two subject officials");
4. a singular subject official present (1);
5. an explicit statement that there was no subject official, or witness officials listed and no subject official anywhere (0, a real answer);
6. otherwise unresolved (`NA`).

French reports use the same rules in their vocabulary (the subject official is tagged "AI no 1", the witness official "AT no 1").

This function is the pure rule set. For reports that already have a panel-reviewed count, prefer that verified value; use these rules for unreviewed reports.

Error: `text` that is not a single non-NA string.

Value

A list with two elements: `count` (integer, `NA` when unresolved) and `reason` (character, the rule that fired and its evidence, for example `"plural cue 'three subject officials'"`).

Examples

```
siuso(
  "Subject Officials\nSO #1 Interviewed\nSO #2 Declined interview")

# witness officials only: zero is the answer
siuso("Witness Officials\nWO #1 Interviewed\nWO #2 Interviewed")

# a spelled-out plural, and an unresolved text
siuso("Three subject officials were designated.")$count
siuso("No officers are named here.")

# HTML or several strings are not text
try(siuso(c("a", "b")))
```

---

siutxt *Convert SIU report HTML to plain text*

Description

Strips the markup from one SIU director's report page and returns the readable text, keeping line breaks so section headings ("Subject Officials", "Witness Officials") stay on lines of their own. Use it before `siuso()`, or to read or search a saved report without a browser.

Usage

```
siutxt(html)
```

Arguments

`html`	A length-1 character vector holding the raw HTML of one report page. A file path is not read here (use `siuprs()`, which accepts a path).

Details

The conversion runs in the native core: tags are removed, block-level tags become line breaks, HTML entities are decoded and runs of spaces are collapsed while newlines are kept. Nothing is fetched and nothing is written.

Errors: `html` must be a single non-NA character string ("html must be a single non-NA character string"), and input over 2 MiB is refused ("html is larger than 2 MiB: not a report page").

Value

A length-1 character vector of plain text.

Limits

The input is capped at 2 MiB (a report page is a few hundred KB). Before the text is read, a line longer than 2000 characters is split at the sentence end nearest the cap (else at a space), and a warning says how many lines were split: a field that spans a split may be incomplete.

Examples

```
siutxt("<p>Number of SIU Investigators assigned: 3</p>")

# the bundled synthetic report, as text
f <- system.file(
  "extdata",
  "siu_synthetic_report.html",
  package = "rmoriebricklayer"
)
txt <- siutxt(paste(readLines(f), collapse = "\n"))
cat(substr(txt, 1, 200))

# more than one string is refused
try(siutxt(c("<p>a</p>", "<p>b</p>")))
```

---

sprd *Variance and standard deviation with the right correction, and why*

Description

The variance, standard deviation and standard error of the mean of a variable, with the denominator its source calls for: Bessel's $n - 1$ when the values are a sample from a larger population, n when they are the whole population and it is being described, and the finite-population correction when they are a sample drawn without replacement from a known, finite population. `design = "auto"` chooses from `N`, and the result says which rule it applied and why.

Usage

```
sprd(
  x,
  des = c("auto", "sample", "census", "finite"),
  N = NULL,
  w = NULL,
  wtype = c("frequency", "reliability"),
  narm = FALSE
)
```

Arguments

x	A numeric vector.
des	"auto" (default), "sample", "census" or "finite".
N	The population size N, for "finite" and "auto".
w	Optional non-negative weights, one per value.
wtype	"frequency" (default) or "reliability".
narm	Drop missing values (default FALSE: they are an error).

Details

With n values x_i and mean $\bar{x}$:

"sample" values drawn independently from a large population (or from a process). $s^2 = \sum(x_i - \bar{x})^2/(n-1)$ is unbiased for the population variance: the $n-1$ (the Bessel correction) makes up for measuring spread around the mean of the sample itself, which is closer to the values than the population mean is. The standard error of the mean is $s/\sqrt{n}$.

"census" every unit of the population is observed (a full count, complete administrative records) and the aim is to describe that population: $\sigma^2 = \sum(x_i - \bar{x})^2/n$, with no correction, because $\bar{x}$ is the population mean itself; the mean has no sampling error. If the population is instead treated as one realisation of a process (the offences of one year as a draw of what the conditions of the city produce), use "sample".

"finite" n units drawn without replacement from N: s^2 (with $n-1$) is unbiased for the population variance defined with $N-1$, and the standard error of the mean is $\sqrt{(1-n/N)s^2/n}$; the factor $1-n/N$ is the finite-population correction, negligible below a sampling fraction of about 5 percent.

"auto" picks "sample" without N, "census" when $n = N$, and "finite" when $n < N$.

With weights: frequency weights (each value stands for w_i identical observations) use $\sum w - 1$; reliability or sampling weights (each value counts w_i times as much) use $\sum w - \sum w^2/\sum w$, which is $n-1$ when the weights are equal, and the standard error of the weighted mean is the sandwich $\sqrt{\sum w_i^2(x_i - \bar{x}_w)^2}/\sum w_i$. Survey weights from a complex design need the design itself for a standard error (svydes).

Value

A list of class rmbl_spread with n, mean, variance, sd, semean, design (the rule applied), denominator, and reason (a sentence). The print method shows them.

References

Cochran WG (1977). Sampling Techniques, 3rd edition. Wiley, New York, sections 2.4 to 2.6.

See Also

svydes for weighted surveys with strata and clusters.

Examples

```
x <- c(12, 15, 9, 20, 14)
sprd(x)                                  # a sample: n - 1
sprd(x, des = "census")                # the whole population: n
sprd(x, N = 40)             # 5 of 40 drawn: finite-population correction
sprd(x, N = 5)              # n = N: a census
sprd(c(1, 2, 3), w = c(10, 5, 1))   # frequency weights
```

`srchfig` *The figures of The Theory of Search*

Description

Draws any figure of Frost (1996), "The Theory of Search: A Simplified Explanation", from the package's search theory functions with the book's parameters, and returns the numbers it drew.

Usage

```
srchfig(fig = NULL, num = NULL, seed = 1, main = NULL)
```

Arguments

`fig`	A figure number as text, `"5-7"`, or the chapter as a number with `num`; `NULL` (the default) lists the figures.
`num`	The figure's number within chapter `fig`, or `NULL`.
`seed`	Number: the seed of figure 6-6's random efforts. Default 1; the session's random number stream is left as it was.
`main`	The title, or `NULL` for the book's caption.

Details

`srchfig()` with no argument lists the book's 36 figures (3-1 to 3-14, 4-1 to 4-7, 5-1 to 5-9 and 6-1 to 6-6) with how each is drawn: `"model"` figures are computed from the functions with the book's parameters and reproduce the values the text states (for example 57.91, 9.08 and 1.42 percent in figure 3-4, 0.75 and 0.6836 in figure 5-7, 0.7899 at coverage 1 in figure 5-8, the optimal factor 1.02 at an effort of 4 E^2 in figure 6-3); `"schematic"` figures are diagrams with no data (3-6, 4-3, 4-6, 5-2); `"approximation"` figures are drawn from the model where the book's plate cannot be recovered exactly: 3-7 (the density implied by boxing in two circles is drawn as the `"line"` datum, a circular normal spread along the segment), 3-13 (the book's surface is illustrative; it is drawn from the cells of figure 3-14), 5-9 (the plate is blank in the published file; it is drawn from the text, an inverse cube curve with a smaller sweep width and flattened by navigation error), 6-6 (the book's random efforts are not given; 250 groups of 5 are drawn here with `seed`).

Distances are in standard deviations of position error (`E = 1.1774`), or in sweep widths (`W = 1`) for the lateral range figures. The general methods draw the same figures with other parameters: `plot(ltrlrng(...), equiv = , spacing = )`, `plot(srchpod(...), models = , m = )`, `plot(datumpmap(...), view = )`.

Value

Without `fig`, a data frame of `fig`, `caption`, `how` and `yours`: one line of R that draws the figure from your own numbers (the function behind it, with example inputs to replace; for a schematic, the computation it pictures). With one, the figure's data, invisibly: a data frame (or a list of them) of what was drawn.

References

Frost, J. R. (1996). The Theory of Search: A Simplified Explanation. Soza and Company and U.S. Coast Guard Office of Search and Rescue. Chapters 3-6.

Examples

```
head(srchfig())
srchfig("3-4")
srchfig(5, 8)
d <- srchfig("6-3")
head(d)
try(srchfig("9-9"))
```

---

staysmry *Length of stay with its distribution and interval*

Description

Where `alos()` takes the total days and returns a mean, this takes one value per person and reports the spread as well, because a mean stay is a poor summary of a distribution that is usually skewed.

Usage

```
staysmry(daysperper, conflvl = 0.95)
```

Arguments

`daysperper`	Numeric vector of days served, one element per person; non-negative, no `NA` or `Inf`, at least one value.
`conflvl`	Numeric scalar strictly between 0 and 1, the confidence level for the interval on the mean. Default `0.95`.

Details

The interval is the ordinary t interval on a mean, $\bar{x} \pm t_{n-1,1-\alpha/2}\, s/\sqrt{n}$. It describes uncertainty about the AVERAGE stay, not the spread of stays, and on a skewed distribution the median and the interquartile range say more about a typical stay than the mean does – which is why they are returned beside it rather than left to be asked for.

Lakner's caveat on alos() applies here too (1976, p.16-17): a person still held has an unfinished stay, so a window shorter than the longest stay biases the mean DOWNWARD.

Errors when daysperper is not numeric, is empty, or has NA, Inf or negative values, and when conflvl is not strictly inside (0, 1).

Value

A one-row data frame of class c("rmbl_stay_summary", "data.frame") (its print method shows it as a box) with numeric columns n (people), totaldays, mean, sd (sample standard deviation), median, iqr (interquartile range, quantile type 7), max, se (standard error of the mean), lower and upper (the t interval on the mean). With a single person sd, se, lower and upper are NA.

See Also

alos(), adp(), stkflow()

Examples

```
# a skewed distribution: most stays short, a few long
stays <- c(rep(1, 40), rep(3, 30), rep(10, 20), 60, 90, 120)
staysmry(stays)

# the mean is pulled well above the median by the long tail
staysmry(stays)[, c("mean", "median", "max")]

# a 90% interval instead of 95%
staysmry(stays, conflvl = 0.90)[, c("mean", "lower", "upper")]

# a confidence level of 1 is refused
try(staysmry(stays, conflvl = 1))
```

---

stdinc *Standardised incidence ratio, with an exact interval*

Description

Divides each area's observed count by its expected count to give the standardised incidence ratio (SIR), and adds an exact Poisson confidence interval. A ratio of one is the overall experience; above one is an excess and below one a deficit. Use it after expcnts to say which areas differ from the overall rate by more than chance.

Usage

```
stdinc(obs, expd, area = NULL, conflvl = 0.95)
```

Arguments

obs	Numeric vector of observed counts (non-negative).
expd	Numeric vector of expected counts (positive), for example from expcnts; the same length as obs, or length one.
area	Optional vector of area labels, added as the first column. Default NULL.
conflvl	Number in (0, 1), default 0.95: the two-sided confidence level of the interval.

Details

With observed count O and expected E, $SIR = O/E$. The interval for $1 - \alpha$ coverage is

$$\left[\frac{G^{-1}(\alpha/2;\, O)}{E},\ \frac{G^{-1}(1-\alpha/2;\, O+1)}{E}\right]$$

where $G^{-1}(p; k)$ is the gamma quantile with shape k; the lower limit is 0 when $O = 0$. This is the exact Poisson interval, from the relation between the Poisson and gamma distributions, and so is identical to stats::poisson.test()'s. It is the interval to use here because the counts that matter are small: a normal approximation on an observed count of three is not an interval, it is a decoration.

Each area's interval is a separate test; with many areas some will exclude one by chance (see scanadj). A row with a missing observed or expected count is dropped whole (its area label with it), after the lengths are compared.

Errors: negative, infinite or non-numeric counts; an expected count of zero or below; lengths that differ (a single expd is recycled). A conflvl outside (0, 1) is an error.

Value

A data frame with one row per area: area (when given), observed, expected, sir, lower, upper (numeric), and logical excess (the interval lies above one), deficit (the interval lies below one) and significant (either).

References

Lawson, A. B. *Using R for Bayesian Spatial and Spatio-Temporal Health Modeling*. Chapman and Hall/CRC, Chapter 1.

See Also

expcnts, ebrates, fnllim

Examples

```
stdinc(obs = c(30, 12, 3), expd = c(20, 14, 5),
    area = c("North", "South", "East"))

# An observed count of three carries almost no information, and the
# interval says so rather than hiding it.
stdinc(3, 5)
```

```
# a 90 percent interval, and a zero count (lower limit exactly 0)
stdinc(c(0, 40), c(2, 25), conflvl = 0.90)

try(stdinc(c(3, 4), c(5, 0)))
```

`stdprev` *Age-standardise survey percentages*

Description

The percentage (or proportion, or rate) a group would have with the age structure of a standard population, from the estimate and published limits in each age group: for comparing First Nations, Metis and Inuit prevalences (suicidal thoughts, mood and anxiety disorders, heavy drinking) with each other and with an older population, for example, when the age pattern of the outcome is steep.

Usage

```
stdprev(
  est,
  lower,
  upper,
  std,
  grp = NULL,
  qual = NULL,
  conflvl = 0.95,
  scale = 100
)
```

Arguments

`est`, `lower`, `upper`	Numeric vectors (not column names): the estimate in each age group (and group) and its published limits.
`std`	The standard population, one count (or share) per age group.
`grp`	Optional labels, one per row: one result per group. Rows are matched to `std` by position, so order each group's rows by age in the order of `std` (sort by an age order, not alphabetically).
`qual`	Optional quality flags, one per row (a vector; see `qltyrcd`).
`conflvl`	Confidence level of the prevalence intervals (default 0.95).
`scale`	The upper bound of the scale: 100 for percentages (the default), 1 for proportions, `Inf` for rates.

Details

With estimates p_i, standard errors $s_i = (u_i - l_i)/(2z)$ and standard weights w_i (the standard's shares), the standardised estimate is $\sum w_i p_i$ with standard error $\sqrt{\sum w_i^2 s_i^2}$, the age groups being disjoint domains of the survey (treated as independent: see `indgap`). The interval is normal, cut at 0 and at `scale`. A group with a suppressed or missing age group, or one whose published interval misses its own estimate, has no standardised value: the age group is not imputed. The standard should be the age structure of the population compared, or a common one: for First Nations, Metis and Inuit comparisons, the Indigenous population's own (`cnsstd`; Kumar and Tjepkema 2019).

Value

A data frame of class `rmbl_prevalence`: `group`, `estimate`, `lower`, `upper`, `se`, `usewithcau`; attribute `conflvl`. `plot()` draws it.

References

Klein RJ, Schoenborn CA (2001). Age adjustment using the 2000 projected U.S. population. Healthy People 2010 Statistical Notes 20. National Center for Health Statistics.

Kumar MB, Tjepkema M (2019). Suicide among First Nations people, Metis and Inuit (2011-2016): findings from the 2011 Canadian Census Health and Environment Cohort (CanCHEC). National Household Survey: Aboriginal Peoples, Statistics Canada catalogue no. 99-011-X2019001.

See Also

`agestd` for counts of events, `cnsstd`, `indgap`.

Examples

```
# seriously contemplated suicide in the past 12 months, 2017 (table
# 41-10-0021-01), standardised to the Indigenous population's age structure
# (2021 Census)
ages <- c(
  "15 to 24 years",
  "25 to 34 years",
  "35 to 44 years",
  "45 to 54 years",
  "55 years and over"
)
si <- indiex("suicide")
si <- si[
  si$measure == "Seriously contemplated suicide in the past 12 months" &
    si$age %in% ages,
]
si <- si[order(si$group, match(si$age, ages)), ]
std <- cnsstd(cnsprof(indiex("censuspop")))
w <- c(
  sum(std$population[4:5]),
  std$population[6:8],
  sum(std$population[9:11])
)
```

```
sp <- stdprev(
  si$estimate,
  si$lower,
  si$upper,
  std = w,
  grp = si$group,
  qual = si$quality
)
sp
plot(sp)

# your own numbers: made-up percentages in three age groups for two groups,
# standardised to a population of 500, 300 and 200 in those age groups
stdprev(
  est = c(12, 9, 6, 8, 6, 4),
  lower = c(9, 6, 3, 7, 5, 3),
  upper = c(15, 12, 9, 9, 7, 5),
  std = c(500, 300, 200),
  grp = rep(c("Nation A", "Everyone else"), each = 3)
)
```

`stepchg` *A single step change, with the scan's own null distribution*

Description

Asks whether a series shifted to a new level at some point (for example after a policy change) and, if so, where. It scans every admissible split of the series, reports the one with the largest difference between the before and after means, and gives it a p-value from the permutation distribution of the MAXIMUM over splits – not from the best split's own test, which is the standard way to find a change point in noise.

Usage

```
stepchg(y, x = NULL, minseg = 2L, nperm = 9999L, seed = 1L)
```

Arguments

`y`	Numeric vector, the series in period order.
`x`	Numeric or date periods, default `NULL` (the position), strictly increasing and not missing where `y` has a value. Used only to label the break.
`minseg`	Whole number, default 2: the fewest periods allowed on either side of the break.
`nperm`	Whole number, default 9999: the number of random permutations for the null distribution. The exact enumeration is used instead when $n!$ is at most `nperm`.
`seed`	Integer, default 1: the seed for the permutations, so the p-value is reproducible.

Details

For a split after period k, with $n_a = k$ and $n_b = n - k$ periods and means $\bar{y}_a$, $\bar{y}_b$, the statistic is the between-segment sum of squares

$$T_k = \frac{n_a n_b}{n}(\bar{y}_a - \bar{y}_b)^2,$$

over $k = m, \ldots, n - m$ with $m =$ `minseg`. The observed $\max_k T_k$ is compared with the same maximum over reorderings of the series, and $p = (1 + \text{count}(T^*_{\max} \geq T_{\max}))/(1 + B)$ over B random reorderings. When $n! \leq$ `nperm` (with the default, $n \leq 7$) every ordering is enumerated, the observed one among them, and the p-value is exact: $\text{count}(T^*_{\max} \geq T_{\max})/n!$; otherwise `nperm` random permutations are drawn under `seed`, and the caller's random number stream is restored afterwards.

The test assumes the periods are exchangeable under no change, so it ignores autocorrelation and smooth trends; a steady trend can produce a `"step"`. With very few periods even a clean step cannot reach small p-values (six periods allow only 720 orderings). Missing values are dropped. A series with values beyond 1e150 is scanned in units of its largest absolute value, so the statistic does not overflow; `before`, `after` and `difference` are given back in the units of `y`, and `statistic` stays in the rescaled units. Errors: infinite values; fewer than `2 * minseg` periods; numeric or date `x` that is not strictly increasing. A progress bar is shown in an interactive session.

Value

A list of class `rmbl_step_change` with `breakafter` (the period the series changes after, from `x`), `index` (its position), `before` and `after` (the segment means), `difference` (after minus before), `statistic` ($T_{\max}$), `p_value`, `nperm` (orderings or permutations used) and `method`.

References

The permutation distribution of the maximum over splits, rather than the chosen split's own test, is what makes this a test of whether there is a break rather than a way of locating the largest wobble. See any treatment of the change-point problem, e.g. Coles, S. *An Introduction to Statistical Modeling of Extreme Values* (Springer), which discusses change-point detection alongside the threshold choices that raise the same multiple-comparison issue.

See Also

`trndtst`, `cnttrnd`.

Examples

```
# A step down after the third period. With six periods there are only
# 720 orderings, so even a clean step has p near 0.1.
stepchg(c(100, 104, 98, 60, 63, 58), x = 2018:2023)

# A longer series with the same step is clearly significant.
stepchg(c(100, 104, 98, 101, 97, 60, 63, 58, 61, 59), nperm = 999)$p_value

# Pure noise: the best split is still found, and is not significant.
set.seed(2)
```

```
stepchg(stats::rnorm(12), nperm = 999)$p_value

try(stepchg(c(1, 2, 3)))
```

`stgatt` *Effects of a policy adopted at different times*

Description

Group-time average treatment effects for staggered adoption, with no unit used as a control after it is treated (Callaway and Sant'Anna 2021), or the extended two-way fixed-effects regression with a coefficient for every cohort and period (Wooldridge 2021); aggregated to one effect and to effects by time since adoption.

Usage

```
stgatt(
  data,
  unit,
  time,
  y,
  ftreat,
  ctrl = c("notyet", "never"),
  meth = c("csa", "wooldridge"),
  cl = NULL,
  reps = 499L,
  seed = NULL,
  conflvl = 0.95,
  covs = NULL,
  estmeth = c("dr", "ipw", "reg")
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`ftreat`	Column name of the period a unit is first treated (`NA`, 0 or `Inf` for a unit never treated). Treatment is taken as staying on.
`ctrl`	`"notyet"` (default) or `"never"`.
`meth`	`"csa"` (default) or `"wooldridge"`.
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one

	column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.
`reps`	Bootstrap replications (default 499; 0 for none).
`seed`	Optional seed for the bootstrap.
`conflvl`	Confidence level of the intervals reported (default 0.95).
`covs`	Optional column names of covariates for conditional parallel trends (`meth = "csa"` only).
`estmeth`	With covariates: `"dr"` (doubly robust, default), `"ipw"` or `"reg"`.

Details

Let the cohort g be the units first treated in period g, and C the comparison units: never treated (`control = "never"`), or not yet treated by period $\max(t, g-1)$ (`"notyet"`, the default). For `meth = "csa"`,

$$ATT(g,t) = \left(\bar{Y}_{g,t} - \bar{Y}_{g,b}\right) - \left(\bar{Y}_{C,t} - \bar{Y}_{C,b}\right),$$

with base period $b = g-1$ for $t \geq g$, and $b = t-1$ before (the `"varying"` base of the `did` package), so a pre-period ATT is a placebo of one period. It needs a balanced panel.

With `covs`, parallel trends need only hold conditionally on them, and each $ATT(g,t)$ is the two-period estimator of Sant'Anna and Zhao (2020) on the cohort and its comparison units, with the covariates taken at the base period b (use covariates fixed before treatment: one the policy moves is a bad control). With $\Delta Y = Y_t - Y_b$, D the cohort indicator, $\hat{m}(X)$ the least-squares fit of ΔY on X among the comparison units and $\hat{p}(X)$ the logistic propensity of belonging to the cohort,

$$\widehat{ATT}_{dr} = \frac{\sum D(\Delta Y - \hat{m})}{\sum D} - \frac{\sum w(\Delta Y - \hat{m})}{\sum w}, \quad w = \frac{\hat{p}(1-D)}{1-\hat{p}};$$

`estmeth = "ipw"` drops $\hat{m}$ (normalised inverse probability weighting) and `"reg"` drops the weights (outcome regression). The doubly robust form is consistent when either the propensity or the outcome model is right. Comparison units with $\hat{p} \geq 0.995$ are trimmed, as the `DRDID` package does. These agree with `did::attgt()` to rounding error.

For `meth = "wooldridge"`, the regression of the outcome on unit and period effects and an indicator for every treated cohort-period cell, with never-treated (or last-treated) units as the comparison, gives each $ATT(g,t)$ as a coefficient; without covariates it equals the imputation estimator.

The overall effect averages the post-adoption $ATT(g,t)$ weighted by cohort size; the event-time effect $\theta(e)$ averages $ATT(g, g+e)$ over the cohorts observed at e, weighted by cohort size. Standard errors come from the cluster bootstrap: units (or clusters) are resampled with replacement `reps` times and everything recomputed; intervals are normal with the bootstrap standard error.

Value

A list of class `rmbl_staggered_att`: `attgt` (a data frame: `cohort`, `period`, `evtm`, `estimate`, `se`, `lower`, `upper`, `ncoh`), `overall` (one row: `estimate`, `se`, `lower`, `upper`), `dynamic` (by `evtm`), `method`, `control`, `conflvl`. `plot()` draws the dynamic effects.

References

Callaway B, Sant'Anna PHC (2021). Difference-in-differences with multiple time periods. Journal of Econometrics 225(2):200-230. DOI 10.1016/j.jeconom.2020.12.001

Sant'Anna PHC, Zhao J (2020). Doubly robust difference-in-differences estimators. Journal of Econometrics 219(1):101-122. DOI 10.1016/j.jeconom.2020.06.003

Wooldridge JM (2021). Two-way fixed effects, the two-way Mundlak regression, and difference-in-differences estimators. DOI 10.2139/ssrn.3906345

Applied to a reform rolled out in four stages across judicial districts: Acosta, Mejia and Zorro Medina (2023), see plndid.

See Also

bacondcmp, evstudy.

Examples

```
p <- polex()
s <- stgatt(p, "district", "month", "crmrate", "refmonth", reps = 99,
                    seed = 1)
s$overall
head(s$dynamic)
plot(s)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# each adoption cohort against the towns not yet treated
stgatt(own, "town", "year", "rate", "first", reps = 49, seed = 1)$overall
```

---

stkflow *Stock and flow side by side, with the decomposition*

Description

Reports the same person-days as a stock and as a flow, and says how much of any change in the stock came from the number of people and how much from how long they stayed. Use it whenever a report compares periods: the count of people (a flow) and the population held (a stock) can move in opposite directions when the length of stay changes, and a table with only one of them implies the wrong direction for the other.

Usage

```
stkflow(
  days,
  people,
  prd = NULL,
  t = 365,
  expo = NULL,
  per = 1e+05,
  base = c("first", "previous")
)
```

Arguments

`days`	Numeric vector of person-days, one element per period; non-negative, no `NA` or `Inf`.
`people`	Numeric vector of the number of people (admissions or persons served), one element per period, the same length as `days`; positive.
`prd`	Optional labels for the periods (any vector, converted to character), one per period. Default `NULL` labels them `1, 2, ...`.
`t`	Numeric, the length of each period in days: one value for all periods, or one per period; positive. Default `365`. `perdays()` turns dates into this.
`expo`	Optional numeric population to express rates against, one value or one per period, positive; for example provincial residents. Default `NULL` (no rate columns).
`per`	Numeric scalar, the rate base when `expo` is given: rates are per `per` residents. Default `100000`.
`base`	Character scalar, what the change columns compare against: `"first"` (the default) measures every period against the first, which is what a report on a whole window wants; `"previous"` measures each period against the one before it, which is what a series wants (the first row's changes are then `NA`).

Details

The decomposition is exact, because days are people times length of stay: a change in days is $(1 + p)(1 + l) - 1$ for proportional changes p in people and l in stay. The two rates can therefore carry OPPOSITE signs, and the point of putting them in one table is that neither can then be quoted alone.

Changes are percentages: $100(v_i/v_b - 1)$ with b the baseline row. Errors when `days` and `people` differ in length, when `expo` has a length other than 1 or the number of periods, when `t` has a length

other than 1 or the number of periods, when prd has a length other than the number of periods, when per has more than one value, when any input has NA, Inf, text, or a value that is negative (days) or not positive (the others), and when base is not one of the two choices.

Value

A data frame of class c("rmbl_stock_flow", "data.frame"), one row per period, with columns period (character), people, days, alos (days per person), adp (days divided by t); when exposure is given also exposure, flowrate (people per per residents) and stkrate (adp per per residents). Then the percentage changes against the baseline: peochg, aloschg, dayschg, adpchg, and with exposure also flowratechg and stkratechg. Attribute "stkflow" is a list with t (the period lengths), baseline and per (NA without exposure). The print method shows the levels and a footer with the change from the baseline to the last period, flagging rates with opposite signs.

References

Lakner, E. (1976) *A Manual of Statistical Sampling Methods for Corrections Planners*. University of Illinois at Urbana-Champaign.

See Also

adp(), alos()

Examples

```
# Fewer people, held longer: the flow falls while the stock rises.
stkflow(
  days = c(115674, 126121), people = c(12647, 9608),
  prd = c("2023", "2025"),
  expo = c(15495050, 16256538)
)

# a series of three years against the previous year, with leap-year lengths
sf <- stkflow(
  days = c(100200, 104000, 99000), people = c(9800, 9500, 9900),
  prd = 2023:2025, t = c(365, 366, 365), base = "previous"
)
as.data.frame(sf)[, c("period", "alos", "adp", "peochg", "adpchg")]

# days and people must line up period by period
try(stkflow(days = c(100, 120), people = c(10, 12, 14)))
```

---

stpdnpv *Westfall-Young stepdown p-values*

Description

Adjusted p-values for several hypotheses tested together, from the joint null distribution of their statistics (permutations or bootstrap draws), holding the family-wise error rate while using the correlation between the tests.

Usage

```
stpdnpv(obs, null)
```

Arguments

`obs`	The observed statistics, one per hypothesis.
`null`	A matrix of statistics under the null: one row per draw, one column per hypothesis.

Details

The statistics are ordered by size, $|t_{(1)}| \geq \cdots \geq |t_{(K)}|$. For each null draw b, $q_{b,j} = \max_{r \geq j} |t^*_{b,(r)}|$. The adjusted p-value of the jth is the share of draws with $q_{b,j} \geq |t_{(j)}|$, made monotone: $\tilde{p}_{(j)} = \max(\tilde{p}_{(j-1)}, p_{(j)})$ (Westfall and Young 1993, algorithm 2.8).

Value

The adjusted p-values, in the order of `observed`.

References

Westfall PH, Young SS (1993). Resampling-Based Multiple Testing. Wiley, New York.

See Also

`perminf`.

Examples

```
set.seed(1)
null <- matrix(rnorm(3000), ncol = 3)
stpdnpv(c(2.5, 0.4, -2.1), null)
```

svcs *The hosted MORIE services, as the project site currently describes them*

Description

Reads `https://rmorie.com/.well-known/morie-services.json`, a signed document naming the curated-data service and its sign-in: their endpoints, whether each is `"key"` (reachable with a key requested on the site) or `"off"`, and a notice. The signature (ML-DSA-44, public key pinned in this package) is checked before anything in it is believed, so the endpoints can move or be switched off on the site without a package release, and a document that does not verify is ignored. The result is cached for a day; without a network the cached copy is used, then the copy bundled with the package.

Usage

```
svcs(refresh = FALSE, maxage = 86400, tmout = 20, offline = FALSE)
```

Arguments

`refresh`	Logical scalar, default `FALSE`: `TRUE` fetches the live document even when the cached copy is fresh.
`maxage`	Numeric scalar, the seconds a cached copy is used without asking the site; default `86400` (one day); `Inf` keeps a cached copy forever.
`tmout`	Numeric scalar, the seconds allowed for each of the two fetches (document and signature); default `20`.
`offline`	Logical scalar, default `FALSE`: `TRUE` never fetches: the session's copy, the cached copy, then the bundled one (what the examples and an offline session get).

Details

Every other MORIE package reads the same document through this function, so one sign-in and one document serve rmorie, rmoriedata and morie alike. Language models are never hosted: you bring your own key or model (`llmask`). The `llm` block of documents signed before that change carries the sign-in address only.

The environment variable `MORIE_SERVICES_URL` points the fetch at a mirror of the document (its signature is fetched from the same place, with the suffix `.sig`); the signature check is unchanged, so a mirror can only serve a document the project signed.

The order of resolution: a cached copy younger than `maxage` (unless `refresh`); the live document, fetched through the package's own checked transport (public https only, at most 1 MiB), verified, and kept only when it is not older than the copy already held or the bundled copy (an older signed document is a rollback and is refused); the stale cached copy; the bundled copy; and finally a document with every mode `"off"`. A cached copy no newer than the bundled one is not used, and one that is older is deleted. Network failures are not errors: they fall through to the next copy.

Side effects: an accepted live document and its signature are written as `morie-services.json` and `morie-services.sig` to the folder `tools::R_user_dir("rmoriebricklayer", "cache")`.

The result is also kept for the session, so later `offline = TRUE` calls (which the rest of the package makes) return it without checking the signature again.

Errors: `refresh` or `offline` not `TRUE` or `FALSE`; `maxage` not a single non-negative number; `tmout` not a single finite number of at least 1.

Value

A list with `version` (integer, 1), `issued` (the signing time, ISO 8601 UTC), `notice` (text for the user, possibly empty), `llm` (`mode`, `base_url`, `auth_url`: where the sign-in for the data key is reached; also `models`, `default_model` and `request_access`) and `data` (`mode`, `base_url`, `license`, `request_access`, and `auth_url` when the document has one), with the attribute `"source"` set to `"live"`, `"cache"`, `"bundled"` or `"off"`.

See Also

`dlogin`, `dmnf`.

Examples

```
# Nothing is fetched here: the cached copy if there is one, else the
# document bundled with the package.
s <- svcs(offline = TRUE)
s$llm$mode
attr(s, "source")

# where the curated-data service is, and whether it takes a key
s$data[c("mode", "base_url")]
s$issued

# arguments are checked
try(svcs(maxage = -1))


# ask the site now (network), and keep the answer for a day
svcs(refresh = TRUE)
```

`svycmp` *Compare survey estimates between domains*

Description

Each domain's estimate against a reference domain: the difference, its standard error from the design (the covariance between the two estimates included, which is what a pair of separate intervals leaves out), a confidence interval and a two-sided p-value from the normal approximation. Use it to test whether two regions or groups differ, rather than checking whether their separate intervals overlap: overlapping intervals do not mean no difference, and domains of one survey are not independent.

Usage

```
svycmp(
  des,
  data,
  y,
  by = NULL,
  stat = c("mean", "total", "proportion", "ratio"),
  den = NULL,
  lvls = NULL,
  ref = NULL,
  conflvl = 0.95
)
```

Arguments

des	A svydes() built on data.
data	The data frame the design was built on.
y	Character scalar, the analysis column: numeric for a mean, total or ratio; any type for a proportion (each distinct value is a category).
by	The domain columns to compare across.
stat	"mean", "total", "proportion" or "ratio" (of y to den).
den	For a ratio, the denominator column.
lvls	For a proportion, the categories to report (default all observed).
ref	The reference domain, as its label (for several by columns, the labels joined with " / "); a number is matched as the label it prints as. Default NULL: the first domain.
conflvl	Numeric scalar in (0, 1), the confidence level; default 0.95.

Details

The estimates come from svyest() with Wald intervals. For domain i and reference 0 the difference is $d = \hat{\theta}_i - \hat{\theta}_0$ with $SE(d) = \sqrt{V_{ii} + V_{00} - 2V_{i0}}$ from the design covariance; the interval is $d \pm z\,SE(d)$ and the p-value $2\Phi(-|d|/SE(d))$, both on the normal distribution (there is no df argument). For a proportion every category is compared separately. No adjustment for multiple comparisons is made.

Errors: by missing or empty; fewer than two domains; a ref that is not a single domain label present in the data; and every error of svyest().

Value

A data frame of class c("rmbl_svy_compare", "data.frame"), one row per non-reference domain (and category): domain, level (for a proportion only), numeric estimate (the domain's), reference (the reference domain's), difference, se, lower, upper and p_value. Attributes ref, stat, y and conflvl.

Examples

```
set.seed(1)
d <- data.frame(region = rep(c("north", "south", "east"), length.out = 150),
                smoker = rbinom(150, 1, 0.3), w = runif(150, 50, 150))
des <- svydes(d, w = "w")
svycmp(des, d, "smoker", by = "region", stat = "proportion",
               lvls = "1", ref = "north")

# means, against the default (first) domain, at 90 percent
d$drinks <- rpois(150, 3)
svycmp(des, d, "drinks", by = "region", conflvl = 0.90)

# the reference must be one of the domains
try(svycmp(des, d, "drinks", by = "region", ref = "west"))
```

---

svydes *Describe a complex survey design*

Description

Collects what design-based estimation needs: the final weight, and either a set of replicate weights or the strata and primary sampling units (PSUs) for Taylor linearization. svyest() and svycmp() take the result. Use it once per survey file: a public-use file from Statistics Canada or Health Canada ships its final weight and a set of bootstrap replicate weights, and estimates computed without them understate the standard errors. The design checks the replicate columns it is given and remembers the rows it was built on, so a later estimate cannot silently be run on a reordered or different table.

Usage

```
svydes(
  data,
  w,
  strata = NULL,
  psu = NULL,
  fpc = NULL,
  reps = NULL,
  id = NULL,
  type = c("bootstrap", "meanboot", "jk1", "jkn", "brr", "fay"),
  msize = 1,
  rho = NULL,
  mse = TRUE,
  lonely = c("fail", "remove", "adjust"),
  rscales = NULL,
  degf = NULL
)
```

Arguments

data	A data frame, one row per respondent.
w	The final (full-sample) weight: a column name or a numeric vector with one non-negative value per row.
strata, psu	Optional stratum and primary sampling unit: column names or vectors. A PSU label need only be unique within its stratum.
fpc	Optional finite population correction per row: a column name or vector holding either the stratum's population count of PSUs or (all values at most 1) the sampling fraction.
reps	Optional replicate weights: a data frame or matrix with one column per replicate, or the names of replicate weight columns in data.
id	With a reps data frame, the column (present in both) that matches its rows to data's; without it the rows must already be in the same order.
type	How the replicates were made: "bootstrap" (the default), "meanboot", "jk1", "jkn", "brr" or "fay"; see Details. Ignored without reps.
msize	For type = "meanboot", the number of bootstrap weights averaged into each replicate; default 1.
rho	For type = "fay", Fay's coefficient, between 0 and 1; default NULL.
mse	Logical scalar, default TRUE: centre replicate deviations on the full-sample estimate (TRUE) or on the mean of the replicate estimates (FALSE).
lonely	What to do with a stratum that has a single PSU under linearization: "fail" (the default; an error naming it), "remove" (it contributes nothing) or "adjust" (its PSU is centred on the mean of all PSU totals).
rscales	Per-replicate factors: required for type = "jkn" ($(n_h - 1)/n_h$ for each replicate's stratum), optional otherwise (multiplying the type's scale).
degf	The design's degrees of freedom, when known (a public-use file's documentation gives them); by default computed as described in Details.

Details

Replicate variances are $v = s \sum_r (\hat{\theta}_r - c)^2$, where $\hat{\theta}_r$ is the estimate under replicate weight r, c is the full-sample estimate (mse = TRUE) or the mean of the replicate estimates, and the scale s depends on how the replicates were made, for B replicates:

"bootstrap" $1/B$: the Rao-Wu rescaled bootstrap Statistics Canada ships with its public-use files. It is not the survey package's type = "bootstrap" (Canty and Davison's), whose scale is $m/((m-1)(B-1))$ for m PSUs a stratum: for those replicates give that factor over $1/B$ in rscales, $Bm/((m-1)(B-1))$ for each, with mse = FALSE (survey's default centring).

"meanboot" R/B, where each replicate weight is the mean of R bootstrap weights (msize). The Canadian Substance Use Survey 2023 ships 1,000 mean bootstrap weights with $R = 12$; Health Canada's technical notes give the factor of 12, equivalently Fay's BRR with $\rho = 1 - 1/\sqrt{12}$.

"jk1" $(B-1)/B$, the delete-one jackknife.

"brr" $1/B$, balanced repeated replication.

`"fay"` $1/(B(1-\rho)^2)$, Fay's BRR with coefficient `rho`.

`"jkn"` the stratified delete-one-PSU jackknife: each replicate has its own factor $(n_h-1)/n_h$ for the n_h PSUs of the stratum it deletes from, given as `rscales` (passed as `"jk1"`, JKn weights overstated the standard error by 37 percent in a test design).

The replicate columns are taken as given: which columns are replicates is not guessed from their values (rules that tried refused genuine designs, a dominant PSU or a jackknife on a subset). Refused, because no genuine replicate can be it: the `id`'s values, the row numbers, a column of `data` passed under another name, a second column named like the weight or the `id` that holds other values, and under JK1, JKn and BRR a copy of the full-sample weight (equal to it, or within 1 percent of it on every row; under JKn, a single replicate equal to it, while two or more equal to it are kept, as a stratum left out of a subset leaves all of its replicates at the weight). Not a copy: a jackknife replicate that scales the rows by one factor $k/(k-1)$ (its deleted PSU outside a subset), and, in a set whose replicates are post-stratified or calibrated (factors off 0, 1, 2 and $k/(k-1)$), a replicate equal or close to the weight. Warned about, because a genuine set can produce it but rarely does: under the bootstrap, more replicates equal to the full-sample weight than chance draws; under the bootstrap or Fay's method, a replicate within 1 percent of the weight on every row, and under Fay's method one equal to it (a post-stratified domain whose rows sit in one PSU per stratum); under JK1 and JKn, one factor $k/(k-1)$ on every row with no row left out that more or fewer replicates than k carry (a rescaled copy beside a full set, or a lone stray; a genuine set carries it on exactly k, on a subset too); under JK1, JKn and BRR, a replicate that leaves no row out; a replicate more than ten times the weight on a typical row (a column on another scale, such as an outcome); and a set whose replicates total, on average, under a tenth or over ten times the weight (multipliers such as **survey**'s `$repweights` passed in place of replicate weights). A column of `data` named in `reps`, and every column of an unnamed matrix, is a replicate because it was given as one; a stray that is itself shaped like a replicate (another design's multipliers, a constant under the bootstrap) cannot be told from one. BRR and Fay replicates must come in a Hadamard order (2, or a multiple of 4).

The data passed to `svyest()` must be the rows the design was built on, in the same order: rows in another order are refused (within strata or PSUs too), and a column changed since (a permuted outcome for a permutation test) is not a reorder.

The design's degrees of freedom (`df = "design"` in `svyest()`) are PSUs less strata under linearization; for replicate weights, the rank of the replicate weight matrix less one, as the `survey` package computes them (twelve BRR half-samples of an eight-stratum design have 8, not 11), or `degf` when given.

Without replicates, the variance is the with-replacement Taylor linearization: within each stratum h with n_h PSUs, $(1-f_h)n_h/(n_h-1)$ times the sum of squared deviations of the PSU totals of the weighted linearized values from their stratum mean, summed over strata (f_h from `fpc`, zero without one). No strata means one stratum; no PSUs means every respondent is its own PSU.

Errors: `data` not a data frame or with fewer than two rows; a named column that `data` does not have, or a vector of the wrong length; weights that are negative, missing or not finite, or replicate weights that are missing or not finite (calibrated replicates may be negative); both replicates and strata or PSUs; fewer than two replicates, or replicate columns refused for the reasons above; BRR or Fay replicates not in a Hadamard order; `type = "jkn"` without `rscales`, or `rscales` not one non-negative number per replicate; `rho` not in [0, 1) for Fay; `msize` or `degf` below 1; missing strata or PSU labels; an `fpc` that is not positive, varies within a stratum, or is smaller than the PSUs sampled. Nothing is computed until `svyest()` is called, so a single-PSU stratum under `lonely = "fail"` is reported then.

Value

An object of class `rmbl_svy_design`: a list with `n` (respondents), `weights`, `type` (the replicate type, or `"linearization"`), `scale` and `reps` (the replicate scale and weights, `NULL` under linearization), `mse`, `strata` and `psu` (integer codes), `lonely`, `check` (the row names and digests used to recognise the data later) and `degf` (the design degrees of freedom). The print method summarises it.

See Also

`svyest()`, `svycmp()`

Examples

```
set.seed(1)
d <- data.frame(region = rep(c("north", "south"), each = 50),
                smoker = rbinom(100, 1, rep(c(0.2, 0.35), each = 50)),
                w = runif(100, 50, 150))
# ten bootstrap replicates, made here by resampling for the example
reps <- replicate(10, d$w * tabulate(sample(100, replace = TRUE), 100))
des <- svydes(d, w = "w", reps = reps, type = "bootstrap")
des
# the same data as a stratified design, variance by linearization
svydes(d, w = "w", strata = "region")

# delete-one jackknife replicates over ten groups of respondents
grp <- rep(1:10, length.out = 100)
jk <- sapply(1:10, function(g) ifelse(grp == g, 0, d$w * 10 / 9))
svydes(d, w = "w", reps = jk, type = "jk1")

# a weight column that does not exist
try(svydes(d, w = "wt"))
```

---

svyest *Estimate a mean, total, proportion or ratio from a survey design*

Description

Design-based point estimates with standard errors, confidence intervals and coefficients of variation, for the whole population or for each domain of `by`. A respondent outside a domain, or missing the analysis variable, counts zero in that domain and stays in the design, so the variance of a subgroup estimate reflects the whole sample design. Use it for the figures a survey report publishes (the prevalence of a behaviour by region, the mean number of drinks, a population total) with the standard errors the survey's design implies rather than those of a simple random sample.

Usage

```
svyest(
  des,
```

```
  data,
  y,
  by = NULL,
  stat = c("mean", "total", "proportion", "ratio"),
  den = NULL,
  lvls = NULL,
  conflvl = 0.95,
  ci = c("auto", "wilson", "logit", "wald"),
  minn = 30,
  df = Inf
)
```

Arguments

des	A svydes() built on data.
data	The data frame the design was built on.
y	Character scalar, the analysis column: numeric for a mean, total or ratio; any type for a proportion (each distinct value is a category).
by	Optional character vector of domain columns; NULL (the default) estimates for the whole population.
stat	"mean", "total", "proportion" or "ratio" (of y to den).
den	For a ratio, the denominator column.
lvls	For a proportion, the categories to report (default all observed).
conflvl	Numeric scalar in (0, 1), the confidence level; default 0.95.
ci	Interval method: "auto" (the default; Wilson for proportions, Wald otherwise), "wilson", "logit" or "wald". Wilson and logit apply to proportions only.
minn	Numeric scalar, default 30: domains with fewer respondents are flagged.
df	Degrees of freedom for the interval: Inf (the default) for the normal quantile, a number for Student's t, or "design" for the design's own (PSUs less strata under linearization, replicates less one otherwise).

Details

The estimates are ratios of weighted totals: a mean is $\sum w_i y_i / \sum w_i$ over the domain, a total $\sum w_i y_i$, a proportion the mean of the 0/1 indicator of each category, and a ratio $\sum w_i y_i / \sum w_i x_i$ with x the den. Their variances come from the design: the replicate formula, or Taylor linearization of the ratio (see svydes()).

Proportions are reported for every category of y (or those in lvls), each with its own interval. The default interval for a proportion is Wilson's score interval at the modified effective sample size of SAS PROC SURVEYFREQ, the software behind Health Canada's published Canadian Substance Use Survey intervals: n/Deff, with the design effect taken against the simple-random-sampling variance $p(1-p)/(n-1)$, scaled by $(t_{n-1}/t_{df})^2$ for the design's degrees of freedom (Korn and Graubard 1998; t_{df} is the normal quantile when the design's degrees of freedom are infinite) and capped at the n respondents behind the estimate. ci = "logit" builds the interval on the logit scale and "wald" symmetrically. Means, totals and ratios get the symmetric normal interval.

A domain with fewer than `minn` respondents is flagged, not withheld: Health Canada suppresses such CSUS estimates in its data tool (denominator under 30), and the flag lets the reader do the same. Other flags say why a standard error is missing (no stratum with two PSUs, a replicate that left the domain empty, weights all zero) or that a domain has no respondents.

Errors: `des` not from `svydes()`; `data` not the rows the design was built on, in the same order; `y`, `by` or `den` naming a column `data` does not have; a ratio without `den`; `conflvl` outside (0, 1); `df` not positive, `Inf` or `"design"`, or a design with no degrees of freedom under `df = "design"`; a single-PSU stratum under `lonely = "fail"`.

Value

A data frame of class `c("rmbl_svy_estimate", "data.frame")`, one row per domain (and category, for a proportion): the domain columns (and `level`), then numeric `estimate`, `se`, `lower`, `upper`, `cv` (coefficient of variation, percent), integer `n` (respondents counted) and character `flag` (`NA` when there is nothing to say). Attributes: `vcov` (the covariance matrix of the estimates, used by `svycmp()`), `stat`, `y`, `by`, `ci`, `conflvl`, `df`, `method` (the variance method) and `denominator`.

See Also

`svydes()`, `svycmp()`

Examples

```
set.seed(1)
d <- data.frame(region = rep(c("north", "south"), each = 50),
                smoker = rbinom(100, 1, rep(c(0.2, 0.35), each = 50)),
                drinks = rpois(100, 3), w = runif(100, 50, 150))
des <- svydes(d, w = "w", strata = "region")
svyest(des, d, "smoker", by = "region", stat = "proportion", lvls = "1")
svyest(des, d, "drinks", stat = "mean")
svyest(des, d, "drinks", by = "region", stat = "total")

# a ratio of two totals, with Student's t on the design's degrees of freedom
d$visits <- rpois(100, 6)
svyest(des, d, "drinks", stat = "ratio", den = "visits", df = "design")

# a logit interval at 90 percent
svyest(des, d, "smoker", stat = "proportion", lvls = "1", ci = "logit",
                conflvl = 0.90)

# the design remembers its rows: a subset is refused
try(svyest(des, d[-1, ], "drinks"))
```

`swtdid` *Difference in differences for treatments that switch on and off*

Description

De Chaisemartin and D'Haultfoeuille's estimator of the effect at the moment units switch: in every period, units that switch into treatment are compared with units untreated in both periods, and units that switch out with units treated in both; with the placebo that runs the same comparison one period earlier.

Usage

```
swtdid(
  data,
  unit,
  time,
  y,
  trtd,
  cl = NULL,
  reps = 499L,
  seed = NULL,
  conflvl = 0.95
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`trtd`	Column name of the treatment indicator (0 or 1).
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.
`reps`	Bootstrap replications (default 499; 0 for none).
`seed`	Optional seed for the bootstrap.
`conflvl`	Confidence level of the intervals reported (default 0.95).

Details

For each period t with joiners ($D_{t-1} = 0, D_t = 1$) and stayers untreated in both, $DID_{+,t}$ is the joiners' mean change in the outcome minus the stayers'; with leavers (1 to 0) and stayers treated in both, $DID_{-,t}$ is the stayers' change minus the leavers'. The estimate is

$$DID_M = \sum_t \frac{N_{J,t}}{N_S} DID_{+,t} + \frac{N_{L,t}}{N_S} DID_{-,t},$$

over the N_S switchers. The placebo compares, between $t-2$ and $t-1$, the change of the units that switch at t with that of the units whose treatment did not change over $t-2$ to t; it should be near

zero. Standard errors come from the cluster bootstrap. Unlike the two-way estimator, DID_M never uses a unit whose treatment changed as a control in the period it changed.

Value

A data frame of class `rmbl_switching_did`, two rows (`effect`, `placebo`): `estimate`, `se`, `lower`, `upper`, `switchers`.

References

de Chaisemartin C, D'Haultfoeuille X (2020). Two-way fixed effects estimators with heterogeneous treatment effects. American Economic Review 110(9):2964-2996. DOI 10.1257/aer.20181169

Applied as a placebo check on ward redistricting: Ternullo S, Zorro-Medina A, Vargas R (2024). How political dynasties concentrate advantage within cities. Social Forces 102(4):1310-1331. DOI 10.1093/sf/soae029

Examples

```
p <- polex()
swtdid(p, "district", "month", "crmrate", "reformed", reps = 99, seed = 1)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
swtdid(own, "town", "year", "rate", "bylaw", reps = 49, seed = 1)
```

---

synctl *Synthetic control for one treated unit*

Description

A weighted average of untreated units that tracks the treated unit before the policy, as its counterfactual after; with the placebo test that runs the same method on every untreated unit.

Usage

```
synctl(data, unit, time, y, treunit, adopt, plac = TRUE)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`treunit`	The treated unit (a value of `unit`).
`adopt`	The first treated period.
`plac`	Run the placebo test (default `TRUE`).

Details

The weights solve

$$\min_{w} \sum_{t<T_0} \Big(Y_{1t} - \sum_{j} w_j Y_{jt}\Big)^2 \quad \text{s.t.} \quad w_j \geq 0,\ \sum_{j} w_j = 1,$$

over the periods before adoption T_0 (accelerated projected gradient onto the simplex; the problem is convex, so the solution is the global one). The effect in each period is the gap $Y_{1t} - \sum_j w_j Y_{jt}$. For inference, each untreated unit in turn is treated as if it had adopted the policy, with the other untreated units as donors; the ratio of root mean squared gap after to before adoption ranks the real unit among the placebos, and p is its rank over the number of units (Abadie, Diamond and Hainmueller 2010).

This version matches on the pre-period outcomes only, not on separate predictors.

Value

A list of class `rmbl_synth`: `weights` (a named vector over the donors), `path` (a data frame: `time`, `treated`, `synthetic`, `gap`), `rmspepre`, `rmsperatio`, `p_value` (`NA` without the placebo), `placrat`. `plot()` draws the two paths and the gap.

References

Abadie A, Diamond A, Hainmueller J (2010). Synthetic control methods for comparative case studies. Journal of the American Statistical Association 105(490):493-505.
DOI 10.1198/jasa.2009.ap08746

Examples

```
p <- polex()
p1 <- p[is.na(p$refmonth) | p$district == "D01", ]
s <- synctl(p1, "district", "month", "crmrate", treunit = "D01",
                        adopt = 13)
round(s$weights, 3)
s$p_value
plot(s)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
```

```
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# one treated town, A, against the four towns never treated
own1 <- own[is.na(own$first) | own$town == "A", ]
synctl(own1, "town", "year", "rate", treunit = "A", adopt = 2019)
```

`syndid` *Synthetic difference in differences*

Description

Arkhangelsky et al.'s estimator for units adopting a policy at one date: comparison units weighted so their pre-period path parallels the treated units', pre-periods weighted so they predict the post-period, and the difference in differences taken with both weights.

Usage

```
syndid(data, unit, time, y, ftreat, reps = 200L, seed = NULL, conflvl = 0.95)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`ftreat`	Column name of the period a unit is first treated (`NA`, 0 or `Inf` for a unit never treated). Treatment is taken as staying on.
`reps`	Placebo replications (default 200).
`seed`	Optional seed for the placebo draws.
`conflvl`	Confidence level of the intervals reported (default 0.95).

Details

With N_0 comparison units, N_1 treated, T_0 pre-periods and T_1 post-periods, the unit weights solve

$$\min_{\omega_0,\omega\in\Delta}\frac{1}{T_0}\sum_{t\le T_0}\left(\omega_0+\sum_j\omega_j Y_{jt}-\bar{Y}_{1t}\right)^2+\zeta^2\|\omega\|^2,$$

with $\zeta = (N_1 T_1)^{1/4}\hat{\sigma}$ and $\hat{\sigma}$ the standard deviation of the comparison units' first differences before adoption; the time weights solve the same problem across comparison units, predicting each unit's post-period mean from its pre-periods, with $\zeta = 10^{-6}\hat{\sigma}$. The estimate is

$$\hat{\tau} = \Big(\bar{Y}_{1,post} - \sum_t \lambda_t \bar{Y}_{1t}\Big) - \sum_j \omega_j \Big(\bar{Y}_{j,post} - \sum_t \lambda_t Y_{jt}\Big).$$

The weights are the exact solutions of these convex problems (the `synthdid` package runs Frank-Wolfe and then sparsifies, so its estimate differs slightly). The standard error is the placebo one when there is one treated unit (each set of N_1 comparison units in turn treated as if adopting, weights re-estimated), else the jackknife over units with the weights held fixed (Arkhangelsky et al. 2021, algorithms 3 and 4).

Value

A list of class `rmbl_synthdid`: `estimate`, `se`, `lower`, `upper`, `unitwts`, `tmwts`, `method` (`"placebo"` or `"jackknife"`), `paths` (treated and weighted comparison means by period). `plot()` draws the paths.

References

Arkhangelsky D, Athey S, Hirshberg DA, Imbens GW, Wager S (2021). Synthetic difference-in-differences. American Economic Review 111(12):4088-4118. DOI 10.1257/aer.20190159

Examples

```
p <- polex()
p1 <- p[is.na(p$refmonth) | p$refmonth == 13, ]
syndid(p1, "district", "month", "crmrate", "refmonth", reps = 50, seed = 1)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
own1 <- own[is.na(own$first) | own$first == 2019, ]
syndid(own1, "town", "year", "rate", "first", reps = 50, seed = 1)
```

`toascii` *Transliterate Text to Plain ASCII*

Description

Converts a character vector to plain 7-bit ASCII, transliterating accented or non-Latin characters to their nearest ASCII equivalent (for example, an accented capital A becomes a plain `"A"`). Falls back to dropping any character that has no transliteration. This is the deterministic `"fallback"` used by `asciifb()`. Use it for destinations that accept only ASCII – a package `DESCRIPTION`, a file name, a legacy system – so a name written with an accented letter arrives with the plain letter rather than as an encoding error or a question mark.

Usage

```
toascii(x)
```

Arguments

`x`	A character vector (other types are converted with `as.character()`).

Details

Bytes that are not valid UTF-8 at all (a mislabelled file) are dropped first. Then, when the **stringi** package is installed, the ICU transform `"Any-Latin; Latin-ASCII"` romanises any script and folds accents (Cyrillic, Greek and Chinese become Latin letters). Without **stringi**, a fixed table folds the common Latin accents and ligatures (sharp s to `"ss"`, the ae ligature to `"ae"`, thorn to `"th"` and so on) and every other non-ASCII character is dropped, so non-Latin scripts disappear rather than being romanised. Either way the result is then stripped of anything outside the printable ASCII range (space to tilde), so control characters such as newlines and tabs are removed too. The result never depends on the session's locale.

A missing value becomes `""`; non-character input is converted with `as.character()` first (so numbers pass through as their digits).

Value

A character vector the length of `x`, containing only printable ASCII characters (bytes 32 to 126); `NA` elements become `""`.

See Also

`asciifb()` to transliterate only when needed, `wrtxtfb()` to write text with an ASCII retry.

Examples

```
# Latin accents fold to their nearest ASCII letter.
toascii("Prof. \u00c1ngela Zorro Medina")  # "Prof. Angela Zorro Medina"

# Vectorised over the input.
```

```
toascii(c("Se\u00e1n", "Zo\u00eb", "na\u00efve"))

# Non-Latin scripts are romanised when stringi is available.
if (requireNamespace("stringi", quietly = TRUE))
  toascii("\u041c\u043e\u0441\u043a\u0432\u0430")  # "Moskva" (Cyrillic)

# Either way the result is guaranteed pure 7-bit ASCII (never "?").
all(charToRaw(toascii("caf\u00e9")) < 128)

# Missing values become empty strings; line breaks are removed.
toascii(c(NA, "two\nlines"))
```

---

tojs *Encode an R object as JSON (jsonlite's toJSON, natively)*

Description

Converts R data (vectors, lists, data frames, matrices, dates, factors) to JSON text, with the same options, the same defaults and the same bytes out as jsonlite's `toJSON()`, but computed by this package's own code, so manifests, provenance records and capsules can be written where jsonlite is not installed and their digests stay the same either way. Use it to write data for an API or a person; use `jsser()` when the JSON must rebuild the exact R object (classes, factor levels, attributes).

Usage

```
tojs(
  x,
  dataframe = c("rows", "columns", "values"),
  matrix = c("rowmajor", "columnmajor"),
  Date = c("ISO8601", "epoch"),
  POSIXt = c("string", "ISO8601", "epoch", "mongo"),
  factor = c("string", "integer"),
  complex = c("string", "list"),
  raw = c("base64", "hex", "mongo", "int", "js"),
  null = c("list", "null"),
  na = NULL,
  auto_unbox = FALSE,
  digits = 4,
  pretty = FALSE,
  force = FALSE,
  ...
)
```

Arguments

x	The object to encode.

dataframe	How a data frame is written: "rows" (the default, an array of records), "columns" (an object of column arrays) or "values" (an array of row arrays, without names).
matrix	How a matrix is written. "rowmajor" (the default) gives an array of rows, and "columnmajor" an array of columns.
Date	How a Date is written: "ISO8601" (the default, "2024-01-02") or "epoch" (days since 1970-01-01).
POSIXt	How a date-time is written: "string" (the default, "2024-01-02 03:04:05"), "ISO8601" ("2024-01-02T03:04:05"), "epoch" (milliseconds since 1970) or "mongo" ({"$date": ms}).
factor	How a factor is written: "string" (the default, its labels) or "integer" (its codes).
complex	How a complex number is written: "string" (the default, "1+2i") or "list" (an object of real and imaginary arrays).
raw	How a raw vector is written: "base64" (the default, one string), "hex", "mongo", "int" or "js".
null	How NULL inside a list is written: "list" (the default, {}; a round trip gives an empty list) or "null" (a JSON null). jsser() round-trips NULL exactly.
na	How missing values are written: "null" or "string" ("NA"). When NULL (the default), the per-type default of jsonlite applies (see Details).
auto_unbox	Logical scalar, default FALSE: write length-one vectors as scalars (1) rather than arrays ([1]).
digits	Numeric scalar, default 4: decimal places kept for doubles. I(n) means n significant digits (I(17) writes every double exactly) and NA means 15 significant digits.
pretty	FALSE (the default) for compact output, TRUE to indent by two spaces, or a number of spaces.
force	Logical scalar, default FALSE: write objects that have no JSON form (as {}) instead of failing.
...	Further options of jsonlite's toJSON() and of this package (see Details).

Details

Atomic vectors become arrays, so a single value is [1] unless auto_unbox = TRUE. Named lists become objects and unnamed lists arrays. Missing values follow jsonlite: with na not given, a missing number is written as the string "NA" and a missing string or logical as null; in a data frame by rows a missing value is left out of its record.

Further options of toJSON() are taken through ...: rownames, keep_vec_names, json_verbatim, always_decimal, time_format, UTC, no_dots, hms ("string" or "secs"), and two of this package's own: keep_empty_names (an empty-string key is written as "" rather than replaced by its position) and quiet_vec_names (no message when keep_vec_names is used). mnfcanon() uses both.

Errors: an option value not among its choices (the message lists them); an object with no JSON form (a function, a symbol) unless force = TRUE.

Value

A character scalar of class `json`, the JSON text.

See Also

`fromjs()` for the reverse, `jsser()` for an exact round trip.

Examples

```
tojs(list(a = 1:3, b = "x"), auto_unbox = TRUE)
tojs(data.frame(id = 1:2, v = c(1.5, NA)), pretty = TRUE)

# a data frame by columns, and a date-time in ISO 8601
df <- data.frame(id = 1:2, v = c(1.5, NA))
tojs(df, dataframe = "columns", na = "null")
tojs(as.POSIXct("2024-01-02 03:04:05", tz = "UTC"), POSIXt = "ISO8601")

# digits: decimal places by default, significant digits with I()
tojs(pi)
tojs(pi, digits = I(17))

# a choice that does not exist is refused
try(tojs(df, dataframe = "table"))
```

---

topcors *Strongest pairwise correlations in a data frame*

Description

Ranks the numeric column pairs by the strength of their association and returns the strongest, so a wide table's structure can be read without squinting at a correlation matrix. Use it on a new data set to find near-duplicate columns, derived columns and the relationships worth a closer look.

Usage

```
topcors(data, n = 10L, meth = c("spearman", "pearson"), minabs = 0)
```

Arguments

`data`	A data frame; its numeric pairs are ranked by correlation.
`n`	Number of pairs to return, one positive whole number (default 10).
`meth`	`"spearman"` (default) or `"pearson"`.
`minabs`	Report only pairs whose absolute correlation reaches this, a number in $[0, 1]$ (default 0).

Details

Spearman is the default deliberately. Pearson measures LINEAR association only, so it understates a relationship that is perfectly monotone but curved, and a single outlier can manufacture or destroy it. On data you have not yet inspected – which is the situation this function is for – the rank correlation is the safer question to ask.

Non-numeric columns are ignored. Each pair uses the rows where both values are present; a pair with fewer than three such rows, or with a constant column, has no correlation and is left out. Pairs are ranked by absolute correlation, those below `minabs` dropped, and the first `n` returned. `cortbl()` gives every pair, with the number of rows behind each. A progress bar is drawn on stderr for many pairs unless `options(morie.quiet = TRUE)`.

Errors: `data` not a data frame; fewer than two numeric columns; `n` not one positive whole number; `minabs` not one number in $[0, 1]$; an invalid `meth`.

Value

A data frame of class `c("bricklayer_correlations", "data.frame")` with `x` and `y` (character, column names), `correlation` (numeric) and `abscor` (numeric), strongest first, at most `n` rows; the method is in `attr(, "method")`. The print method draws a signed bar per pair.

See Also

`cspr()`, `ccovm()`, `cortbl()`

Examples

```
set.seed(1)
n <- 200
df <- data.frame(
  a = stats::rnorm(n),
  b = stats::rnorm(n),
  grade = sample(letters[1:3], n, TRUE)
)
df$c <- df$a * 2 + stats::rnorm(n, sd = 0.1) # strongly related to a
df$d <- exp(df$a) # monotone but curved

topcors(df)

# The curved pair is ranked correctly by Spearman and understated by
# Pearson, which is why Spearman is the default.
tc <- topcors(df, meth = "spearman")
tc[tc$x == "a" & tc$y == "d", "correlation"]
tp <- topcors(df, meth = "pearson")
tp[tp$x == "a" & tp$y == "d", "correlation"]

# Filter to the pairs worth looking at, and keep the top two.
topcors(df, n = 2, minabs = 0.5)

# a threshold above 1 is an error
try(topcors(df, minabs = 2))
```

`trccmpl` *Calls to Action completed, year by year*

Description

The number of the Truth and Reconciliation Commission's 94 Calls to Action that the Yellowhead Institute's annual accountability reports judged complete, by the year they were completed, to the end of 2022: the only counts here are those the reports state.

Usage

```
trccmpl()
```

Details

2015-2019: 8 (13 in all by December 2022, less those of 2020-2022); 2020: none; 2021: 3, all in June; 2022: 2 (Jewell and Mosby 2021, 2022). Other trackers count otherwise: the federal government reports by theme rather than call by call; CBC's Beyond 94 and Indigenous Watchdog grade each call on their own scales and are copyrighted, so their tables are not reproduced. The reports are licensed CC BY-NC 2.5 Canada; the counts are facts taken from them, with the reports cited.

Value

A data frame with `period`, `yearend`, `completed`, `cumulative` and `source`.

References

Jewell E, Mosby I (2021). Calls to Action Accountability: A 2021 Status Update on Reconciliation. Yellowhead Institute, Toronto Metropolitan University.

Jewell E, Mosby I (2022). Calls to Action Accountability: A 2022 Status Update on Reconciliation. Yellowhead Institute, Toronto Metropolitan University.

Truth and Reconciliation Commission of Canada (2015). Truth and Reconciliation Commission of Canada: Calls to Action. Winnipeg.

See Also

`trcprog`.

Examples

```
trccmpl()
```

`trchlthgap` *The Call to Action 19 health gaps, in one place*

Description

For each health indicator of Call to Action 19 that the published tables measure, each group's estimate against non-Indigenous people's, with its interval, turned into one scale, the disadvantage ratio: the factor by which the group's outcome is worse (above

1. or better (below 1).

Usage

```
trchlthgap(
  health = indiex("health"),
  lifeexp = indiex("lifeexp"),
  infmort = indiex("infantmort"),
  suiide = indiex("suicide"),
  ref = "Non-Indigenous",
  agetot = NULL
)
```

Arguments

`health`	A health indicators table (default `indiex("health")`, table 13-10-0924), or `NULL`.
`lifeexp`	Life expectancy (default `indiex("lifeexp")`, table 17-10-0160), or `NULL`.
`infmort`	Infant mortality (default `indiex("infantmort")`, table 17-10-0161), or `NULL`.
`suiide`	Suicidal thoughts (default `indiex("suicide")`, table 41-10-0021), or `NULL`. The name is kept as it was first published; it is the suicidal-ideation table.
`ref`	The reference group's label (default `"Non-Indigenous"`).
`agetot`	The label of the all-ages row in the `age` column, one string for both tables or two (`health`, then `suiide`). Default `NULL`: Statistics Canada's `"Total, 18 years and over"` and `"Total, 15 years and over"`. A table of your own with no `age` column is used whole.

Details

The gaps come from `indgap` on the latest period of each table, with `ref` (non-Indigenous people) as the reference. To put years, rates and percentages on one scale, each gap is given as a ratio oriented by the indicator's better direction (`trcind`): the group's estimate over the reference's where lower is better (infant mortality, chronic conditions), the reference's over the group's where higher is better (life expectancy, good mental health, a regular provider):

$$D = \hat{\theta}_g / \hat{\theta}_r \text{ (lower is better)}, \qquad D = \hat{\theta}_r / \hat{\theta}_g \text{ (higher is better)}.$$

It is a ratio of the published estimates, not a relative risk: for life expectancy, D = 1.1 means the reference lives 10% longer, not that the group's risk is 10% higher. Above 1 the group fares worse; the interval is the published one carried through (`indgap`'s log-scale interval, inverted where needed). The difference on the indicator's own scale is kept beside it.

This is a description of gaps that Call 19 asks to be closed, measured as the surveys measure them; it is not a ranking of peoples. Report it with the trends (`healchg`) and the strengths (`welprof`) that sit beside it.

Each table is in the long layout of `scests` (or `indiex()`): columns `group`, `indicator` or `measure`, `period`, `estimate`, `lower`, `upper`, `quality`, and `age` for the health and suicidal-ideation tables (the all-ages total is used). The latest period with estimates is taken. Table 41-10-0021 publishes Indigenous groups only, so it adds a gap only when given a table with `ref` rows.

Value

A data frame of class `rmbl_trc_gaps`: `indicator`, `measure`, `period`, `group`, `estimate`, `refest`, `difference` (group minus reference, on the indicator's scale), `disratio`, `lower`, `upper`, `worse` (the interval above 1), `usewithcau`. `plot()` draws it.

References

Truth and Reconciliation Commission of Canada (2015), see `trcind`.

See Also

`trcind`, `indgap`.

Examples

```
g <- trchlthgap()
g[g$group == "Inuit", c("indicator", "measure", "period", "disratio",
                        "lower",
                        "upper")]
plot(g)

# your own table: made-up life expectancies with their limits, in the
# layout of indiex("lifeexp"); the tables you do not have are NULL
le <- data.frame(
  period = "2017/2019",
  group = c("First Nations", "Inuit", "Non-Indigenous"),
  estimate = c(74.2, 70.9, 81.8),
  lower = c(73.6, 69.1, 81.6),
  upper = c(74.8, 72.7, 82.0),
  quality = "ok"
)
trchlthgap(health = NULL, lifeexp = le, infmort = NULL, suiide = NULL)
```

`trcind` *The indicators of Calls to Action 19 and 55, and the tables that measure them*

Description

Call to Action 19 asks the federal government to "establish measurable goals to identify and close the gaps in health outcomes between Aboriginal and non-Aboriginal communities, and to publish annual progress reports and assess long-term trends", focusing on "infant mortality, maternal health, suicide, mental health, addictions, life expectancy, birth rates, infant and child health issues, chronic diseases, illness and injury incidence, and the availability of appropriate health services". Call to Action 55 asks for annual data on, among others, "the number of Aboriginal children–including Metis and Inuit children–in care, compared with non-Aboriginal children" and "the educational and income attainments of Aboriginal peoples in Canada compared with non-Aboriginal people". This lists, for each indicator, the Statistics Canada tables at the data service (or the census profile) that measure it, the measure, and which direction is better.

Usage

```
trcind()
```

Details

The list is a map of what the published tables can measure, not a judgement of what the Calls require: maternal health is measured here only through birth outcomes in Inuit regions, and several indicators (addictions, the availability of services) only through proxies. The direction is the direction of better health for the people measured.

Value

A data frame with `call`, `indicator` (the Call's words), `table`, `measure`, `better` (`"higher"` or `"lower"`) and `note`.

References

Truth and Reconciliation Commission of Canada (2015). Truth and Reconciliation Commission of Canada: Calls to Action. Winnipeg. Government of Canada catalogue IR4-8/2015E. publications.gc.ca/collections/collection 8-2015-eng.pdf

See Also

`trchlthgap`, `trcprog`.

Examples

```
ti <- trcind()
ti[ti$call == "19", c("indicator", "table", "measure", "better")]
```

`trcprog` *How fast the Calls to Action are being completed*

Description

From cumulative counts of completed Calls to Action over time, the completion rate (calls a year), its exact Poisson interval, and the year all 94 would be complete at that rate.

Usage

```
trcprog(
  done = trccmpl()$cumulative,
  yearend = trccmpl()$yearend,
  total = 94,
  start = 2015.96,
  conflvl = 0.95
)
```

Arguments

`done`	Cumulative number completed at each date (default: `trccmpl`).
`yearend`	The year each count was made to (end of year).
`total`	Number of Calls to Action (94).
`start`	When counting began, as a year with a fraction (2015.96: December 15, 2015).
`conflvl`	Confidence level of the completion rate (default 0.95).

Details

Completions are taken as a Poisson process with a constant rate over the span from the start (`start`, the Commission's final report in December 2015) to the last count: the rate is c/T with the exact (Garwood) interval for c completed calls, and the projection is the year $94 - c$ more would take at that rate (at the interval's ends, its range). A constant rate is the simplest model, and the one behind the Yellowhead Institute's projection ("42 years, or until 2065", 2022); with the counts to the end of 2022 and December 2015 as the start it gives 2067. The pace has not been constant, so the projection is a description of the pace so far, not a forecast. Any count of commitments completed over time can be modelled the same way: set `total` and `start`.

Value

A one-row data frame of class `rmbl_trc_progress` with the columns `completed`, `remaining`, `years`, `rate`, `ratelo`, `rateup`, `cmplby`, `cmplbyearly` and `cmplbylate`; `plot()` draws the completions and the projection, with the band the rate's interval gives.

See Also

`trccmpl`.

Examples

```
p <- trcprog()
p
plot(p)

# your own count: Calls completed by the end of each year, made up
trcprog(done = c(2, 6, 9, 13), yearend = 2016:2019)
```

`trndtst` *Trend in a short series*

Description

Tests whether a short series (for example five to ten fiscal years of an administrative count) is rising or falling, and estimates the size of the change per period. It uses the Mann-Kendall rank test for monotone trend and the Theil-Sen median-of-slopes estimator: no distributional assumption, resistant to a single aberrant period, and meaningful at the series lengths an annual administrative extract actually has.

Usage

```
trndtst(
  y,
  x = NULL,
  value = NULL,
  prd = NULL,
  exact = NULL,
  alt = c("two.sided", "increasing", "decreasing"),
  conflvl = 0.95
)
```

Arguments

`y`	Numeric vector, the series in period order, or a data frame.
`x`	Numeric vector of periods, default `NULL`: the position `1, 2, ...`, which is right for an evenly spaced series. Ignored when `y` is a data frame and `prd` is given.
`value`, `prd`	Column names (strings), when `y` is a data frame. `value` is required; `prd` is optional and must be convertible with `as.numeric()`.
`exact`	Logical or `NULL` (the default). Whether to compute the exact null distribution of Mann-Kendall's S by enumeration. `NULL` uses it up to `n = 8` (40,320 orderings); above that the normal approximation with the tie and continuity corrections is used.
`alt`	`"two.sided"` (the default), `"increasing"` or `"decreasing"`.
`conflvl`	Number in (0, 1), default 0.95: the confidence level of the slope interval.

Details

With values $y_1, \ldots, y_n$ in period order,

$$S = \sum_{i<j} \operatorname{sign}(y_j - y_i),$$

and Kendall's tau here is $S/(n(n-1)/2)$, the rank correlation between the value and the period. The Theil-Sen slope is the median of the pairwise slopes $(y_j - y_i)/(x_j - x_i)$; `intercept` is the line's intercept from the compiled Theil-Sen routine.

For $n \leq 8$ the p-value is exact by default: S is computed over every ordering of the observed values (with ties kept, so tied data get the correct conditional null). Otherwise the normal approximation is used, with the tie-corrected variance of S and a continuity correction of one unit toward zero. Exact tables are cached in the session, up to 32 of them. A series whose values are all tied has $S = 0$ in every ordering, so its p-value is 1 at every length.

The slope interval is the standard rank-based one: the pairwise slopes are sorted and the interval runs between the order statistics that Mann-Kendall's variance places at the chosen level, so it is consistent with the test rather than derived from a different model. At small n the slopes may not reach the requested confidence; the bound is then the most extreme slope and `slopecicla` is `TRUE`.

Missing periods are dropped and the series is sorted by `x`. Errors: fewer than three periods; infinite values; `x` and `y` of different lengths; a period given twice in `x`; a data frame without `value` or with a column not found; more than 20,000 periods (the pairwise slopes would not fit in memory); a `conflvl` outside (0, 1). `exact = TRUE` with $n > 8$ warns and uses the normal approximation.

Value

A list of class `rmbl_trend_test` with `S`, `tau`, `p_value`, `slope` (Theil-Sen, in units of `y` per unit of `x`), `intercept`, `slopelo` and `slopeup` (the distribution-free interval), `slopecicla` (`TRUE` when the pairwise slopes do not reach the requested confidence on a side, so that bound is the extreme slope and the true interval is wider), `n`, `var_S` (the tie-corrected variance of S), `alternative`, `conflvl` and `method`.

References

Sen, P. K. (1968). Estimates of the regression coefficient based on Kendall's tau. *Journal of the American Statistical Association* 63(324), 1379-1389, for the median-of-slopes estimator and the distribution-free interval.

Wilcox, R. R. *Modern Statistics for the Social and Behavioral Sciences: A Practical Introduction* treats Theil-Sen among the regression methods that carry no normality assumption, which is the reason for preferring it on a series this short.

See Also

`stepchg`, `cnttrnd`

Examples

```
# Five years of placements.
y <- c(402, 377, 190, 268, 331)
```

```
trndtst(y)

# A monotone series is detected even at n = 5, where a regression's
# standard error would be nearly uninformative.
trndtst(c(1, 2, 3, 4, 5))

# One aberrant period does not create a trend.
trndtst(c(100, 100, 100, 100, 900))$p_value

# From a data frame, testing for a decrease only.
d <- data.frame(year = 2019:2023, n = y)
trndtst(d, value = "n", prd = "year", alt = "decreasing")$p_value

try(trndtst(c(5, 6)))
```

`tse` *How far parallel trends can fail before an effect disappears*

Description

Bounds on the effects of an `evstudy` when the trend after adoption may depart from parallel by up to $\bar{M}$ times the largest departure seen before it (Rambachan and Roth's relative magnitudes), with a conservative interval and the `"breakdown"` $\bar{M}$ at which the interval first includes zero.

Usage

```
tse(
  x,
  mbar = c(0, 0.5, 1, 1.5, 2),
  target = "average",
  draws = 5000L,
  seed = NULL,
  conflvl = 0.95
)
```

Arguments

`x`	An `evstudy` result with one lead at least.
`mbar`	The values of $\bar{M}$ (default 0, 0.5, 1, 1.5, 2).
`target`	`"average"` (the mean of the post-period effects, default) or an event time.
`draws`	Normal draws for the interval (default 5000).
`seed`	Optional seed for the bootstrap draws.
`conflvl`	Confidence level of the robust interval, a number strictly between 0 and 1 (default 0.95: the 2.5\ drawn bounds).

Details

Let δ_e be the bias of the event-time coefficient β_e, with $\delta_{-1} = 0$ at the reference period and the pre-period biases identified by the leads. The relative-magnitudes restriction is $|\delta_{e+1} - \delta_e| \leq \bar{M}D$, with $D = \max_{s<0} |\delta_{s+1} - \delta_s|$ the largest change between consecutive pre-periods. The bias at post-period $e \geq 0$ is then at most $(e+1)\bar{M}D$ in absolute value, so the identified set for the effect τ_e is $\beta_e \pm (e+1)\bar{M}D$, and for the average post-period effect the average of those. The interval draws (β^*) from the normal distribution of the estimates (with the event study's cluster-robust covariance), computes the bounds for each, and takes the $\alpha/2$ quantile of the lower bound and the $1 - \alpha/2$ quantile of the upper: by Bonferroni it covers the identified set, and so the effect, with probability at least `conflvl`. It is conservative; Rambachan and Roth's conditional and hybrid intervals (the `HonestDiD` package) are shorter.

Value

A data frame of class `rmbl_sensitivity`: `mbar`, `lowerbnd`, `upbound` (the identified set at the estimates), `lower`, `upper` (the interval); attribute `breakdown` (the smallest $\bar{M}$ whose interval includes 0, found by bisection; `Inf` if none up to 10).

References

Rambachan A, Roth J (2023). A more credible approach to parallel trends. Review of Economic Studies 90(5):2555-2591. DOI 10.1093/restud/rdad018

Examples

```
p <- polex()
es <- evstudy(p, "district", "month", "crmrate", "refmonth",
                  leads = 3, lags = 4)
tse(es, seed = 1)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
es <- evstudy(own, "town", "year", "rate", "first", leads = 3, lags = 3)
# a 90 percent robust interval
tse(es, seed = 1, conflvl = 0.9)
```

`tsstmp` *Get a Public Timestamp for a Commitment*

Description

Asks an RFC 3161 time-stamping authority to sign the SHA-256 of `data` together with the current time, then checks the token it returns with `tsvrf()`. Anyone holding the same bytes, the token and the authority's certificate can later prove the bytes existed at that time, without trusting the person who asked.

Usage

```
tsstmp(
  data,
  tsa = "https://freetsa.org/tsr",
  file = NULL,
  timeout = 60,
  trust = NULL
)
```

Arguments

`data`	What to timestamp: a raw vector, the path of an existing file, or a list from `ralloc()` (its commitment is stamped).
`tsa`	The time-stamping authority's URL.
`file`	Optional path to save the token to (a `.tsr` file). An existing file is not overwritten.
`timeout`	Seconds to wait for the authority.
`trust`	Trust anchors for the authority's certificate, passed to `tsvrf()`: for the default authority, its root certificate from `https://freetsa.org/files/cacert.pem`. Without one the token's signature is still checked, but the result is not VERIFIED, because nothing vouches for the certificate that made it.

Details

Use it on anything that has to be fixed before the data are seen: the commitment of a randomisation list from `ralloc()` (pass the list itself), the file of declared outcomes, a manifest or a capsule. Only the 32-byte SHA-256 leaves the machine; the data never do.

The request carries a random 8-byte nonce from the operating system's generator and asks for the authority's certificate in the reply, so the token can be checked offline later. The default authority, `"https://freetsa.org/tsr"`, is free and needs no account; any RFC 3161 service can be named instead, and an institution's own is better where there is one. Pass `cert` and `trust` to `tsvrf()` when re-checking a saved token to tie it to a certificate you trust.

Value

The tsvrf() result for the new token (class bricklayer_timestamp, printed as a box with the time it asserts), with three more elements: token (the raw token), tsa and file (the path it was saved to, or NA). An authority that does not answer is an error.

References

Adams, C., Cain, P., Pinkas, D., and Zuccherato, R. (2001). Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP). RFC 3161.

See Also

tsvrf() to check a saved token, ralloc(), prrgdecl().

Examples

```
# Fix a trial's allocation before enrolment, then stamp its commitment.
a <- ralloc(40, seed = "a demonstration only")
ts <- tsstmp(a, file = file.path(tempdir(), "allocation.tsr"))
ts$time

# Years later, anyone re-checks the saved token against the commitment.
tsvrf(ts$file, charToRaw(attr(a, "alloc")$digest))
```

---

tsvrf *Verify an RFC 3161 timestamp token*

Description

A timestamp token is a small file, signed by an independent timestamping authority, saying that a given set of bytes (for example a manifest of results) existed at a given time. tsvrf() checks that a token covers the bytes given, reports the time it asserts, and verifies the timestamping authority's signature under a certificate you supply or the one the token carries. tsinfo() reads the token's fields without verifying anything. Use them to prove that an analysis output existed by a certain date.

Usage

```
tsvrf(
  token,
  data,
  cert = NULL,
  trust = NULL,
  crls = list(),
  attm = NULL,
  nonce = NULL
)
```

```
tsinfo(token)
```

Arguments

token	The token: a raw vector, or a path to a `.tsr` / `.tst` file. A full `TimeStampResp` or a bare `TimeStampToken` are both accepted.
data	The bytes the token should cover: a raw vector, or a path to a file.
cert	The authority's certificate, DER or PEM, as raw or a path. Optional: when the token embeds a certificate, that one is used and the fact is reported.
trust	Trust anchors for validating that certificate – see `certchnvrf()`. Without them the signature is still checked, but nothing vouches for the key that made it.
crls	Optional CRLs to check the chain against, as raw vectors or paths.
attm	The time to check certificate validity at. Defaults to the time the token asserts, which is usually what is wanted.
nonce	The raw nonce the request sent, or `NULL` (the default) when it is not known. Given, the token must carry it: a token answering another request is refused.

Details

Every step is recorded as a row of `checks`:

- `token_parses`;
- `token_structure`;
- `message_imprint` (the hash of `data` equals the token's imprint);
- `certificate_source`;
- `signature`;
- `attribute_digest` (the signed attributes commit to the token's contents);
- `certificate_trust`;
- with `trust`, the `chain:` checks of `certchnvrf()`.

`ok` is `TRUE` only when every row passes. A token that cannot be parsed is not an error: it gives `ok = FALSE` and the rows up to the failure.

Pass `trust` and the certificate is validated too: the chain is built to an anchor you name, every signature in it is verified, every validity window is checked, an issuer must be a CA, and the leaf must carry the timeStamping extended key usage. Omit `trust` and the signature is still checked but `certificate_trust` is reported as failed, because without an anchor a passing signature says only that the key in the certificate signed the token – not that anyone should believe that certificate.

Validity windows are checked at the time the TOKEN asserts, unless `attm` says otherwise. A token signed in 2020 under a certificate that expired in 2021 was validly signed, and judging it by today's date would reject it for a reason unconnected to its validity.

RSA and ECDSA over the NIST prime curves P-256, P-384 and P-521 are verified. Anything else – a post-quantum signature, a compressed EC point, an Edwards curve – is reported as unverifiable rather than treated as valid, which is the safe direction for a verifier.

Revocation data are not fetched: a CRL has to be handed in through `crls`, and nothing is retrieved over the network (chain checks such as name constraints and certificate policies are done by `certchnvrf()`). Nothing is written to disk.

Errors: `token`, `data` or `cert` neither a raw vector nor the path of an existing file. `tsinfo()` errors on a token it cannot parse.

Value

`tsvrf()` returns a list of class `bricklayer_timestamp`: `ok` (logical), `time` (a `POSIXct` in UTC, or `NA` when the token could not be read), `serial` and `policy` (character), `hash_algorithm` (for example `"sha256"`), `signature_algorithm` (an OID string), and a `checks` data frame with columns `check`, `ok` and `detail`. The print method lists the checks. `tsinfo()` returns a plain list with `time`, `serial`, `policy`, `hash_algorithm`, `imprint` (hex), `signature_algorithm` and `has_certificate` (logical).

References

Adams, C., Cain, P., Pinkas, D., and Zuccherato, R. (2001). Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP). RFC 3161.

See Also

`chnseal()` for the ordering a timestamp cannot give, `cpsatst()` for authorship.

Examples

```
# Offline: bytes that are not a timestamp token are reported, not
# thrown, so a batch check can carry on.
bad <- tsvrf(as.raw(c(0x30, 0x03, 0x02, 0x01, 0x05)),
                        data = charToRaw("manifest"))
bad$ok
bad$checks

# tsinfo() does refuse an unreadable token
try(tsinfo(as.raw(1:10)))

# Real tokens come from a timestamping authority; this is the shape of
# the call.


res <- tsvrf("response.tsr", data = "manifest.json")
res$ok
res$time
```

`usecpstmpl` *Start a capsule from a template*

Description

Creates a new folder with the three files a reproducible analysis `"capsule"` needs, ready to be filled in: a `data_provenance.json` that says where the data come from, an `analysis.R` script that fetches, checks, analyses and records, and a `README.md` that says how to run it. Use it to start a new project in the package's fetch-verify-analyse-record pattern instead of writing that scaffolding by hand.

Usage

```
usecpstmpl(path, name = basename(path), ex = FALSE)
```

Arguments

`path`	Character string: the directory to create. It may exist only if it is empty.
`name`	Character string: the capsule name written into the provenance and README. Defaults to the last part of `path`.
`ex`	Logical, default `FALSE`. `TRUE` uses the shipped OTIS table as the source so the skeleton runs immediately; `FALSE` leaves a `TODO` URL.

Details

The files:

`data_provenance.json` the fields `loadprov` and `vrfcps` expect: `name`, `source` (`url`, `publisher`, `licence`, `retrieved`, `sha256`) and `table` (`value`, `period`, `by`, `rounding`, `suplim`). Every field that must be filled in is marked `TODO`.

`analysis.R` reads the provenance, downloads the source with `fdl` (a local path is read directly), computes its SHA-256 with `sha256file` and stops if it no longer matches the pinned value, runs `anltab`, and writes `manifest.json`, `report.html` and `report.md` in `results/`.

`README.md` the three steps to fill in and run the capsule.

With `example = TRUE` the source is the OTIS table `otis_a01_individuals.csv` shipped in the package's `extdata` folder, so `Rscript analysis.R` runs as written. The table settings in the template (`value = "individuals"`, `period = "year"`, `by = c("table", "group")`) fit that example and must be edited for other data.

Side effects: creates `path` (and its parents) and writes the three files. Errors: `path` exists and is not empty; `name` is not a single non-empty string. The `name` and source are written as escaped JSON strings, so quotes and backslashes are safe.

Value

The directory path, invisibly.

See Also

loadprov, vrfcps, anltab.

Examples

```
d <- usecpstmpl(tempfile("capsule-"), ex = TRUE)
list.files(d)

# the provenance file to fill in
cat(readLines(file.path(d, "data_provenance.json")), sep = "\n")

# a blank template with a chosen name
d2 <- usecpstmpl(tempfile("capsule-"), name = "segregation-2024")
grep("TODO", readLines(file.path(d2, "data_provenance.json")), value = TRUE)

# an existing, non-empty folder is never overwritten
try(usecpstmpl(d))
```

---

valrules *Apply declared rules to a data frame*

Description

Evaluates each rule from rul() on a data frame and returns the failures in the same shape valsch() uses, so the two can be combined and handed to schval() together. Use it each time a data set is refreshed, to see which of its own invariants still hold.

Usage

```
valrules(data, rules)
```

Arguments

data	A data frame the rules are applied to.
rules	A list of rul() objects, or a single rule.

Details

A rule whose column is absent is SKIPPED rather than failed – a missing column is valsch()'s business, and reporting it twice buries the real finding. A rule that ERRORS is reported as a failure naming the error, never swallowed: a rule that cannot run has not passed. On a table of no rows a row-wise rule has no row to fail and passes.

How a rule's result is read: a result that is not logical, or has length 0, is a failure ("rule returned <class>, not a logical"); a single value fails unless it is TRUE (so NA fails); a longer vector is read row by row, and rows that are FALSE or NA fail (a rule that cannot decide has not been satisfied). The length of a row-wise result is not checked against the number of rows. The issue is named by the

rule's name; two rules with the same name keep only the later failure. Rules run in the order given and only read the data.

Errors: data not a data frame (or with missing or duplicated column names); rules not a rule or a list; an element that is not a rule.

Value

A named list of issues, one per failed rule, each a list with severity (character) and message (character; for a row-wise rule it ends with "(k of n rows)"), and for a row-wise rule n_failed (integer) and rows (integer, the first ten failing row indices). Empty when everything passes.

See Also

rul(), valsch(), schval()

Examples

```
df <- data.frame(id = c(1, 2, 2), age = c(30, -5, 40),
                 start = c(1, 5, 3), end = c(2, 4, 9))

rules <- list(
  rul("agenonnegative", function(v) v >= 0, col = "age"),
  rul("idunique", function(v) !anyDuplicated(v), col = "id",
      sev = "fatal"),
  rul("datesordered", function(d) all(d$start <= d$end))
)

issues <- valrules(df, rules)
names(issues)

# A row-wise failure reports how many rows and which.
issues$agenonnegative$n_failed
issues$agenonnegative$rows

# Clean data produces nothing.
clean <- data.frame(id = 1:3, age = c(30, 31, 40), start = 1:3, end = 4:6)
length(valrules(clean, rules))

# A rule for an absent column is skipped, not failed -- a missing
# column is valsch()'s finding to report, not this one's.
length(valrules(data.frame(id = 1:3), rules[1:2]))

# A rule that errors is a failure, not a silent pass.
broken <- list(rul("bad", function(v) stop("boom"), col = "age"))
valrules(clean, broken)$bad$message

# NA counts as a failure
valrules(data.frame(age = c(30, NA)), rules[1])$agenonnegative$message

# only rule() objects are accepted
try(valrules(df, list(function(v) v >= 0)))
```

valsch *Validate a Data Frame Against a Provenance Schema*

Description

Checks a data frame against the schema block of a provenance record: required columns, row-count bounds, column types, numeric ranges, missingness and allowed category values. It returns the problems it finds instead of stopping, so the caller can log them, show them, or pass the same schema to schval(), the wrapper that turns them into errors and warnings. Use it each time a file is re-downloaded, to catch a re-release that renamed, retyped or recoded a column.

Usage

```
valsch(dfraw, prov)
```

Arguments

dfraw	The data frame to validate.
prov	A provenance list as returned by loadprov(), or list(schema = infsch(df)). The schema block may contain:

- expected_columns – required column names (character). A missing one is "fatal"; everything else below is a "warning".
- structural_invariants – a list with min_data_rows and max_data_rows (numbers of rows).
- expected_value_sets – a named list of allowed values per column.
- expected_types – a named character vector of expected classes. integer, numeric and double are treated as interchangeable, since a re-release legitimately widens one to another.
- numeric_ranges – a named list of c(min =, max =) bounds.
- max_missing_fraction – a named numeric of per-column ceilings on the share of NA, between 0 and 1.

infsch() produces all six from data you already trust.

Details

Every schema field is optional and is checked only when present, so a schema written for an earlier version keeps working unchanged.

This is the STRUCTURAL check – names, types, bounds, value sets. It cannot tell you that a column kept its name, type and range while its distribution moved; cpsdrft() answers that. Rules that are particular to a data set (an age must be non-negative, two dates must be ordered) belong in rul() and valrules().

The checks, and the name of the issue each one adds:

- a missing expected_columns entry: missing_columns, "fatal";
- fewer than min_data_rows rows: row_count_low;

- more than max_data_rows rows: row_count_high;
- a column whose class differs from expected_types: type_<col>. integer, numeric and double are one family, and so are character and factor;
- non-missing values outside numeric_ranges (inclusive bounds; non-numeric columns are skipped): range_<col>;
- a share of NA above max_missing_fraction: missing_<col>;
- values not in expected_value_sets: unexpected_<col>.

A column that a check names but the data lack is skipped by that check. Only the expected_columns check reports such a column. All of these are "warning" except missing_columns.

The schema itself is checked too. An empty schema (list()) gives one schema_empty warning. An unknown field name within three edits of a known one (expected_colums) is "fatal", because it would otherwise be ignored silently; any other unknown name is a warning. An unknown structural invariant one edit, or one swap of neighbouring letters, away from min_data_rows or max_data_rows is fatal; other invariant names are left alone. expected_dtypes is accepted as another name for expected_types, and synthetic_recipe is a known field that is not checked here.

Errors: dfraw that is not a data frame; a schema that is not a list. A prov that is NULL or has no schema returns an empty list (use schval() to make that an error).

Value

A named list of issues; each issue is a list with severity ("fatal" or "warning") and a human-readable message. The names are those listed in Details. A zero-length list means the data frame is clean.

See Also

infsch() to derive a schema, cpsdrft() for the distributional check, schval() to raise on the issues.

Examples

```
prov <- list(schema = list(
  expected_columns      = c("id", "year"),
  structural_invariants = list(min_data_rows = 1),
  expected_value_sets   = list(year = 2020:2025)
))
df <- data.frame(id = 1:3, year = c(2020, 2021, 2030))
issues <- valsch(df, prov)
names(issues) # flags the out-of-set year value

# A column that silently changed type is caught.
typed <- list(schema = list(expected_types = c(id = "integer")))
valsch(data.frame(id = c("1", "2")), typed)[[1]]$message

# So is a value outside its pinned range, and excess missingness.
ranged <- list(schema = list(numeric_ranges = list(v = c(min = 0, max = 1))))
valsch(data.frame(v = c(0.5, 9)), ranged)[[1]]$message
```

```
gappy <- list(schema = list(max_missing_fraction = c(v = 0.1)))
valsch(data.frame(v = c(1, NA, NA, 4)), gappy)[[1]]$message

# A clean frame produces nothing.
length(valsch(data.frame(id = 1:3, year = 2021), prov))

# a misspelt schema field is fatal rather than silently ignored
valsch(df, list(schema = list(expcol = "id")))[[1]]

# a schema inferred from trusted data, checked against a new release
sch <- infsch(data.frame(id = 1:3, year = 2020:2022))
names(valsch(data.frame(id = 1:2), list(schema = sch)))

# the data must be a data frame
try(valsch(list(id = 1), prov))
```

`vrfcps` *Re-Verify an Entire Reproducible Data Capsule*

Description

Runs the full custody chain over a capsule directory in one call and reports, check by check, whether the capsule is still the one the analysis was run on: the provenance record is readable, the pinned data file is present and has its recorded SHA-256, the schema still validates, a recorded analysis script still has its pinned hash, every numeric cross-check in a results manifest still reproduces its recorded verdict, and the data are real rather than synthetic. Use it before re-running or publishing an analysis, or when receiving a capsule from someone else.

Usage

```
vrfcps(
  cpsdir,
  provfile = "data_provenance.json",
  dfile = NULL,
  mnffile = NULL,
  scrfile = NULL
)
```

Arguments

`cpsdir`	Directory containing the capsule, a single string.
`provfile`	Provenance JSON filename, relative to `cpsdir`. `"data_provenance.json"` is the default.
`dfile`	Data filename relative to `cpsdir`. `NULL` (the default) uses the provenance's `resource$filename`.

`mnffile`	Optional results-manifest JSON (as written by `wrmnfjs()`), relative to `cpsdir`. `NULL` (the default) uses `manifest.json` when the capsule has one.
`scrfile`	Optional analysis-script filename relative to `cpsdir`; compared against the pinned script digest. `NULL` (the default) uses the script named in the provenance or the manifest, if any.

Details

The checks, in the order they appear in `checks`:

`provenance_readable` `provfile` is a plain relative path inside the capsule (no `..`, no absolute path, no symlink out) and parses as JSON.

`data_present` the data file (`dfile`, else the provenance's `resource$filename`) is inside the capsule and is a file.

`data_sha256` `vrfsha256()` finds that its SHA-256 equals `resource$sha256` (64 hex characters).

`data_size_bytes` when `resource$size_bytes` is recorded: the size on disk matches.

`data_readable` a `.csv` file is read with `utils::read.csv()`; a non-CSV file fails this row when a row count or schema is recorded (they cannot be checked).

`data_row_count` when `resource$row_count_data_rows` is recorded: the CSV has that many rows.

`schema_valid` when a `schema` is recorded: `valsch()` finds no fatal issue (warnings are listed in `detail` but pass).

`manifest_consistent` when `mnffile` is given, or the capsule holds `manifest.json`: every result with numeric `observed` and `expected`, a `tol` and a status of `PASS` or `DIFFER` still gets that status from $|observed - expected| \leq tol$.

`script_sha256` when a script is named (by `scrfile`, by `script$filename` in the provenance or by `meta$scrfile` in the manifest) or a script digest is pinned (`meta$script_sha256` or `script$sha256`): the script's SHA-256 matches the pin.

`data_not_synthetic` no `<data>.synthetic` sidecar (written by `mksyncsv()`), and neither the manifest nor the provenance records `synthetic = true`.

A check that cannot be made because the provenance lacks a field is a FAILED row, never skipped: a provenance of `{}` verifies nothing. `ok` is `TRUE` only when every row passed and every required check was made. The required checks are `provenance_readable`, `data_present`, `data_sha256` and `data_not_synthetic`, plus `manifest_consistent` and `script_sha256` when a manifest or a script is in play.

Entirely offline: nothing is downloaded and nothing is written. Error: `cpsdir` that is not one existing directory. Every other problem is a failed row, not an error.

Value

An object of class `c("bricklayer_capsule_check", "list")`, printed as one line per check:

`ok` logical scalar, the overall verdict described in Details.

`checks` data.frame with columns `check` (character, the check name), `ok` (logical) and `detail` (character: the path checked, the digest found, or why it failed).

`required` character, the checks that had to be made.

`missing` character, required checks that were never made (`character(0)` when none).

Examples

```
dir <- file.path(tempdir(), "capsule-example")
dir.create(dir, showWarnings = FALSE)
write.csv(data.frame(id = 1:3), file.path(dir, "d.csv"), row.names = FALSE)
prov <- list(resource = list(filename = "d.csv",
                             sha256 = sha256file(file.path(dir, "d.csv")),
                             row_count_data_rows = 3),
             schema = list(expected_columns = "id"))
writeLines(tojs(prov, auto_unbox = TRUE),
           file.path(dir, "data_provenance.json"))
vrfcps(dir)$ok
vrfcps(dir)

# an edited data file fails its digest and its row count
write.csv(data.frame(id = 1:4), file.path(dir, "d.csv"), row.names = FALSE)
chk <- vrfcps(dir)
chk$ok
chk$checks[!chk$checks$ok, c("check", "detail")]

# a path out of the capsule is a failed check, not a read
vrfcps(dir, provfile = "../elsewhere.json")$ok

# a missing directory is an error
try(vrfcps(file.path(tempdir(), "no-such-capsule")))
unlink(dir, recursive = TRUE)
```

---

vrfmrg *Verify recoded category counts against the counts a release published*

Description

Compares the number of rows per category after a recode with the counts the data source published for those categories (for example the frequency table in a codebook or a statistical release). If they disagree, it tries every reordering of the labels and names the one under which the observed counts would match, which is the signature of labels attached to the wrong groups. Use it after decoding or recoding a column whenever published totals exist.

Usage

```
vrfmrg(x, pub, tol = 0, strict = TRUE)
```

Arguments

x	A character or factor vector after recoding.
pub	Named numeric vector, label = published count.
tol	Non-negative number, default `0`. The absolute difference in counts allowed per label, for releases that round or suppress small counts.

`strict` Logical, default `TRUE`: a mismatch is an error, so a pipeline stops. `FALSE` returns the result with `ok = FALSE`, the `permutation` that would explain the counts and the `message` the error would have carried, for callers that want to report it or pass the permutation to `rlblfrns`.

Details

For each label l in `names(published)` the observed count is $n_l = \sum_i 1[x_i = l]$ (missing values excluded). The counts match when $|n_l - p_l| \leq$ `tol` for every label.

On a mismatch, and when there are at most 7 labels, the function searches the $k!$ permutations of the labels (5040 at $k = 7$) for the first non-identity one under which the observed counts match the published counts, and names it in the message. With more than 7 labels no search is done and `permutation` is `NULL`. Several permutations can match when counts are tied; only the first is reported.

A label that occurs in `x` but not in `pub` is always a failure (no permutation search). `pub` must be a named numeric vector, otherwise an error.

Value

A list of class `rmbl_margin_check` (it prints as a box) with elements `counts` (named numeric, observed count per published label), `published` (named numeric), `ok` (logical), `permutation` (named character, true label = label it would have to be read as, or `NULL`) and `message` (character, or `NULL` when `ok`). It is returned invisibly when the counts match and visibly when `strict = FALSE` and they do not. With `strict = TRUE` a mismatch is an error instead.

See Also

`xfrvrf`, `rlblfrns`.

Examples

```
x <- c("White", "White", "Black", "Indigenous", "White", "Black")
vrfmrg(x, c(White = 3, Black = 2, Indigenous = 1))$ok

# swapped labels: the permutation that explains the counts is named
r <- vrfmrg(x, c(White = 2, Black = 3, Indigenous = 1),
                      strict = FALSE)
r$permutation

# a release that rounds counts to the nearest 5
vrfmrg(rep(c("A", "B"), c(48, 21)), c(A = 50, B = 20),
                 tol = 2)$ok

# the default is to stop
try(vrfmrg(x, c(White = 2, Black = 3, Indigenous = 1)))
```

vrfpol *Run the pollution-to-health pipeline and report it*

Description

Resolves the exposure (synthetic demo data, a CSV with an `exposure` column and an optional `income` column, a NAPS-style pull with a `value` column and a `unit`, or scalar arguments), logs the assumptions, and when they all hold runs the concentration-response, attributable-fraction, displaced-mortality, burden and equity stages. `polrpttxt()` renders the result as plain text. It is the one-call version of the pollution functions, made for scripts and the command line: it never stops with an R error for bad input, but returns a report whose status says what went wrong.

Usage

```
vrfpol(
  poll,
  y = "allcause",
  reg = NULL,
  years = NULL,
  demo = FALSE,
  expcsv = NULL,
  expomean = NULL,
  expoprev = NULL,
  ref = NULL,
  brate = 500,
  basrateper = 1e+05,
  pop = 1e+06,
  rare = FALSE,
  hirate = FALSE
)

polrpttxt(report)
```

Arguments

`poll`	`"no2"` or `"pm25"` (also `"PM2.5"`), in either case.
`y`	Outcome key for the concentration-response function; the mortality outcomes count deaths, `"chiast"` incident cases.
`reg`, `years`	Labels for the report.
`demo`	Use synthetic demo data (`TRUE` or `FALSE`).
`expcsv`	Path of a CSV with an `exposure` column in micrograms per cubic metre, or a NAPS pull with a `value` column and a `unit` column (required with `value`): each row is converted by its own unit, NO2 in ppb at 1.88 micrograms per cubic metre per ppb (the WHO 2021 conversion at 25 C and 1 atm), and any other unit is an error. With a CSV or the demo data the exposed are the rows above `ref`: `expoprev` is their share and `expomean` their mean, so Levin's formula is not diluted twice.

`expomean`, `expoprev`
: Scalar inputs used when neither `demo` nor `expcsv` is given: the mean exposure among the exposed and the share of the population exposed. A wrapper must never default one of them to zero: a prevalence of zero silently zeroes the burden, which is why the two are required together.

`ref`
: Counterfactual reference concentration; `NULL` takes the pollutant's default (PM2.5 5.8, NO2 10; see `crfpm25`).

`brate`
: Baseline outcome rate per 100,000 per year (unlike `polbrdn`, which takes a rate per person-year).

`basrateper`
: The denominator `brate` is quoted per: 100,000 (the default), or 1 for a rate per person-year such as a crude death rate of 0.012.

`pop`
: Population at risk (persons).

`rare`
: `TRUE` when the outcome really is rarer than one case per 100,000 per year. Without it a rate that converts to less than that is an assumption failure, because it is also what a rate per person-year passed as per 100,000 looks like (0.008 per person-year is 800 per 100,000; as 0.008 per 100,000 the burden is 100,000 times too small). The check is on the converted rate, so it holds whatever `basrateper` is.

`hirate`
: `TRUE` when the outcome really is more common than 10,000 per 100,000 a year (no population's all-cause death rate is); without it such a rate is an assumption failure.

`report`
: A result of `vrfpol()` (or a list with the same parts); anything else is refused with an R error.

Details

The assumption log checks that the exposure mean is finite and non-negative and at most 10,000 micrograms per cubic metre, the prevalence is in $[0, 1]$, the baseline rate (converted to per 100,000) is zero or between 1 and 10,000 per 100,000 a year (from 1e-6 with `rare`, up to 100,000 with `hirate`), the population is positive and at most 10 billion, the reference is finite and non-negative, and the pollutant has a CRF. Any failure gives status `"assumption_failure"` and skips the stages.

With data (a CSV or the demo), the `"exposed"` are the rows above the reference: their mean is `expomean` and their share `expoprev`. Empty cells and missing-value tokens (`NA`, `n/a`, `null`, `-` ...) are dropped with a message; text that is not a number (a detection-limit flag such as `"<2"`) and negative values stop the run with an error report. A gzip, bzip2 or xz file is decompressed; UTF-16 and Latin-1 files are decoded. An `income` column, when present and numeric, gives the equity stage (`expconcidx`).

The stages: `crfpm25` or `crfno2`; `attrfrac` with the CRF's interval; `mortdisp` for removing the excess exposure over the exposed population, with $\beta = \log(RR)/(\bar{x} - x_{ref})$; and `polbrdn`. The demo data are 1,000 synthetic units with lognormal exposures at the scale of Toronto NO2 or PM2.5 levels and five income groups, drawn with a fixed seed (R's own random stream is restored).

Value

`vrfpol()`: the report as a list of class `rmbl_pollution_report` (`plot()` draws it); its `status` is `"ok"`, `"assumption_failure"` or `"error"`, and the attribute `exitstat` carries a command-line exit code (0, 1 or 2). An argument the pipeline cannot use (no exposure data at all, a vector where a

single number is expected, an unknown outcome, a missing reference, a directory as expcsv) is an "error" report with exit status 2, never a raw R error; an exposure at or below the reference is a valid run with zero attributable cases, and the scalar inputs expomean and expoprev go together (one without the other is an error report). On "ok" the report also holds inputs (the resolved exposure mean and prevalence, the rate per 100,000, the population, the reference and the rows dropped), assumptions (a list of assumption, ok, note) and pipeline (crf, paf, pafci, displaced, burden and equity, which is NULL without income data). polrpttxt(): a single character string with embedded newlines, ending in a STATUS: line (or "ERROR:" and the message for an error report).

See Also

polbrdn and the other stage functions to run one step on its own.

Examples

```
r <- vrfpol("no2", demo = TRUE)
r$status
r$pipeline$paf
cat(polrpttxt(r))

# Scalar inputs: a city of 2.8 million at a PM2.5 mean of 9.
s <- vrfpol(
  "pm25",
  expomean = 9,
  expoprev = 1,
  brate = 750,
  pop = 2.8e6,
  reg = "Toronto"
)
s$pipeline$burden$attrcases
attr(s, "exitstat")

# A CSV of measurements with an income column.
f <- tempfile(fileext = ".csv")
utils::write.csv(
  data.frame(exposure = c(12, 18, 25, 31, 9), income = 1:5),
  f,
  row.names = FALSE
)
vrfpol("no2", expcsv = f)$pipeline$equity$concidx

# A rate per person-year passed as per 100,000 fails an assumption ...
bad <- vrfpol("no2", expomean = 20, expoprev = 1, brate = 0.008)
bad$status
# ... and an argument it cannot use is an error report, not an R error.
vrfpol("no2")$error
```

vrfrcd *Prove a recode with a before/after cross-tabulation*

Description

Checks, after the fact, that a recode did what you said it did. It cross-tabulates the original values against the recoded ones and stops with an error unless each original category went to exactly one new category, that category is the one in your declared mapping, no rows were lost or added, and no value became missing (or was filled in). Use it directly after any recode, including recodes done by other code or other software, to catch swapped groups before a model runs.

Usage

```
vrfrcd(orig, recd, decl)
```

Arguments

`orig`, `recd`	Atomic vectors of the same length: the values before and after the recode, row for row.
`decl`	Named character vector, old = new: the mapping the analyst says was applied. Old values not listed are expected to be unchanged.

Details

Factors are compared through their labels. The checks, in order:

1. `orig` and `recd` are atomic and the same length;
2. `is.na(original)` equals `is.na(recoded)` element by element;
3. in the table of non-missing pairs, no original value has more than one recoded value (the recode is a function of the label);
4. each original value's recoded value equals `declared[old]`; an original value that is not a name of `decl` must be unchanged (identity).

The first failed check is an error naming the offending categories.

Value

Invisibly, when every check passes, the cross-tabulation as a data frame with columns `original`, `recoded` (factors) and `Freq` (integer), including the zero cells of the full table. Otherwise an error.

See Also

`grdrcd`, `rcdmnf`.

Examples

```
before <- c("W", "B", "W", NA)
after <- c("White", "Black", "White", NA)
tab <- vrfrcd(before, after, c(W = "White", B = "Black"))
tab[tab$Freq > 0, ]

# a value not in the declared mapping must be unchanged
vrfrcd(c("W", "X"), c("White", "X"), c(W = "White"))

# swapped groups are caught
try(vrfrcd(c("W", "B"), c("Black", "White"), c(W = "White", B = "Black")))

# a value that became NA is caught
try(vrfrcd(c("W", "B"), c("White", NA), c(W = "White", B = "Black")))
```

`vrfsha256` *Verify a File's SHA256 Against an Expected Digest*

Description

Computes the SHA-256 digest of a file and compares it with the digest pinned in a provenance record, so a pipeline can tell whether the file it downloaded is byte for byte the one the analysis was run on. A changed file (a silent re-release, a truncated download, an edited copy) gives a different digest.

Usage

```
vrfsha256(path, expsha)
```

Arguments

`path`	Path to the file to hash, a single string.
`expsha`	The expected SHA-256 digest: a hex string (64 characters, any case), or a length-1 list holding one, as read from provenance JSON.

Details

The digest is computed by `sha256file()`, the package's own compiled SHA-256 core (FIPS 180-4), reading the file in chunks. The expected value is normalised before the comparison: a length-1 list (as `loadprov()` returns a pinned digest) is unlisted, surrounding spaces are trimmed and letters are lower-cased, so an upper-case digest copied from a web page still matches. The function reports and does not stop on a mismatch; the caller decides.

Errors: `path` that does not name a readable file; `expsha` that is not exactly one non-NA value.

Value

A list of three elements: `actual` (character, the computed digest, 64 lower-case hex characters), `expected` (character, the normalised expected value) and `match` (logical, `TRUE` when they are identical).

Examples

```
f <- tempfile()
writeLines("hello capsule", f)

# Matching digest -> match TRUE.
chk <- vrfsha256(f, sha256file(f))
chk$match # TRUE

# A wrong expected digest -> match FALSE, with both values reported.
bad <- vrfsha256(f, strrep("0", 64L))
bad$match # FALSE
bad$actual # the real digest

# case and a list wrapper (as from provenance JSON) do not matter
vrfsha256(f, list(toupper(sha256file(f))))$match

# a missing file, or more than one expected digest, is an error
try(vrfsha256(file.path(tempdir(), "missing.csv"), strrep("0", 64L)))
try(vrfsha256(f, c("a", "b")))
```

`wbrtry` *Retry the failures of a bulk Wayback run*

Description

Picks up where a `wbsave` run left off: it re-tries only the URLs whose captures failed or were still running, after first checking whether the Wayback Machine already holds a copy of each. Use it after a run that hit archive.org's limits, typically the next day.

Usage

```
wbrtry(output, incgone = FALSE, lookup = TRUE, ...)
```

Arguments

`output`	The JSON Lines results file of an earlier `wbsave()` run (or of `rmbl wayback`); results of the retry are appended to it.
`incgone`	Logical, default `FALSE`. Also capture again the pages whose failure says the page is gone.
`lookup`	Logical, default `TRUE`. Look each failure up in the Wayback Machine (one lookup a second) before capturing again.
`...`	Passed to `wbsave()`: `jobs`, `within`, the keys, `quiet`, `permin`, `maxcap`.

Details

Reads the results file a wbsave run wrote and tries again every URL whose latest result is "failed" (or "submitted": a capture a stopped run had started, which is polled rather than submitted again), so nobody has to dig the failures out by hand. Each failed URL is first looked up in the Wayback Machine, one lookup a second: a capture made at any time (an earlier one of a page that has since gone, or one made today by another run when the daily limit refused this one) is recorded as "existing" without capturing again. What is left is captured again with wbsave(), appending to the same file. A page its source no longer has is not sent again unless incgone = TRUE: a new capture would only archive the error. "Gone" means the recorded message contains archive.org's report of the target server's reply ("target server cannot find", "returned 404", "returned 410", "error:not-found" or "error:gone"); an HTTP 404 from archive.org's own API does not count. A failure for a daily limit recorded today (UTC) is not sent again until tomorrow.

Network: the lookups use the Wayback availability API and the new captures use wbsave(), which needs the archive.org keys. When there is nothing left to send, no keys and no network are needed. Each lookup that finds a capture is appended to output as an "existing" line.

Errors: output not a single path to an existing file; incgone or lookup not TRUE or FALSE; any error from wbsave().

Value

The whole run as it now stands, one row per URL (a data frame of class rmbl_wayback_save), with the attribute retry: how many failures there were, how many a lookup found, how many were captured again, how many were left as gone, and how many were left for tomorrow (a daily limit hit today), as a one-row data frame with integer columns failed, found_by_lookup, captured_again, left_as_gone and left_for_tomorrow.

See Also

wbsave; rmbl wayback --retry FILE.jsonl from the command line.

Examples

```
# Offline: a results file whose only failure is a page that is gone
# has nothing to send again, so no keys or network are used.
f <- tempfile(fileext = ".jsonl")
writeLines(
  c(
    paste0(
      '{"url":"https://example.org/","status":"saved",',
      '"timestamp":"20260101120000",',
      '"snapshot":"https://web.archive.org/web/20260101120000/https://',
      'example.org/",',
      '"message":"","at":"2026-01-01T12:00:00Z"}'
    ),
    paste0(
      '{"url":"https://example.org/old","status":"failed",',
      '"message":"target server cannot find the ',
      'page","at":"2026-01-01T12:00:00Z"}'
    )
```

```
  ),
  f
)
res <- wbrtry(f, lookup = FALSE)
res
attr(res, "retry")

try(wbrtry(tempfile(fileext = ".jsonl")))

# With failures to resend, keys and the network are needed.


res <- wbsave(urls, output = "snapshots.jsonl")
# later, or the next day once the daily capture limits have reset:
res <- wbrtry("snapshots.jsonl")
attr(res, "retry")
```

`wbsave` *Archive many pages in the Wayback Machine*

Description

Asks the Internet Archive to take a dated copy of each web page or file in a list, so that the exact source an analysis used (a statistics page, a data download, a policy document) can be cited and checked later even if the original changes or disappears. It needs a free archive.org account and its API keys.

Saves every URL in the Internet Archive's Wayback Machine with Save Page Now 2, the API archive.org offers to account holders, keeping several captures in flight at once while keeping to archive.org's rate limits. A page archived recently is kept as it is (`within`); a page archive.org has already captured today (its daily limit) is recorded with that capture. Each result is written to `output` as it arrives, so an interrupted run started again with the same `output` carries on where it stopped. A live progress bar shows the run while a person is watching.

Usage

```
wbsave(
  urls,
  ak = NULL,
  sk = NULL,
  within = "30d",
  jobs = 4L,
  output = NULL,
  poll = 10,
  maxwait = 600,
  capall = FALSE,
  quiet = NULL,
  permin = 6,
```

```
  maxcap = 1000,
  fallback = NULL,
  reason = NULL
)
```

Arguments

urls	Character vector of http or https URLs. Duplicates, NA and empty strings are dropped. An address on your own network (a private or loopback IP, a .local or one-label name) is refused: the archive cannot reach it, and the request would tell it where your intranet is.
ak, sk	Your archive.org S3-like API keys (archive.org/account/s3.php). By default they are read from the environment variables IA_S3_ACCESS_KEY and IA_S3_SECRET_KEY, or from the files named by IA_S3_ACCESS_KEY_FILE and IA_S3_SECRET_KEY_FILE (a key file other users can read is warned about). They are sent to web.archive.org only and never printed.
within	Keep a capture made within this long instead of making a new one: a number followed by s, m, h or d (default "30d"); "0" always captures.
jobs	Captures in flight at once, 1 to 12. Default 4.
output	Optional path of a JSON Lines file: one line per URL, written as each finishes, and read at the start so a run can be resumed. Its folder must exist; this is checked before anything is sent.
poll	Seconds between status checks of a capture. Default 10.
maxwait	Seconds a capture may take before it is given up as failed. Default 600.
capall	Also capture error pages (4xx and 5xx); default FALSE.
quiet	TRUE for no progress output; the default shows it while a person is watching.
permin	Captures started per minute, at most; default 6 (archive.org allows an account 7).
maxcap	The most captures this run starts; default 1000. The rest are left for the next run (with the same output, it carries on from them).
fallback	"rmorie" saves what the Wayback Machine refused (a server that refuses its crawler, Statistics Canada among them) from this connection to archive.rmorie.com, with arccap; the result is attribute archive. NULL (default) does not.
reason	With fallback = "rmorie": why the files are kept (20 characters or more).

Details

Captures start no faster than permin. When archive.org asks the run to slow down (HTTP 429, or a reply saying the account has too many captures running, inside an HTTP 200 as SPN2 sends it), nothing more is submitted until the wait is over: as long as its Retry-After says, or 15 seconds doubling to 10 minutes; eight such replies in a row stop the run. A reply saying the account's daily cap is reached stops submitting for the day. A server error or a lost connection is retried with back-off, up to six times per URL; status checks that fail back off the same way. A refused key (HTTP 401 or 403) stops the run.

A run of many URLs is slow by design: at the default 6 captures a minute, 1,000 URLs take close to three hours. URLs already finished in `output` are skipped, so a run can be stopped and restarted.

Network: every request goes to web.archive.org; the keys travel in the `Authorization` header and are never written to `output` or printed.

Errors, all raised before anything is sent: no usable URL; a URL with an invisible or control character, or not starting with `http://` or `https://`; missing keys, or keys that are not letters and digits; `within` not of the form `"30d"`; `jobs` outside 1 to 12; `poll` below 1; `maxwait` below `poll`; `permin` outside (0, 60]; `maxcap` below 1; `capall` not `TRUE` or `FALSE`; an `output` whose folder does not exist or that cannot be written.

Value

A data frame of class `rmbl_wayback_save`, one row per URL with a result: `url`, `status` (`"saved"`, a new capture; `"existing"`, a capture already there; `"failed"`; or `"submitted"`, a capture started when the run was stopped, polled again by the next run), `timestamp` (`YYYYMMDDhhmmss`), `snapshot` (the archived copy's address) and `message` (archive.org's reason, for a failure). The attribute `not_sent` lists the URLs this run did not start (`maxcap`, or archive.org's daily cap for the account).

References

Internet Archive. Save Page Now 2 (SPN2) public API.
archive.org/details/spn-2-public-api-page-docs

See Also

`wbsnpnat` to look up an existing capture; `wbrtry`; `arccap` for URLs whose server refuses the Internet Archive (`"cannot find"` although the page is there); `rmbl wayback FILE` runs this from the command line.

Examples

```
# The checks run before any request, so these need no network or keys:
try(wbsave("ftp://example.org/report.pdf"))
try(wbsave("https://example.org/", ak = "abc", sk = "def", jobs = 20))
try(wbsave(
  "https://example.org/",
  ak = "abc",
  sk = "def",
  output = file.path(tempdir(), "no-such-folder", "runs.jsonl")
))

# A real run needs archive.org keys and the network.


Sys.setenv(
  IA_S3_ACCESS_KEY_FILE = "~/.config/ia/s3_access_key.txt",
  IA_S3_SECRET_KEY_FILE = "~/.config/ia/s3_secret_key.txt"
)
res <- wbsave(
  c(
```

```
    "https://www.ontariocourts.ca/ocj/statistics/",
    "https://www150.statcan.gc.ca/n1/tbl/csv/35100177-eng.zip"
  ),
  output = "snapshots.jsonl"
)
res
```

---

wbsnpnat *Resolve a Wayback Machine snapshot URL (C++/libcurl)*

Description

Queries the Internet Archive "available" API for the closest archived snapshot of `url` and returns its address, or `NULL` when nothing is archived. Use it to find an archived copy of a data file or page that has moved or been removed. C++ backend; supersedes the older R-level resolver `wbsnpurl`, which is kept for the pure-R path and now parses with the package's own JSON codec.

Usage

```
wbsnpnat(url, tmout = 30L)
```

Arguments

`url`	URL to resolve, a single string (`http` or `https`).
`tmout`	Request timeout in seconds, default 30.

Details

`url` is only a query parameter of the lookup, never fetched, so a plain `http://` page can be looked up; a local or private address is still refused. One request is made to `https://archive.org/`; the most recent snapshot is asked for (there is no timestamp argument; use `wbsnpurl()` for one). Any network failure, and a URL that was never archived, give `NULL` rather than an error. A spinner is drawn on stderr while waiting, unless `options(morie.quiet = TRUE)`.

Errors: `url` not a single non-empty string; a local or private address.

Value

The `https` snapshot URL (a length-1 character string), or `NULL` if none is archived or the lookup failed.

Examples

```
# Input is validated before any network call:
try(wbsnpnat(""))     # empty url -> error
try(wbsnpnat("http://localhost/report.csv"))
```

```
# needs the network; any network failure returns NULL rather than erroring.
# Closest archived snapshot of a live page (or NULL if none archived).
wbsnpnat("https://www.r-project.org/")

# A shorter timeout for a quick lookup.
wbsnpnat("https://cloud.r-project.org/", tmout = 10)

# A never-archived URL returns NULL rather than erroring.
wbsnpnat("https://example.invalid/never-archived")
```

`wbsnpurl` *Resolve a Wayback Machine snapshot URL*

Description

Asks the Internet Archive which archived copy of a web address is closest to a given time and returns a URL that downloads that copy. This is the fetch failsafe the wider morie package family relies on: callers try the live source first and fall back to the snapshot when the source is down, has moved or has been taken offline. `fdl()` uses it automatically.

Usage

```
wbsnpurl(url, ts = NULL)
```

Arguments

`url`	The original source URL to look up, a single string.
`ts`	Optional target time as a string of digits `YYYYMMDDhhmmss` (or a prefix such as `"2023"` or `"20230115"`); the API returns the snapshot closest to it. `NULL` or `""` (the default) asks for the most recent.

Details

The function sends one request to the availability API, `https://archive.org/wayback/available?url=<url>[&timestamp=<ts>]`, where the timestamp part is sent only when `ts` is given.

The answer's `archived_snapshots$closest` is read. When the API reports an available snapshot its URL is returned, with `http://` rewritten to `https://` and passed through the same public-URL check as every other URL the package fetches. The timestamp is passed to the API as given; the API accepts any prefix of `YYYYMMDDhhmmss` and returns the snapshot nearest to it, before or after.

`NULL` is returned when the API has no snapshot, when the request fails (offline, an HTTP error, invalid JSON) or when the answer has no usable URL. Errors: `url` that is not a single non-empty string or is not a public web address; `ts` that is not `NULL`, `""` or a string of 1 to 14 digits. Nothing is written.

Value

A length-1 character string, the `https` snapshot URL of the form `https://web.archive.org/web/` followed by `<timestamp>/<url>`, or `NULL`.

Examples

```
# argument errors come before any request
try(wbsnpurl(NA))
try(wbsnpurl("https://www.r-project.org/", ts = "2020&x=1"))
try(wbsnpurl(c("https://a.org", "https://b.org")))


# needs the network
wbsnpurl("https://www.r-project.org/")
# the snapshot nearest to January 2020
wbsnpurl("https://www.r-project.org/", ts = "202001")
```

---

wcb *The wild cluster bootstrap for a difference in differences with few clusters*

Description

A p-value and confidence interval for the treatment effect of `plndid` that stay accurate when there are few clusters (a policy assigned to a handful of districts or states), where the cluster-robust t test over-rejects.

Usage

```
wcb(
  data,
  unit,
  time,
  y,
  trtd,
  covs = NULL,
  cl = NULL,
  reps = 9999L,
  wtype = c("rademacher", "webb"),
  seed = NULL,
  ties = c("count", "strict"),
  conflvl = 0.95
)
```

Arguments

`data`	A data frame, one row per unit and period.
`unit`, `time`, `y`	Column names: the unit, the period (numeric, for example a year or a month index) and the outcome.
`trtd`	Column name of the treatment indicator (0 or 1).
`covs`	Optional column names of time-varying controls.
`cl`	Column name of the clusters (default the unit), or two column names for two-way clustering (Cameron, Gelbach and Miller 2011): $V_a + V_b - V_{ab}$, each term with the small-sample factor of the dimension with fewer clusters, and t on its $G - 1$ degrees of freedom. A composite cluster (district by month, say) is one column made with `interaction()`. Fewer than ten clusters give a warning (CR1 errors are unreliable with so few; `wcb` is made for that case), and so does a cluster that changes within a unit.
`reps`	Bootstrap replications (default 9999).
`wtype`	`"rademacher"` (default) or `"webb"`.
`seed`	Optional seed for the bootstrap weights.
`ties`	`"count"` (default) or `"strict"`: whether a bootstrap $\|t^*\|$ equal to $\|t\|$ counts against the null.
`conflvl`	Confidence level of the intervals reported (default 0.95).

Details

This is the restricted wild cluster bootstrap (WCR) of Cameron, Gelbach and Miller (2008). The model is fitted with the null $\beta = \beta_0$ imposed; each bootstrap outcome is the restricted fit plus its residuals multiplied, cluster by cluster, by a random sign (Rademacher) or by one of Webb's six values; the cluster-robust t statistic is recomputed each time. The p-value is the share of bootstrap $|t^*|$ at least $|t|$. With `ties = "count"` (the default) a draw that equals $|t|$ counts: the sign patterns that reproduce the data are among the draws, as the observed assignment is in a permutation test, and the test is exact at its level. With `"strict"` only draws above $|t|$ count, as `fwildclusterboot` does; under full enumeration this gives a smaller p-value (the two sign patterns that reproduce $|t|$ are left out). With Rademacher weights and $2^G \leq$ `reps` every sign pattern is enumerated (as `fwildclusterboot` does); with very few clusters (under about 12) Webb's weights, which have more distinct values, are better. The interval collects the β_0 not rejected at $1-$ `conflvl`, found by bisection with the same draws; it is unbounded when no value can be rejected (with Rademacher weights and G clusters the smallest attainable p-value is $2/2^G$, so four clusters can never reject at 5%: that is the honest answer, not a failure of the method).

Value

A data frame of class `rmbl_wild` with the columns `estimate`, `t`, `p_value`, `lower`, `upper`, `clusters`, `draws` and `enumerated`.

References

Cameron AC, Gelbach JB, Miller DL (2008). Bootstrap-based improvements for inference with clustered errors. Review of Economics and Statistics 90(3):414-427. DOI 10.1162/rest.90.3.414

Webb MD (2023). Reworking wild bootstrap-based inference for clustered errors. Canadian Journal of Economics 56(3):839-858. DOI 10.1111/caje.12661

Roodman D, Nielsen MO, MacKinnon JG, Webb MD (2019). Fast and wild: bootstrap inference in Stata using boottest. The Stata Journal 19(1):4-60. DOI 10.1177/1536867X19830877

Examples

```
p <- polex()
# the reform was assigned to four regions: four clusters
wcb(p, "district", "month", "crmrate", "reformed",
                       cl = "region", wtype = "webb", reps = 99,
                       seed = 1)

# your own panel: eight towns over ten years, four adopting a by-law (two
# in 2019, two in 2021), with each town's population, county and income
own <- data.frame(
  town = rep(c("A", "B", "C", "D", "E", "F", "G", "H"), each = 10),
  year = rep(2015:2024, times = 8)
)
i <- match(own$town, unique(own$town))
own$first <- c(2019, 2019, 2021, 2021, NA, NA, NA, NA)[i]
own$bylaw <- as.integer(!is.na(own$first) & own$year >= own$first)
own$pop <- c(12, 30, 8, 22, 15, 40, 9, 18)[i] * 1000
own$county <- rep(c("North", "South"), 4)[i]
own$income <- 50 + own$year - 2015 + cos(seq_len(80))
own$rate <- 30 + i + 0.5 * (own$year - 2015) - 4 * own$bylaw +
  0.2 * own$income + sin(seq_len(80))
# eight towns: every one of the 2^8 sign patterns
wcb(own, "town", "year", "rate", "bylaw", covs = "income")
```

---

welprof *Wellness beside distress: a strengths-first profile of published indicators*

Description

Puts a group's wellness indicators (life satisfaction, perceived mental health, a hopeful outlook, a sense of belonging, a strong cultural identity) first and its distress indicators (suicidal thoughts, mood and anxiety disorders, fair or poor mental health) after them, each with its published interval, for each First Nations, Metis and Inuit group: so that a table of mental health is never read, or plotted, as a list of deficits alone.

Usage

```
welprof(
  data,
  grp,
  ind,
```

```
  est,
  lower,
  upper,
  well,
  distress,
  domain = NULL,
  qual = NULL
)
```

Arguments

`data`	A data frame, one row per group and indicator (as `scests` gives).
`grp`, `ind`, `est`, `lower`, `upper`	Column names: the group, the indicator, the percentage and its published limits.
`well`, `distress`	The indicators that measure wellness or strength, and those that measure distress; rows for any other indicator (totals, "not specified") are left out.
`domain`	Optional named character vector giving each indicator's holistic domain, for example `c("Life satisfaction 8 to 10" = "mental", ...)`.
`qual`	Optional column of quality flags, one per row of `data`.

Details

Indigenous scholars of mental health define it as wellness and as a process of healing, grounded in community, cultural identity, a holistic view and interdependence (Stewart 2008), and ask that strengths be reported together with disparities (Stewart 2020). This function puts that into how published estimates are reported; it adds no estimate of its own. The optional `domain` groups indicators by the holistic domains of the Medicine Wheel teachings that Gabriel, Mirza and Stewart (2022) describe (physical, emotional, mental, spiritual). That grouping is a way to lay out the table, not a validated scale, and published tables have few indicators of the spiritual domain: the profile says what is there and does not stand one indicator in for another.

Aggregate tables show how indicators vary together across groups; they cannot show that culture, land or belonging protects mental health within a group, which would be an ecological inference. The differences these tables show between groups are the effects of colonial policy and of structural conditions, not of the people in the groups (Gabriel et al. 2022); their interpretation belongs with First Nations, Metis and Inuit communities.

Value

A data frame of class `rmbl_wellness_profile`, wellness rows first, with the columns `group`, `indicator`, `side` (`"wellness"` or `"distress"`), `domain`, `estimate`, `lower`, `upper` and `usewithcau`; `plot()` draws it.

Acknowledgement

This estimator puts to quantitative use the framework of Professor Suzanne Stewart (Yellowknives Dene First Nation, University of Toronto).

See the acknowledgements of the `first-nations-metis-inuit-health` vignette.

References

Stewart SL (2008). Promoting Indigenous mental health: cultural perspectives on healing from Native counsellors in Canada. International Journal of Health Promotion and Education 46(2):49-56. DOI 10.1080/14635240.2008.10708129

Stewart S (2020). Addressing strengths and disparities in Indigenous health. International Journal of Indigenous Health 15(1):4. DOI 10.32799/ijih.v15i1.34976

Gabriel MD, Mirza S, Stewart SL (2022). Exploring mental health and holistic healing through the life stories of Indigenous youth who have experienced homelessness. International Journal of Environmental Research and Public Health 19(20):13402. DOI 10.3390/ijerph192013402

See Also

scests, indgap, ordcmp.

Examples

```
# well-being, 2024 (table 41-10-0082-01)
wb <- indiex("wellbeing")
w <- welprof(
  wb,
  "group",
  "indicator",
  "estimate",
  "lower",
  "upper",
  well = c(
    "Life satisfaction rating of 8, 9 or 10",
    "Sense of meaning and purpose rating of 8, 9 or 10",
    "Always or often has a hopeful outlook"
  ),
  distress = c(
    "Life satisfaction rating between 0 and 5",
    "Rarely or never has a hopeful outlook"
  ),
  qual = "quality"
)
w
plot(w)

# your own table: made-up percentages for two communities, with limits
own <- data.frame(
  community = rep(c("Lakeside", "Riverbend"), each = 4),
  item = rep(c("Hopeful", "Satisfied", "Rarely hopeful", "Low satisfaction"),
             2),
  pct = c(64, 71, 12, 18, 58, 66, 16, 22),
  lo = c(59, 66, 9, 14, 52, 60, 12, 17),
  hi = c(69, 76, 15, 22, 64, 72, 20, 27)
)
welprof(own, "community", "item", "pct", "lo", "hi",
        well = c("Hopeful", "Satisfied"),
        distress = c("Rarely hopeful", "Low satisfaction"))
```

`wrmnfjs` *Write a Manifest to JSON*

Description

Writes a manifest to a JSON file, so the record of a run (its metadata, cross-checks and environment) can be kept next to the outputs, shared, and read back by any JSON reader. By default the file is pretty-printed for people; `canonical = TRUE` writes the byte-stable form that `mnfdgst()` hashes. The package's own JSON codec (`tojs()`) is used; jsonlite is not needed.

Usage

```
wrmnfjs(man, path, canon = FALSE)
```

Arguments

`man`	A manifest as returned by `mkmnf()` and built up with `recrd()`.
`path`	Character scalar, the destination file path.
`canon`	Logical scalar, default `FALSE`. `TRUE` writes the canonical form instead of the pretty-printed one: one line, keys sorted, which is what `mnfdgst()` hashes. Use it when the file itself has to be byte-stable rather than read by a person.

Details

Numbers are written with 17 significant digits, so every double reads back exactly; `NA` and `NULL` are written as `null`, and length-one vectors as scalars. When the manifest's `meta$synthetic` is `TRUE`, or the session is flagged synthetic (see `mkmnf()`), `meta$synthetic = true` is written whatever the list says, so a file written from generated data always says so. An existing file at `path` is overwritten; the folder must exist.

Errors when `path` is not a single non-empty string, and (from base R) when the file cannot be opened for writing.

Value

`path` (character scalar), invisibly.

See Also

`mnfcanon()`, `wrsmrytxt()`.

Examples

```
man <- mkmnf(list(project = "demo"), env = FALSE)
man <- recrd(man, "rowcount", obs = 20, expd = 20)
path <- wrmnfjs(man, tempfile(fileext = ".json"))
file.exists(path)

# Round-trips back through the package's own codec.
back <- fromjs(path, simplifyVector = FALSE)
# "PASS" ("INFO" after synthetic data, see mkmnf())
back$results$rowcount$status

# the canonical form: one line whose SHA-256 is the manifest digest
cpath <- wrmnfjs(man, tempfile(fileext = ".json"), canon = TRUE)
identical(readLines(cpath), as.character(mnfcanon(man)))

# the path must be one string
try(wrmnfjs(man, c("a.json", "b.json")))
```

---

wrrcdmnf *Write and verify a recode manifest*

Description

`wrrcdmnf()` saves a manifest from `rcdmnf` as a JSON file. `vrfrcdmnf()` reads such a file back and checks it against the data: that the mapping and the manifest body are unchanged since they were written, that the recode in the data still matches the recorded mapping, that the recoded counts are the recorded counts, and, for a signed manifest, that the signature verifies. Use the pair to hand a recode to a reviewer or to re-check it before a later analysis.

Usage

```
wrrcdmnf(man, path)

vrfrcdmnf(path, orig, recd)
```

Arguments

`man`	A manifest returned by `rcdmnf`.
`path`	Character string: the JSON file to write (`wrrcdmnf()`) or read (`vrfrcdmnf()`).
`orig`, `recd`	The vectors before and after the recode, to check the manifest against.

Details

`vrfrcdmnf()` collects every failure in `reasons` instead of stopping at the first:

- the SHA-256 of the stored mapping differs from `mapping_sha256`;

- the SHA-256 of the body (all fields except body_sha256, signature and public_key) differs from body_sha256, meaning the file was edited after it was written;
- vrfrcd(original, recoded, mapping) fails;
- the counts of recoded differ from counts_after.

The signature, when present, is checked with cpsvrf over the stored body_sha256. It therefore proves who wrote that hash; the body check above proves the body still matches it. Both are needed for ok = TRUE.

wrrcdmnf() errors when man is not a bricklayer_recode_manifest; it overwrites path.

Value

wrrcdmnf() returns path invisibly.

vrfrcdmnf() returns a list of class rmbl_recode_manifest_check (it prints as a box) with ok (logical), reasons (character; empty, character(0), when ok), signature_ok (logical, NA when the manifest is unsigned) and manifest (the parsed JSON as a list).

Examples

```
x <- c("W", "B", "W")
y <- grdrcd(x, c(W = "White", B = "Black"))
m <- rcdmnf(x, y, c(W = "White", B = "Black"))
p <- wrrcdmnf(m, tempfile(fileext = ".json"))
vrfrcdmnf(p, x, y)$ok

# the data changed after the manifest was written
y2 <- y
y2[2] <- "White"
vrfrcdmnf(p, x, y2)$reasons

# only manifests can be written
try(wrrcdmnf(list(a = 1), tempfile()))
```

---

wrsmrytxt *Write a Plain-Language Run Summary*

Description

Writes SUMMARY.txt into an output folder: a plain-text page a person can read without any software, giving the project, author, time, operating system and R version of the run, whether it ran on real or synthetic data, the paths it used, the tally of cross-check results, the files in the folder, and optional notes, contact and licence lines. It is the human-readable companion of wrmnfjs().

Usage

```
wrsmrytxt(man, outdir, paths, whatwasdone = NULL, contact = NULL, lic = NULL)
```

Arguments

man	A manifest as returned by mkmnf(); its meta supplies the header and its results the counts.
outdir	Character scalar, an existing folder to write SUMMARY.txt into and to list the files of.
paths	A named list (or named character vector) of paths to report, one line each, labelled by the capitalised name. Names such as capsule, input, results, analysis_script and provenance are typical. list() reports none.
whatwasdone	Optional character vector, one line each, describing what the run did. Default NULL (no section).
contact	Optional character scalar appended as a Contact: line. Default NULL.
lic	Optional character scalar appended as a Licence: line. Default NULL.

Details

Header fields come from manifest$meta: project (default "(unnamed)"), author ("(unknown)"), run_at (default the current time), os (default Sys.info()), r_version (default R.version.string) and synthetic. When meta$r_package_versions is present a section lists those versions. Result counts are tallied from the status of each entry of manifest$results (PASS, DIFFER, WARN, INFO).

The file list is list.files(outdir) at the time of the call, sorted, so write the summary last. The paths are printed as given (they are not made absolute). An existing SUMMARY.txt is overwritten.

Errors when man is not a list, and (from base R) when outdir does not exist or cannot be written.

Value

The path to the written SUMMARY.txt (character scalar), invisibly.

See Also

mkmnf(), recrd(), wrmnfjs().

Examples

```
man <- mkmnf(list(project = "demo", author = "A. Author"),
  env = FALSE
)
man <- recrd(man, "rowcount", obs = 20, expd = 20)
out <- file.path(tempdir(), "demo-run")
dir.create(out, showWarnings = FALSE)
s <- wrsmrytxt(man, out, paths = list(results = out))
readLines(s)[7:8]

# with notes, contact and licence lines
s <- wrsmrytxt(man, out, paths = list(results = out),
  whatwasdone = c("- read the extract", "- recomputed table 2"),
  contact = "analyst at example.org", lic = "CC BY 4.0")
tail(readLines(s), 6)
```

```
# the manifest has to be a list
try(wrsmrytxt("demo", out, paths = list()))
```

`wrtxtfb` *Write Text as UTF-8, Falling Back to ASCII on an Encoding Error*

Description

Writes `text` to `path` as UTF-8, one element per line. If the write fails (for example a destination or locale that cannot represent the characters), it retries with an ASCII transliteration produced by `toascii()`, so capsule generation never fails on a non-ASCII name. Use it to write generated text files (README, citation, script) whose content may include names with accents.

Usage

```
wrtxtfb(text, path)
```

Arguments

`text`	A character vector of lines to write; `as.character()` converts any other type.
`path`	The destination file, a single non-empty string. Its directory must exist.

Details

The lines are joined with `"\n"`, a final newline is added, and the UTF-8 bytes are written through a binary connection, so the bytes on disk are UTF-8 whatever the session locale (a text connection would re-encode from the native locale) and line endings are `"\n"` on every platform. An existing file is overwritten. A missing element is written as the two letters `NA`, and an empty `text` writes a file holding a single newline.

The ASCII retry is taken on any error from the first write, not only on an encoding error; if the retry also fails (a directory that does not exist, a read-only location), that error reaches the caller. A `path` that is not a single non-empty string is refused before anything is written (`"path must be a single non-empty string"`).

Value

Invisibly, `path`.

See Also

`toascii()` for the retry's transliteration.

Examples

```
p <- wrtxtfb(c("line one", "line two"),
                         tempfile(fileext = ".txt"))
readLines(p)

# Accented names are written as UTF-8 bytes, whatever the locale.
q <- wrtxtfb("Author: \u00c1ngela Zorro Medina", tempfile())
readLines(q, encoding = "UTF-8")
readBin(q, "raw", 20)

# The path must be one string.
try(wrtxtfb("x", c("a.txt", "b.txt")))
```

---

xfrvrf *Verify a categorical variable that crossed from one program to another*

Description

Checks a categorical variable that was moved from SPSS, Stata or SAS into R against what the source program reported for it. You give the frequency table the source program printed (label = count) and, optionally, its code book (code = label); the function decodes the imported vector and refuses to continue unless it reproduces those counts exactly and its value labels agree with the code book. Rotated, swapped or positionally relabelled groups fail at import, with the permutation named.

Usage

```
xfrvrf(imported, soucnts, codebook = NULL, tol = 0, strict = TRUE)
```

Arguments

`imported`	The vector as it arrived. It is a haven-style labelled vector, plain codes given with `codebook`, or labels already decoded.
`soucnts`	Named numeric vector, label = count, as printed by the source program (SPSS FREQUENCIES, Stata tabulate).
`codebook`	Optional named character vector, code = label, from the source program's variable view. Default `NULL`. When `imported` carries a `labels` attribute the two are compared and any disagreement fails the check.
`tol`	Non-negative number, default `0`: the absolute difference in count allowed per label, passed to `vrfmrg`.
`strict`	Logical, default `TRUE`: any disagreement is an error. When `FALSE` the result comes back with `ok = FALSE`, `reasons`, and `marginals$permutation` ready for `rlblfrns`.

Details

Steps:

1. When `imported` has a `labels` attribute and `codebook` is given, the two are compared code by code; any code whose label differs (or that is present in only one) is a disagreement.
2. The vector is decoded: by its `labels` attribute (`declbld`), else by `codebook`, else taken as already-decoded labels.
3. The decoded labels are checked with `vrfmrg` against `soucnts`.

With `strict = TRUE` the first disagreement is an error. With `strict = FALSE` both checks run and are reported. A code without a label is an error in either mode (from the decoding step).

Value

A list of class `rmbl_transfer_check` (it prints as a box) with `ok` (logical), `decoded` (a factor; in code order when decoded from codes), `marginals` (the `vrfmrg` result), `code_book_ok` (logical, or `NA` when there was nothing to compare) and `reasons` (character messages, or `NULL` when `ok`).

See Also

`declbld`, `vrfmrg`, `rlblfrns`

Examples

```
x <- structure(
  c(1, 1, 2, 4, 1),
  labels = c(White = 1, Black = 2, Other = 3, Unknown = 4)
)
xfrvrf(x, c(White = 3, Black = 1, Unknown = 1))$ok

# counts that only match under swapped labels, reported without stopping
r <- xfrvrf(x, c(White = 1, Black = 3, Unknown = 1), strict = FALSE)
r$ok
r$marginals$permutation

# value labels that disagree with the source code book
try(xfrvrf(
  x,
  c(White = 3, Black = 1, Unknown = 1),
  codebook = c("1" = "White", "2" = "Other", "3" = "Black", "4" = "Unknown")
))
```

xmssadv *Advance an XMSS key past a signature it made*

Description

An XMSS signature consumes one leaf, so the key that made it must not sign at that index again. This returns the key with `next_index` set to one past the signature's index (refusing a signature that was not made by this key), which is the key to save and to use for the next signature. The signature object itself carries no key material.

Usage

```
xmssadv(key, sig, msg = NULL)

xmssregkey(keytoreg)

xmssfgt(keytofgt = NULL)
```

Arguments

`key`	The `bricklayer_signing_key` that made `sig`.
`sig`	The XMSS signature it made, from `cpssign()`, or the attestation, bundle or recode manifest that carries it.
`msg`	The message the signature was made over; when given, the signature is verified against the key first, so an edited or foreign signature cannot advance it. Without it, the signature's index is accepted only up to the highest index the record shows this key has used here, so an edited index cannot burn the key's remaining leaves; a signature made on another machine needs `msg`.
`keytoreg`	For `xmssregkey()`: a signing key generated on another machine, regenerated from given seeds, or whose record was deleted, that has never signed; it is added to the record so that it may sign from `next_index`. A key the record shows has signed is refused.
`keytofgt`	For `xmssfgt()`: a signing key (the record is keyed by its secret seed) whose entry alone is reset to "nothing used" (it may sign from index 0 again); `NULL` removes the whole record (every key, and the record of the releases before 0.5.12), after which only keys generated afterwards, or vouched for by `xmssregkey()`, may sign.

Details

The package records the highest index each key has signed with and refuses that index again, within the session and across sessions on this machine. The record is the file `xmss_used_indices.txt` under `tools::R_user_dir("rmoriebricklayer", "config")`, which is where CRAN policy lets a package keep its own configuration. A key is entered under a fingerprint of its secret seed (`skseed`), not under the `root` it carries, so editing a stale copy's `root` does not make it a new key, and a signature whose `root` does not match the seeds is withheld. `pqckg()` registers a key it generates

from fresh randomness. A key built from given seeds is not registered, because it may be one that has signed elsewhere: it signs from index 0 only after `xmssregkey()` vouches for it, and the same holds for any key the record does not know (from another machine, or whose record was deleted). Every reservation runs under an operating-system file lock (`flock` on Unix, `LockFileEx` on Windows) on `xmss_used_indices.txt.flock`, which the system releases when the process ends, so a crashed session leaves no lock for another process to take over. A record that cannot be read, or that cannot be written in full (a full disk), stops the signature. The record of the releases before 0.5.12 (`xmss_index.json` in the same directory) is read and merged, and kept as `xmss_index.json.migrated` once the new record has been written. Another path can be set with the option `rmoriebricklayer.xmssidxfile` or the environment variable `RMBL_XMSS_INDEX_FILE`: an absolute path to a regular file, or to one that does not exist yet (a device such as `/dev/null`, a directory or a relative path is refused). The option `FALSE` (or, except on Windows, where an empty environment variable is an unset one, the value `""`) turns the file off: then only the session remembers, every signature warns, and a stale copy of the key reloaded elsewhere signs a used index again. The record does not travel with the key and is lost if the file is deleted, so a key copied to another machine is protected only by the `next_index` you saved: save the advanced key after every signature. In CI or a container with a fresh home directory, a key that was not generated in that run is unknown to the record: register it once with `xmssregkey()`, or point the record at a file that persists. Examples and vignettes point the record at a temporary file so that checks and builds leave the configuration directory alone. The record keeps one line per key for good (an exhausted key's line is what stops a stale copy of it), and is read and rewritten whole at every signature: a whole signature took 0.5 s with a hundred thousand keys in the record (measured on an x86-64 laptop; a slower machine takes longer), and longer when several processes sign at once (each waits its turn for the lock, up to 10 s).

`xmssfgt()` resets one key's entry, or removes the whole record, and with it the protection against re-signing: use it for test keys only.

Value

The key with `next_index` advanced past the signature's index and past every index the record knows this key has used.

See Also

`cpssign()`, `xmssfgt()`

Examples

```
# keep this example's
op <- options(rmoriebricklayer.xmssidxfile = tempfile())
# one-time index record out of your configuration directory; leave the
# default in use
key <- pqckg(height = 2)
s1 <- cpssign("manifest-1", key)
key <- xmssadv(key, s1)
key$next_index
s2 <- cpssign("manifest-2", key)
cpsvrf("manifest-2", s2, signpubkey(key))

# With the message, the signature is verified before the key advances.
```

```
key <- xmssadv(key, s2, msg = "manifest-2")
key$next_index

# A key regenerated from seeds is unknown to the record until registered.
s <- paste(rep("ab", 32), collapse = "")
k2 <- pqckg(height = 1, skseed = s, pubseed = s, skprf = s)
try(cpssign("m", k2))
invisible(xmssregkey(k2))
cpsvrf("m", cpssign("m", k2), signpubkey(k2))

# Reset a test key's entry so it may sign from index 0 again.
xmssfgt(k2)

# A signature that is not an XMSS signature cannot advance a key.
try(xmssadv(key, list()))
options(op)
```

`xmssidxseen` *Find XMSS signature indices used twice*

Description

Lists every XMSS signature in the capsule bundles under a directory, and in seen-logs, and marks an index that one key signed with two different signatures. An XMSS index must sign once: two signatures at one index let anyone forge a third message, so a reused index means the key's state was copied or rolled back (RFC 10033, section 4: a verifier that sees all of a signer's signatures can detect reuse, since each carries its index).

Usage

```
xmssidxseen(path = ".", seenlog = NULL)
```

Arguments

`path`	A directory (every `capsule_bundle.json` under it) or one bundle file.
`seenlog`	Seen-log files to read as well, or NULL. A file that does not exist yet is an empty log; a directory, or a log with a malformed line, is refused.

Details

A seen-log is the text file `rmbl sign --seen-log FILE` appends to: one line per signature it made, with the key fingerprint, the index, a digest of the signature and the bundle it went into, tab-separated; lines starting with # are comments. The same signature found twice (one bundle copied to two places, or a bundle and its log line) is not a reuse.

Value

A data frame, one row per signature seen: `key` (the key fingerprint, the first 16 hex characters of SHA-256 over the public key), `index`, `sig` (a digest of the signature), `where` (the file) and `reused` (`TRUE` when this key signed this index with another signature too).

See Also

`cpsbndlvrf()` (its `seen` argument), `blcli()` (`rmbl sign --seen-log`), `pqckg()`.

Examples

```
# keep this example's
op <- options(rmoriebricklayer.xmssidxfile = tempfile())
# one-time index record out of your configuration directory
root <- tempfile("capsules-")
d <- file.path(root, "one")
dir.create(d, recursive = TRUE)
writeLines("x\n1", file.path(d, "data.csv"))
key <- pqckg(height = 2)
b <- cpsbndl(d, mkmnf(list(), env = FALSE), key)
xmssidxseen(root)

# the same index signed again from a stale copy of the key: reused
e <- file.path(root, "two")
dir.create(e)
writeLines("x\n2", file.path(e, "data.csv"))
# forget the record, as a copy on another machine would
invisible(xmssfgt(key))
b2 <- cpsbndl(e, mkmnf(list(), env = FALSE), key)
xmssidxseen(root)[, c("index", "reused")]

try(xmssidxseen(tempfile()))
options(op)
```

---

yoy *Year-over-year (and period-over-period) change*

Description

Computes the change from one period to the period `lag` places before it, matched on the period's own value rather than on row order, and carries an exact interval for count data. Use it for the "up 12% on last year" figures of an annual report or an open-data release: a missing year shows as a gap instead of shifting every comparison, a percentage is compared in percentage points, and a change off a small count comes with an interval as wide as the count is small.

Usage

```
yoy(x, ...)

## Default S3 method:
yoy(x, ...)

## S3 method for class 'data.frame'
yoy(
  x,
  value,
  prd,
  by = NULL,
  lag = 1L,
  fun = NULL,
  units = c("count", "continuous", "percent"),
  minbase = NULL,
  conflvl = 0.95,
  dir = c("neutral", "higher", "lower"),
  complete = TRUE,
  prdord = NULL,
  ...
)

## S3 method for class 'numeric'
yoy(x, prd = seq_along(x), ...)

## S3 method for class 'integer'
yoy(x, prd = seq_along(x), ...)

## S3 method for class 'ts'
yoy(x, lag = NULL, ...)

## S3 method for class 'rmbl_yoy'
format(x, digits = 1L, palette = "diverging",
  color = FALSE, n = 30, ...)

## S3 method for class 'rmbl_yoy'
print(x, digits = 1L, palette = "diverging", color = NULL, n = 30, ...)
```

Arguments

x	An rmbl_yoy object returned by yoy().
...	Further arguments from print(); the table ignores them.
value	For a data frame, the column holding the measure, as a string or a bare name.
prd	For a data frame, the column holding the period (a year, a Date, a fiscal-year integer, an ordered factor, or text). An ordered factor is compared by level order, one level being one period; an unordered factor has no order, so every row is flagged “no comparison period”. For a numeric vector, the periods themselves.

Text periods are read when every label has the same one of these forms (one step is the period itself, so a missing period is a gap):

- a year: `"1987"`, `"2031"`;
- a fiscal year by the two years it spans, ordered by the first: `"2019/20"`, `"2019/2020"`, `"2019-20"`, `"2019-2020"`, `"FY 2019-20"`, `"FY2019/20"`; or by one year: `"FY2020"`, `"FY 2020"` (the two kinds do not mix);
- a quarter: `"2021 Q3"`, `"2021Q3"`, `"2021-Q3"`, `"Q3 2021"`;
- a month: `"2021-03"`, `"2021/03"`, `"2021-3"`, or an English month name or its three-letter abbreviation: `"Mar 2021"`, `"March 2021"`, `"2021 Mar"`;
- a date: `"2021-03-15"`, `"2021/03/15"`, compared as a `Date`.

The result keeps the labels as text, with any period the grid inserts spelled like the first one. Labels of mixed forms, or of a form not listed (`"Q3"`, `"FY19/20"`), or that read as two forms alike (`"2010-11"`, `"2011-12"`: fiscal years or months), have no order: yoy() warns once, naming them, and no row has a comparison period. Give their order with `prdord`.

`by` Optional grouping columns, as a character vector. The change is computed within each group.

`lag` How many periods back to compare with. `1` is year-over-year on annual data; for a `ts` the default follows the series' own frequency, so monthly data compares with the same month a year earlier.

`fun` How several rows of one period (and group) are combined into one value: `NULL` (the default) sums counts (`sum()`) and averages continuous values and percentages (`mean()`); pass a function of a numeric vector returning one number to choose.

`units` What the measure is. `"count"` gets the exact rate-ratio interval. `"continuous"` gets the percent change without one, since a single pair of totals carries no information about its own variability. `"percent"` reports a percentage-POINT change and withholds the percent change, which for a percentage is a different quantity.

`minbase` Optional: previous-period values below this get a "small base" note. Nothing is withheld either way: every non-zero base gets its percent change and exact interval, which is as wide as the base is small. Defaults to 0 (no note); only a zero base has no percent change.

`conflvl` Numeric scalar in (0, 1), the confidence level for the count interval; default 0.95.

`dir` Which way is an improvement: `"higher"`, `"lower"` (segregation days, deaths in custody, use of force), or `"neutral"`. Affects colour and the `verdict` column only, never the arithmetic.

`complete` Logical scalar, default `TRUE`: insert the missing periods in the observed range so that a gap is visible as a gap instead of closing up.

`prdord` Optional character vector: the period labels in order, one label being one step, for text (or factor) periods the reader above does not know. Every label in the data must be in it; one it lists that the data lacks is a gap. `NULL` (the default) reads the labels as above.

`digits` Integer, default `1`. Decimal places shown in the percent (or percentage-point) change column. The interval column is always shown to whole percent.

palette — Character, default "diverging". One of yoypals(); chooses the colours used when color = TRUE. An unknown name is an error.

color — Logical or NULL (the default). TRUE wraps each row in 24-bit ANSI colour codes chosen from its verdict; FALSE prints plain text. NULL means colour only in an interactive session with the NO_COLOR environment variable unset, so output that is redirected or captured stays plain text. format() takes TRUE or FALSE (default FALSE).

n — Integer, default 30. Maximum number of rows printed; the number of rows left out is reported on a final line.

Details

Rows are first combined to one per period (and group) with fun. With complete = TRUE the periods missing from a regular grid (numbers, Date or POSIXct with a common step, the levels of an ordered factor, text periods read as described under prd) are inserted with NA values and flagged "no data". Each period is then compared with the period lag steps earlier in that grid: change is the difference, pctchg $100(v_t/v_{t-lag} - 1)$ (NA off a zero base), and for percentages ppchg is the difference in percentage points.

For counts, conditional on their sum $n = a + b$ the current count a is binomial with probability $\rho/(1 + \rho)$, ρ the rate ratio; the Clopper-Pearson interval for that probability, transformed back, is an exact interval for the ratio, reported as pctlo and pctup (percent change). It is the interval stats::poisson.test() gives for two counts with equal exposure. A zero base has no percent change but still an interval (with an infinite upper end).

verdict is "up", "down" or "unchanged" for direction = "neutral", and "better", "worse" or "unchanged" otherwise; it reads the sign of the change only and makes no claim of significance.

Errors: x neither a data frame, a numeric vector nor a ts; value, period or by naming a column that does not exist; a non-numeric or infinite value; for units = "count", negative or fractional values; lag not a positive whole number; conflvl outside (0, 1); an unknown units or direction; a prdord that is not distinct labels, lacks a label of the data, or is given for periods that are not text.

Value

An rmbl_yoy object (a data frame of class c("rmbl_yoy", "data.frame")) with one row per period (and group): the period column under its own name (period for a vector or a ts), the grouping columns, then value, previous, change, pctchg (or ppchg for percentages), flag (character: "no comparison period", "no data", "previous period is zero", or a small-base note when minbase is set; NA otherwise), pctlo and pctup for counts, and verdict. Attribute "yoy" records value, period, by, lag, units, minbase, conflvl, direction and the display digits; the writers (yoywr()) and yoysmry() read it.

0.5.x argument names

The 0.5.x names period, min_base, conf_level and direction are still accepted for prd, minbase, conflvl and dir, the names to use, as are the 0.5.x values "higher_is_better" and "lower_is_better" of dir. A call that gives both names of one argument is refused.

References

The exact interval for a ratio of two counts is the conditional-binomial (Clopper-Pearson) one, which is the construction `stats::poisson.test` uses and which this is verified against. On the reporting conventions, the Toronto Police Service's *Understanding Strip Searches in 2020 Methodological Report* – in the local corpus – reports year-over-year change on exactly this kind of administrative extract, and is the shape this function is built for.

See Also

`yoyhtml()`, `yoypdf()`, `yoysmry()`

Examples

```
seg <- data.frame(
  EndFiscalYear = rep(2019:2023, each = 2),
  Gender = rep(c("Female", "Male"), 5),
  NumberOfPlacements = c(31, 402, 28, 377, 12, 190, 19, 268, 24, 331)
)

y <- yoy(
  seg,
  value = "NumberOfPlacements",
  prd = "EndFiscalYear",
  by = "Gender",
  dir = "lower"
)
y

# The interval is exact, so a small group does not get a confident
# percent it has not earned.
subset(as.data.frame(y), Gender == "Female")

# A percentage is handled as percentage points, not as a percent of a
# percent.
rate <- data.frame(year = 2019:2023, share = c(4.1, 4.6, 5.2, 5.0, 5.4))
yoy(rate, value = "share", prd = "year", units = "percent")

# a missing year stays a gap; a bare column name works too
yoy(
  data.frame(year = c(2019, 2021, 2022), n = c(10, 12, 15)),
  value = n,
  prd = year
)

# a quarterly series compares each quarter with the same quarter a year
# earlier
q <- ts(c(5, 6, 7, 8, 6, 7, 9, 9), frequency = 4, start = c(2020, 1))
as.data.frame(yoy(q))[5:8, c("period", "value", "previous", "pctchg")]

# text periods are read: fiscal years, quarters, months
fy <- data.frame(fy = c("2019/20", "2020/21", "2022/23"), n = c(5, 7, 9))
as.data.frame(yoy(fy, "n", "fy"))[, c("fy", "previous", "flag")]
```

```
qt <- data.frame(q = c("2021 Q4", "2022 Q1", "2022 Q2"), n = c(3, 4, 6))
as.data.frame(yoy(qt, "n", "q"))[, c("q", "previous", "pctchg")]

# labels the reader does not know take their order from `prdord`
ws <- data.frame(s = c("spring", "summer", "fall"), n = c(4, 6, 5))
yoy(ws, "n", "s", prdord = c("spring", "summer", "fall", "winter"))

# counts must be whole numbers
try(yoy(data.frame(year = 1:3, n = c(1.5, 2, 3)), "n", "year"))
```

---

yoybnd *Add publication bounds to a year-over-year table*

Description

yoy() gives an exact conditional-binomial interval for the percent change of a count, which covers sampling variation. This adds the interval that rounding and suppression in the published cells imply (pubbnd() carried through chgenv()), and a combined interval that is the union of the two, which is the honest range for a figure read off a rounded table.

Usage

```
yoybnd(y, rnd = NULL, rndkind = c("nearest", "random"), suplim = NULL)
```

Arguments

y	An rmbl_yoy object from yoy(), with counts as values.
rnd	Numeric scalar, the base the release rounds to (5, 10, ...), or NULL (the default) when the counts are exact.
rndkind	"nearest" (the printed value is the observed value rounded to the nearest multiple of rnd, so the observed value lies within half a base) or "random" (Statistics Canada's random rounding, where the observed value lies within rounding - 1).
suplim	Numeric scalar or NULL (the default). For a cell printed as suppressed with no explicit limit, the smallest count that would have been published: the cell then lies in [0, suplim - 1].

Details

The value and previous columns of y are passed to pubbnd() with the same rounding and suppression settings, and the two intervals to chgenv(). The combined interval is the smallest interval that contains both the sampling interval (pctlo, pctup) and the publication envelope: its lower end is the smaller of the two lower ends, its upper end the larger of the two upper ends. Where either is NA (the first period, or a previous cell that could be zero) the combined end is NA. The function does not check that y holds counts; publication bounds make sense only for counts.

Errors when y does not come from yoy(), and for the reasons pubbnd() gives.

Value

y (still an rmbl_yoy) with added numeric columns envchglo, envchgup, envpctlo, envpctup and, when y has percent-change intervals, compctlo and compctup. The rounding settings are recorded as bounds in the "yoy" attribute.

See Also

pubbnd(), chgenv(), yoy().

Examples

```
seg <- data.frame(year = 2021:2023, n = c(40, 45, 60))
y <- yoy(seg, value = "n", prd = "year")
yoybnd(y, rnd = 5)

# random rounding to base 5 widens the envelope
yb <- yoybnd(y, rnd = 5, rndkind = "random")
as.data.frame(yb)[, c("year", "envpctlo", "envpctup",
                      "compctlo", "compctup")]

# only yoy() results are accepted
try(yoybnd(seg, rnd = 5))
```

---

yoydelim *Write a change table as delimited text, JSON or Markdown*

Description

yoycsv() separates fields with a comma and quotes any field that contains one; yoytsv() separates with a tab and replaces any tab inside a field, since a tab-separated field cannot contain one and writing it would shift every column after it. yoyjs() writes an object with settings (value, period, grouping, lag, units, minimum base, confidence level, direction), summary (from yoysmry()) and rows, using the package's own JSON codec. yoymd() writes a pipe table meant for reading: values formatted (every numeric column right-aligned), percentages signed and rounded, the interval as [lower, upper], notes in the last column, and the settings in an italic line under the table.

Usage

```
yoycsv(x, file, digits = NULL, na = "", meta = TRUE, comment = "#", ...)

yoytsv(x, file, digits = NULL, na = "", meta = TRUE, comment = "#", ...)

yoyjs(x, file, digits = NULL, pretty = TRUE, ...)

yoymd(x, file, digits = 1L, align = TRUE, ...)
```

Arguments

x	An rmbl_yoy object from yoy().
file	Character scalar, the output path, or NULL to return the text instead of writing it.
digits	A whole number from 0 to 15, or NULL: decimal places for the percent columns. For the CSV, TSV and JSON writers NULL, the default, writes them unrounded, which is what a downstream calculation wants; give a number for a figure meant to be read. For yoymd() the default is 1.
na	Character scalar, how to write a missing value (CSV and TSV). The default empty string is what most readers expect; "NA" keeps R's own spelling.
meta	Whether to lead with comment lines recording the lag, units, confidence level, base gate and direction. On by default, because a percent column means different things under different settings and the file is the only place a later reader can look. Markdown and JSON carry the same information structurally.
comment	Character scalar, the comment prefix for the metadata lines; default "#".
...	Ignored: what yoywr() passes on for its HTML and PDF writers.
pretty	Logical scalar, default TRUE: indent the JSON.
align	Logical scalar, default TRUE: pad the Markdown columns so the source table is readable unrendered.

Details

The CSV and TSV writers export every column of the object (the period and grouping columns, value, previous, change, the percent or percentage-point change, flag, the interval and verdict), so the note explaining a missing or flagged figure stays with its row. Quoting in the CSV follows RFC 4180 (a field with a comma, a quote or a line break is quoted, quotes doubled). The metadata line names the value and period columns, the grouping, the lag, the units, and for counts the interval's confidence level and the base below which changes are noted. A pipe inside a Markdown cell is escaped. An existing file is overwritten. Errors when x does not come from yoy(), or digits is not a whole number.

Value

file, invisibly; or, when file is NULL, the text as a character scalar.

Examples

```
d <- data.frame(year = 2019:2023, n = c(402, 377, 190, 268, 331))
y <- yoy(d, value = "n", prd = "year")

# Straight to text, for inspection.
cat(yoycsv(y, NULL))

# Markdown, for a report.
cat(yoymd(y, NULL))

# JSON keeps the settings as fields rather than as comments.
substr(yoyjs(y, NULL), 1, 80)
```

```
# rounded percentages, no metadata line, and R's own NA
cat(yoytsv(y, NULL, digits = 1, meta = FALSE, na = "NA"))

# written to a file
f <- tempfile(fileext = ".csv")
yoycsv(y, f)
read.csv(f, comment.char = "#")[, c("year", "pctchg", "flag")]

# only yoy() results are accepted
try(yoycsv(d, NULL))
```

`yoylbl` *Label a period without changing it*

Description

Attaches display labels to the periods of a change table from `yoy()`, so a fiscal year keyed as 2023 can print as `"2022/23"`, or a year can carry a note such as "(COVID)". The arithmetic already ran on the period's value, so a label can only affect what is printed: relabelling cannot move a number.

Usage

```
yoylbl(x, labels)
```

Arguments

`x`	An `rmbl_yoy` object from `yoy()`, whose periods are labelled.
`labels`	Either a function applied to the period column, or a character vector the same length as the number of distinct periods, or a named character vector mapping a period (as a string) to its label.

Details

The labels are stored in a new `perlbl` column, and the table's `yoy` attribute is marked so that `print()`, `yoyhtml()`, `yoypdf()` and the delimited writers show the label in place of the period. The period column itself is unchanged, as are all values, changes and verdicts.

`labels` is interpreted as follows: a function is called once on the period column and must return one label per row; a named character vector maps periods (matched as strings) to labels, and periods it does not name keep their own value; an unnamed character vector gives one label per distinct period, in order of first appearance.

Errors: `x` not from `yoy()`; an unnamed vector whose length is not the number of distinct periods; a function that does not return one label per row.

Value

The object, still of class `rmbl_yoy`, with an added character column `perlbl`. Its `yoy` attribute gets `perlbl` set to `TRUE`, as `attr(x, "yoy")$perlbl`, so the labels are used by `print()`, `yoyhtml()`, `yoypdf()` and the delimited writers.

See Also

`fylbl()`

Examples

```
seg <- data.frame(EndFiscalYear = 2019:2023,
                  n = c(402, 377, 190, 268, 331))
y <- yoy(seg, value = n, prd = EndFiscalYear)

# 2023 means the fiscal year 2022/23, so say so.
yoylbl(y, fylbl)

# Any function will do.
yoylbl(y, function(p) paste0("FY", substr(p, 3, 4)))

# Or an explicit mapping, for the periods that need one.
yoylbl(y, c("2020" = "2019/20 (COVID)"))

# the numbers are untouched
identical(yoylbl(y, fylbl)$change, y$change)

# an unnamed vector must have one label per distinct period
try(yoylbl(y, c("a", "b")))
```

---

yoypals *Colour palettes for change tables*

Description

Lists the names of the colour palettes that the change-table renderers accept. In every palette the colour says whether a change is good or bad news for the series, not whether the number went up or down: the arrow in the table carries the sign, and the colour carries the `direction` chosen in `yoy`. Use this to see which values the `palette` argument of `print()`, `yoyhtml` and `yoypdf` takes.

Usage

```
yoypals()
```

Details

Three palettes are defined:

`"diverging"` the default; red-orange (`#c2492f`) for bad news and teal (`#1b7f6b`) for good news, hues pulled apart so the pair stays distinguishable in the common forms of colour blindness.

`"safe"` orange (`#b8621b`) and blue (`#18567d`), safe for deuteranopia and protanopia alike.

`"mono"` black on grey; `print()` emits no colour codes at all with this palette.

Unchanged rows and rows without a comparison use grey (`#6b6b6b`). The verdict-to-colour mapping is: `"worse"` uses the bad colour and `"better"` the good colour. With `direction = "neutral"` in yoy the verdicts are `"up"`/`"down"`, which say nothing about good or bad news, so those rows use the grey as well; the arrow still shows the sign. A palette name not in this list is an error in every renderer.

Value

A character vector of palette names, currently `c("diverging", "safe", "mono")`.

See Also

yoy, yoysmry, yoyhtml, yoypdf.

Examples

```
# the palette names
yoypals()

# use one of them when printing a change table
d <- data.frame(year = 2019:2023, n = c(40, 46, 38, 52, 49))
print(yoy(d, value = "n", prd = "year"), palette = "safe", color = FALSE)

# a name that is not a palette is refused
try(print(yoy(d, value = "n", prd = "year"), palette = "rainbow"))
```

---

yoypv *Exact p-values for the rows of a year-over-year table*

Description

Adds a two-sided exact p-value to each row of a period-over-period change table of counts from yoy, testing whether the count changed by more than chance between the two periods. Use it when you want a test to go with each change, and then pass the table to scanadj to correct for the number of rows tested.

Usage

```
yoypv(y)
```

Arguments

y	An rmbl_yoy object from yoy with units = "count".

Details

For count units the interval in yoy() is the exact conditional binomial interval for the ratio of two Poisson means; this is the matching two-sided test. The test conditions on the total of the two periods and asks whether the split is compatible with equal means: with current count a and previous count b, $a \sim \text{Binomial}(a+b, 1/2)$ under the null, and the p-value is that of stats::binom.test(a, a + b, p = 0.5). The two periods are treated as having equal exposure; for changing populations use ratechg and scanadj, which weight by the exposures. Counts are rounded to whole numbers.

Errors: y not a yoy() result; a yoy() result whose units are not counts (percentages and measured values have no Poisson test). A progress indicator is shown for long tables in an interactive session.

Value

y, with its class and attributes, and an added numeric p_value column; NA where there is no comparison period or both counts are zero.

See Also

scanadj, yoy.

Examples

```
seg <- data.frame(year = rep(2022:2023, each = 3),
                  site = rep(c("A", "B", "C"), 2),
                  n = c(30, 50, 80, 45, 52, 79))
y <- yoy(seg, value = "n", prd = "year", by = "site")
yoypv(y)$p_value

# the same test by hand for site A: 45 against 30
stats::binom.test(45, 75)$p.value

# anything that is not a yoy() table is refused
try(yoypv(seg))
```

---

yoyren *Render a change table to HTML or PDF*

Description

Writes a change table from yoy() to a file for people who will not run R: yoyhtml() to a single self-contained web page, yoypdf() to a PDF. Each row shows the period, the value, the previous value, the change and the percent change, coloured by whether it is better or worse, with the exact interval and any flags; the HTML page also has summary cards from yoysmry(). Use them to hand a reviewed set of period-over-period changes to a manager, a journalist or a court.

Usage

```
yoyhtml(
  x,
  file,
  title = "Year-over-year change",
  sub = NULL,
  pal = "diverging",
  digits = 1L,
  bars = TRUE,
  intv = TRUE,
  notes = NULL,
  ...
)

yoypdf(
  x,
  file,
  title = "Year-over-year change",
  sub = NULL,
  pal = "diverging",
  digits = 1L,
  intv = TRUE,
  notes = NULL,
  width = 11,
  height = 8.5,
  ...
)
```

Arguments

x	An rmbl_yoy object from yoy(), written as HTML.
file	Output path (a string). yoyhtml() writes a single self-contained HTML file with no external requests; yoypdf() writes a PDF using R's own device, so neither needs a package beyond base R.
title	Heading for the page (and the HTML page title or PDF title), a string; default "Year-over-year change".
sub	Optional line under the heading. Defaults to the table's own settings – lag, units, interval and base gate – so the reader can see what the percentages mean.
pal	One of yoypals(): "diverging" (the default), "safe" or "mono".
digits	Digits for the percent column, default 1.
bars	Logical (HTML only): draw an in-cell bar proportional to the change, scaled to the largest absolute change in the table (default TRUE).
intv	Logical: show the exact interval column for counts (default TRUE).
notes	Optional character vector of footnotes.
...	Further arguments; the HTML writer does not use them.
width, height	PDF page size in inches (default 11 by 8.5, landscape US letter).

Details

Neither backend needs a package beyond base R. The HTML page inlines its stylesheet, makes no external requests (no fonts, scripts or images from the network) and escapes every value, so the file stays the artefact and renders the same later. Rows are coloured by the table's `verdict`: `"worse"` in the palette's bad colour, `"better"` in its good colour, everything else (including the `"up"` and `"down"` of `direction = "neutral"`) in its neutral grey. Labels set with `yoylbl()` replace the period. For percentage units the change is shown in points. The page ends with the package version that wrote it.

The PDF is drawn on R's own `pdf()` device with its standard fonts, which cannot draw arrows or dashes, so those are written in ASCII (+, -). Long tables are split over pages at 0.22 inch per row, with "page i of n" labels; notes go under the last page's rows when they fit and on pages of their own otherwise. The interval column appears only when the table has `pctlo` and `pctup` (counts).

Both overwrite `file`. Errors: `x` not from `yoy()`; a `pal` not in `yoypals()`; a file that cannot be written.

Value

The path `file`, invisibly.

Examples

```
d <- data.frame(year = rep(2019:2023, each = 2),
                region = rep(c("North", "South"), 5),
                n = c(31, 402, 28, 377, 12, 190, 19, 268, 24, 331))
y <- yoy(d, value = "n", prd = "year", by = "region",
         dir = "lower")

h <- file.path(tempdir(), "change.html")
yoyhtml(y, h, title = "Placements by region")
file.exists(h)

p <- file.path(tempdir(), "change.pdf")
yoypdf(y, p, title = "Placements by region")
file.exists(p)

# a colour-blind-safe palette, no bars or intervals, and a footnote
yoyhtml(y, h, pal = "safe", bars = FALSE, intv = FALSE,
         notes = "Counts below 10 are suppressed at source.")
any(grepl("suppressed", readLines(h)))

# portrait letter pages, with fiscal-year labels
yoypdf(yoylbl(y, fylbl), p, width = 8.5, height = 11)

# only yoy() tables can be rendered
try(yoyhtml(d, h))
unlink(c(h, p))
```

`yoysmry` *Summarise a change table*

Description

Collapses a period-over-period change table from `yoy` into one row per series (or one row per group when `yoy()` was called with `by`). Each row gives the first and last observed periods, the total change between them, the compound annual growth rate, and how many period-to-period changes went up, went down or stayed flat. Use it to report the whole span of a series in one line next to the detailed table.

Usage

```
yoysmry(object, ...)

## Default S3 method:
yoysmry(object, ...)

## S3 method for class 'rmbl_yoy'
yoysmry(object, ...)
```

Arguments

`object`	An `rmbl_yoy` object returned by `yoy`.
`...`	Further arguments for methods; not used.

Details

Rows whose `value` is missing are dropped before the summary, and the remaining rows are sorted by period, text periods in the order `yoy()` read them (its `prd` and `prdord`). Periods with no order (text `yoy()` could not read, an unordered factor) have no first or last: `from`, `to`, `first`, `last`, `totalpct` and `cagrpct` are then `NA`, with a warning saying why. With v_1 the first and v_T the last observed value, the total change is

$$100(v_T/v_1 - 1)$$

and, with $s = (T-1)/\text{lag}$ steps of length `lag`,

$$100\left((v_T/v_1)^{1/s} - 1\right).$$

Both are `NA` when $v_1 \le 0$, because a growth rate out of zero does not exist; `cagrpct` is also `NA` for a single period. `"Annual"` in CAGR assumes one period is a year; for monthly or quarterly data read it as the compound rate per `lag` periods.

The `up`, `down` and `flat` counts are taken from the percent-change column (`pctchg`, or `ppchg` for `units = "percent"`). A change out of a zero base has no percent change and is not counted in any of the three. A group with no non-missing value gives a row of `NA` with `periods = 0`.

Anything other than a `yoy()` result is refused with the error "`object` must be a yoy() result".

Value

A data frame with one row per series or group and columns: the by columns (character), when `yoy()` had by; `from`, `to` (the first and last observed period, same type as the period column); `first`, `last` (numeric values at those periods); `totalpct`, `cagrpct` (numeric, percent); `periods` (integer, observed periods); `up`, `down`, `flat` (integer counts of period-to-period changes).

See Also

`yoy`, `yoypals`.

Examples

```
# one series
d <- data.frame(year = 2018:2023, n = c(120, 131, 98, 140, 155, 149))
yoysmry(yoy(d, value = "n", prd = "year"))

# one row per group; a zero base gives no percent change
g <- data.frame(year = rep(2019:2022, 2), unit = rep(c("A", "B"), each = 4),
                n = c(10, 12, 0, 15, 5, 7, 6, 9))
yoysmry(yoy(g, value = "n", prd = "year", by = "unit"))

# anything that is not a yoy() result is refused
try(yoysmry(d))
```

---

yoywr *Write a change table to a file, in whatever format the name implies*

Description

Writes a period-over-period table from `yoy()` in the format the file name implies, so one call serves a web page, a PDF for a report, a CSV for a spreadsheet, JSON for a program or Markdown for a README. Every format is written from the same object, with each row's note attached, so the number a reader sees and the number a pipeline reads are the same.

Usage

```
yoywr(x, file, format = "auto", ...)
```

Arguments

`x`	An `rmbl_yoy` object from `yoy()`.
`file`	Character scalar, the output path.
`format`	Character scalar, the output format: `"auto"` (the default) reads it from the file extension; otherwise one of `"html"`, `"pdf"`, `"csv"`, `"tsv"`, `"json"` or `"markdown"` (case-insensitive).
`...`	Passed to the format's own writer, so the colour, digit and layout options of `yoyhtml()` and `yoypdf()` are available here too.

Details

The extension is matched case-insensitively: `.html` and `.htm` go to `yoyhtml()`, `.pdf` to `yoypdf()`, `.csv` to `yoycsv()`, `.tsv` and `.tab` to `yoytsv()`, `.json` to `yoyjs()`, `.md` and `.markdown` to `yoymd()`. An existing file is overwritten. Errors when `x` does not come from `yoy()`, when the format cannot be read from the extension (pass `format`), or when `format` is not one of the names below; and whatever error the chosen writer raises.

Value

`file`, invisibly (as returned by the chosen writer).

See Also

`yoyhtml()`, `yoypdf()`, `yoycsv()`

Examples

```
d <- data.frame(year = 2019:2023, n = c(402, 377, 190, 268, 331))
y <- yoy(d, value = "n", prd = "year")

for (ext in c("csv", "tsv", "json", "md", "html")) {
  f <- file.path(tempdir(), paste0("change.", ext))
  yoywr(y, f)
  cat(ext, file.size(f), "bytes\n")
  unlink(f)
}

# an extension that names no format: say which one
f <- file.path(tempdir(), "change.txt")
yoywr(y, f, format = "csv", digits = 1)
readLines(f, n = 3)
unlink(f)
try(yoywr(y, file.path(tempdir(), "change.xlsx")))
```

Index

.Deprecated(), *368*

admits, 10
admits(), *11*, *373*
adp, 11
adp(), *10*, *12, 13*, *19, 20*, *373, 374*, *508*, *518*
adpcnts, 12
adpcnts(), *12*, *373*
advdif2d, *402*
advdif2d *(*pgsig*)*, 376
agentbndl, 13, *43*
agestd, 15, *100, 101*, *279*, *402*, *445*, *511*
aipwcf, 17, *403*
all.equal(), *455*
alos, 19
alos(), *10–12*, *507, 508*, *518*
anltab, 20, *402*, *552, 553*
anltab(), *347*, *466*
arccap, 23, *26*, *43*, *230*, *569, 570*
arccap(), *28, 29*
arcfind, 26, *27*, *43*
arcget, *26*, 27, *43*, *230*
arcpush, 28, *43*
as.character(), *29*, *534*, *582*
as.integer(), *159*
as.numeric(), *93*, *154*, *159*, *162*
asciifb, 29
asciifb(), *534*
attrfrac, 30, *419*, *562*
audcats, 32, *168*, *254*

b64, 34
bacondcmp, 35, *385*, *403*, *516*
bacondcmp(), *461*
baltab, *18*, 37, *364*, *400*, *403*
baltab(), *461*
bandsens, 38, *40*, *218*, *401*, *433*, *435*
bandsens(), *112*
bandvals, *38*, 39, *217, 218*, *433*, *435*
benfordtst, 41, *122*, *401*
benfordtst(), *69*, *137*
blcli, 42, *125*, *139*, *286*, *487*
blcli(), *423*, *462*, *588*
bldl, 46, *52*
bldl(), *173*
blfetch, *48*, 48, *229, 230*, *498*
blpal, 50, *399*, *411*
blpal(), *407*, *410*, *413, 414*
blprog, 52, *416*
blprog(), *234*, *422*
briggs *(*pgsig*)*, 376
bufasgn, 53

capdeps, 54
capenv, 56
capenv(), *54, 55*, *212*, *345*, *356*
caseflow, 57, *91*, *291*, *304*, *381*, *403*
caseflow(), *462*
cblake2b, 59, *157*
cblake2b(), *170*
cbootmean, 60, *224*
ccorr *(*cstat*)*, 160
ccorr(), *63*, *149*
ccovm, 62
ccovm(), *116*, *538*
ccrc32, 63
ccrc32(), *158*, *170*, *231*
cdgsteq *(*keyeddgst*)*, 298
cdgsteq(), *72*, *361*
cdistm *(*cspread*)*, 158
certchnvrf, 64
certchnvrf(), *67, 68*, *482, 483*, *550, 551*
certprs, 67
certprs(), *65, 66*
cgamcdf, 68
chgenv, 70
chgenv(), *435, 436*, *593, 594*
chmac *(*keyeddgst*)*, 298
chmac(), *34*, *59, 60*, *142*, *200*
chn, 71

chnapp *(chn)*, 71
chnhead *(chn)*, 71
chnkfile, 74
chnkfile(), *136*, *361*
chnnew *(chn)*, 71
chnseal *(chn)*, 71
chnseal(), *118*, *551*
chnvrf *(chn)*, 71
chnvrf(), *71, 72*
chwkfit, 75
chwkfit(), *79–84*, *409, 410*, *462*
chwkjit, 79
chwkjit(), *77, 78*, *462*
chwknll, 80
chwkres, 82
chwkres(), *77, 78*, *409*
chwksim, 84
chwksim(), *462*
cipw, 85
cipw(), *166*
ciqr *(crob)*, 154
citecps, 86
clncolnms, 88
clrdcmp, 89, *291*, *403*
clrdcmp(), *462*
cmad *(crob)*, 154
cmad(), *94*, *159*, *162*
cmed *(crob)*, 154
cmed(), *162*
cmidrk *(crank)*, 148
cmlrisk, 91
cmn *(cstat)*, 160
cmom, 93
cmom(), *63*, *95*, *155*, *159*, *162*, *166*, *365, 366*, *429*
cnormlpdf, 94
cnormpdf, 96, *161*
cnormpdf(), *95*
cnscwb, 97, *99*, *402*
cnsprof, *97*, 98, *100*, *280*, *430, 431*
cnsstd, *99*, 100, *511*
cnttrnd, 101, *402*, *513*, *545*
cnttrnd(), *22*, *461*
co2fp, *43*, 103, *123*, *243*, *402*, *441, 442*
co2int, *103–105*, 106, *206*
co2inttab *(co2int)*, 106
cohtrnd, 107, *403*
cohtrnd(), *461*
commidx, 109, *402*
concentration, 111
connassoc, 113, *402*
cortbl, 115, *401*
cortbl(), *461*, *538*
cpsatst, 116
cpsatst(), *120, 121*, *129*, *276*, *344*, *352*, *432, 433*, *551*
cpsbndl, 119, *135*, *138, 139*, *353*
cpsbndl(), *276*, *448*
cpsbndlrd *(cpsbndl)*, 119
cpsbndlrd(), *276*
cpsbndlvrf, *43*, *122*
cpsbndlvrf *(cpsbndl)*, 119
cpsbndlvrf(), *119*, *276*, *302, 303*, *588*
cpschk, *43*, 122, *138, 139*, *251, 252*
cpschk(), *302, 303*, *462*
cpschkatst *(cpsatst)*, 116
cpschkatst(), *120*, *276*
cpsdrft, *43*, *122*, 125, *187*, *401*
cpsdrft(), *22*, *42*, *136, 137*, *185–187*, *189–196*, *198*, *284*, *429*, *462*, *555, 556*
cpsfals, 128, *402*
cpsfals(), *118*, *131–133*, *214*, *226, 227*, *355, 356*, *433*, *447*
cpsplant, 130
cpsplant(), *143*, *447*
cpspwr, 132, *401*
cpspwr(), *129*
cpsrfrsh, *43*, *123–125*, 134, *138*, *352, 353*
cpsrfrsh(), *462*
cpsrpt, 135
cpsrpt(), *467, 468*
cpsseal, *44*, 138
cpssign, *45*, *138*, 139, *446*
cpssign(), *73*, *117, 118*, *136*, *144–146*, *200*, *232–239*, *298*, *423*, *439*, *585, 586*
cpsspec, 142
cpsspec(), *131*, *447*
cpsvrf, 144, *580*
cpsvrf(), *137*, *139*, *142*, *145, 146*, *232, 233*, *235*, *239*, *423*, *495*
cpsxvrf, 145
cpsxvrf(), *235*, *237*
cqnt *(crob)*, 154
cqnt(), *94*
cramerv, 147

crank, 148
crc32file *(filedgst)*, 231
crc32file(), *64*
crfno2, *419*, *562*
crfno2 *(crfpm25)*, 150
crfno2(), *462*
crfpm25, *31*, 150, *402*, *419*, *562*
crfpm25(), *462*
crmdens, 152, *291*, *301*, *403*
crmdens(), *462*
crob, 154
csdev *(cspread)*, 158
csdev(), *94*
csha256, 156, *161*, *255*, *446*
csha256(), *34*, *59, 60*, *64*, *157, 158*, *170*, *289*, *298*, *492*
csha512, 157
csha512(), *60*, *170*, *231*
cspr *(crank)*, 148
cspr(), *538*
cspread, 158
cstat, *96*, 160
ctkyfnc *(crob)*, 154
ctkyfnc(), *328*
ctrm, 161
ctrmmean *(ctrm)*, 161
ctrmmean(), *155*
cunif, 163, *378*
cunif(), *79, 80*
cvari *(cstat)*, 160
cwbscr, *97, 98*, 164
cwinsmean *(ctrm)*, 161
cwtd, 166
cwtd(), *86*

datumpmap *(ltrlrng)*, 321
datumpmap(), *462*
deccodes, *32, 33*, 167, *169*
declbld, *32*, *168*, 168, *450*, *584*
dgstobj, *157*, 170
dgstobj(), *137*
dispmix, *58*, 171, *291*, *403*
dispmix(), *462*
dldata, 173
dldata(), *228*
dload, *174*, *290*, *489*
dload *(dmnf)*, 178
dload(), *186*, *462*
dlogin, *24*, *26*, 174, *177–179*, *282*, *489*, *521*
dlogout, *176*, 177
dmnf, 178, *521*
dnorm, *96*
drbggen, 181
drbggen(), *183, 184*
drbgnew, 182
drbgnew(), *181*, *184*, *438*
drbgrsd, 184
drbgrsd(), *181*, *183*
drftapp, *43*, *125*, 185
drftapp(), *127*, *193*, *462*
drftcal, 186, *401*, *484*
drftcal(), *127*
drftchisq, 188
drftchisq(), *42*, *69*, *126*, *191*, *195*
drfthomog, 190
drfthomog(), *126*, *188, 189*
drfthtml, *43*, *125*, 192
drfthtml(), *127*, *185, 186*
drftks, 194
drftks(), *126*, *188*, *196*
drftpsi, *123*, 195
drftpsi(), *126*, *195*
drp, 197
drpconst *(drp)*, 197
drpempty *(drp)*, 197
drvkey, 199
drvkey(), *465*
dstfit, 200, *348*
dstfit(), *407*, *461*
dstn, 202
dstncnt *(dstn)*, 202
dstnskch *(dstn)*, 202
dstntst, 204, *279*, *402*, *431*
dstntst(), *462*
dtabs *(dmnf)*, 178
dtctloc, *105, 106*, 206
dualcont, *114*, 207, *246*, *262*, *402*
dualcont(), *462*
duprows, 209

ebrates, 210, *218, 219*, *242*, *509*
envdiff, 212
envdiff(), *55, 56*
evlrr, 213
evstudy, *54*, 215, *385*, *402*, *408*, *516*, *546*
evstudy(), *461*
expbands, *40*, 217, *433*, *435*
expcnts, *210*, 218, *508, 509*

expconcidx, 220, *402*, *418*, *562*
explerr, *43*, *124*, *125*, 221
explerr(), *461*
exprspplr, 223, *416*
exprspplr(), *61*
exptrnd, 225, *250*, *402*

falsfam, 226, *484*
falsfam(), *433*
fdl, *135*, 228, *552*
fdl(), *173*, *572*
fetchvrf, *23–27*, 229
fetchvrf(), *29*
filedgst, 231
fipskey, *138*, 232
fipskey(), *234*, *238–240*, *421*
fipskg, *44*, *45*, *138*, 233, *446*
fipskg(), *117*, *118*, *120*, *140*, *141*, *144*, *146*, *232*, *235*, *236*, *238–240*, *275*, *276*, *296*, *368*, *369*, *421–423*, *495*
fipsmkg, 235
fipsmkg(), *140*, *236*, *237*
fipsmod, 236
fipsmod(), *235*, *236*
fipsmu, 238
fipspubkey *(*fipskg*)*, 233
fipspubkey(), *117*, *144–146*, *368*, *495*
fipssignmu *(*fipsmu*)*, 238
fipssz, 240
fipssz(), *232–235*, *421*
fipsvrfmu *(*fipsmu*)*, 238
fnllim, *211*, *218*, *219*, 241, *410*, *411*, *509*
format.rmbl_band_sensitivity *(*bandsens*)*, 38
format.rmbl_cpschk *(*cpschk*)*, 122
format.rmbl_dstfit *(*dstfit*)*, 200
format.rmbl_ledger *(*ldgrread*)*, 301
format.rmbl_ledger_check *(*ldgrread*)*, 301
format.rmbl_mnfdiff *(*mnfdiff*)*, 352
format.rmbl_yoy *(*yoy*)*, 588
fpeqv, 243
freqtbl, 244, *401*
freqtbl(), *461*
frmaud, 245
fromjs, 247
fromjs(), *287*, *288*, *319*, *537*
fylbl, 249
fylbl(), *597*

genexpo, *226*, 250
ghann, *122*, *125*, 251
ghcmt, *122*
ghcmt *(*ghann*)*, 251
ghout, *122*
ghout *(*ghann*)*, 251
ginicoef, *38*, *217*, *411*
ginicoef *(*concentration*)*, 111
gplume, *382*, *383*
gplume *(*pgsig*)*, 376
gplume(), *462*, *464*
grdbin, 252
grdlvls, *33*, 253, *255*
grdrcd, *168*, *169*, *253*, *254*, 254, *450*, *564*
grepl(), *470*
gspuff *(*pgsig*)*, 376

healchg, 256, *402*, *541*
healchg(), *462*
hilltlidx, 258
hilltlidx(), *112*, *273*
holbal, *110*, *208*, *257*, 260, *332*, *402*
hotpers, 263, *403*
hqccmpkey, 265
hqccmpkey(), *267*, *270*
hqcdecap, 267
hqcdecap(), *265*, *266*, *268*, *269*, *271*
hqcencap, 268
hqcencap(), *267*, *269*, *271*, *273*
hqckg, 269
hqckg(), *265–269*, *272*, *273*
hqcpubkey *(*hqckg*)*, 269
hqcpubkey(), *268*, *269*
hqcsz, 272
hqcsz(), *266*, *268–271*
hurwzeta, 273
hurwzeta(), *259*

idcrd, 275
idcrd(), *439*
idcrdrd *(*idcrd*)*, 275
idcrdvrf *(*idcrd*)*, 275
indgap, *205*, 277, *306*, *371*, *372*, *402*, *431*, *437*, *485*, *486*, *490*, *511*, *540*, *541*, *577*
indgap(), *462*
indiex, *99*, 280
indiex(), *462*, *463*
indtabs, *281*, 282, *490*

infsch, 282
infsch(), *136*, *480*, *555*, *556*
inlhist, 284
inlhist(), *429*
instcli, *44*, *46*, 286

jsb64dec *(*b64*)*, 34
jsb64enc *(*b64*)*, 34
jsb64enc(), *288*
jsb64udec *(*b64*)*, 34
jsb64uenc *(*b64*)*, 34
jsgz, 287
jsgzdec *(*jsgz*)*, 287
jsgzenc *(*jsgz*)*, 287
jsgzenc(), *35*
jsser *(*jsserdoc*)*, 289
jsser(), *170*, *288*, *535–537*
jsserdoc, 289
jsunser *(*jsserdoc*)*, 289
justice-data, 290

kemdecap, 292
kemdecap(), *293*, *294*, *296*
kemencap, 293
kemencap(), *292–294*, *296*, *297*
kemkg, 294
kemkg(), *270*, *271*, *292–294*, *297*
kempubkey *(*kemkg*)*, 294
kempubkey(), *293*, *294*
kemsz, 296
kemsz(), *273*, *292*, *294–296*
keyeddgst, 298
knoxtst, *153*, *291*, 299, *403*
knoxtst(), *462*

lagrprt, *402*
lagrprt *(*pgsig*)*, 376
ldgrchk *(*ldgrread*)*, 301
ldgrchk(), *462*, *464*
ldgrread, 301
ldgrread(), *462*, *464*
lifetbl, *58*, *291*, 303, *403*
lifetbl(), *462*
limfrom, *278*, *279*, 305, *437*
llmask, *14*, *174*, *223*, 306, *309–315*, *317*, *520*
llmask(), *14*
llmcfg, *42*, 309
llmchat, *308*, 311, *315*
llmmdls, 313
llmprvd, *306*, *307*, *309–311*, *313*, 314, *315*, *316*, *318*
llmrte, *308*, *312*, *313*, *315*, 315, *318*
llmstat, *306*, *308*, *310*, *313*, *315*, *316*, 317
lm.fit, *416*
loadprov, 318, *552*, *553*
loadprov(), *86*, *192*, *469–472*, *488*, *555*, *565*
loounit, 319, *403*
loounit(), *461*
lorenzc, *410*, *411*
lorenzc *(*concentration*)*, 111
lorenzc(), *412*
ltrlrng, 321
ltrlrng(), *462*

mahaoutl, 327, *401*
mahaoutl(), *137*, *461*
mcartst, 329
mean(), *590*
mhcont, 331, *402*
mhcont(), *462*
missgrid, 333
missgrid(), *461*
misspat, 334, *401*
misspat(), *330*, *334*, *337*, *338*
missruns, 336
missruns(), *137*, *334*
misssmry, 337
misssmry(), *22*, *136*, *330*, *337*
mkbndl, *43*, 338
mkbndl(), *341–344*
mkbndlchk, *43*, 341
mkbndlproj, *43*, 343
mkbndlproj(), *339*, *340*, *343*
mkmnf, 344
mkmnf(), *117*, *120*, *212*, *339*, *340*, *351*, *352*, *354*, *356*, *447–449*, *578*, *581*
mkset, 346
mksyncol, 347, *349*, *350*
mksyncol(), *202*, *461*, *463*, *464*
mksyncsv, *347*, *348*, 349
mksyncsv(), *558*
mnfcanon, 351
mnfcanon(), *117*, *536*, *578*
mnfdgst *(*mnfcanon*)*, 351
mnfdgst(), *117*, *118*, *120*, *121*, *344*, *345*, *351*, *578*
mnfdiff, *43*, *124*, *135*, 352
mnfdiff(), *186*, *192*, *193*

mnfrcmp, *124*, 354
mnfrecseed, 356
mnfrstseed *(*mnfrecseed*)*, 356
morani, 357
mortdisp, 359, *419*, *562*
mrk, 360
mrklvs *(*mrk*)*, 360
mrklvs(), *74*
mrkprf *(*mrk*)*, 360
mrkprf(), *74*
mrkroot *(*mrk*)*, 360
mrkroot(), *73*, *74*, *136*
mrkvrf *(*mrk*)*, 360
mtchdid, 362
mtchdid(), *461*

na.omit, *161*

onl, 365
onlsmry *(*onl*)*, 365
optcrcsrch *(*ltrlrng*)*, 321
optcrcsrch(), *462*
opteffort *(*ltrlrng*)*, 321
opteffort(), *462*
optsqr, 367
oqskg, 368
oqspubkey *(*oqskg*)*, 368
orchk, 369
ordcmp, 371, *402*, *485*, *486*, *577*
ordcmp(), *462*

p.adjust, *483*
perdays, *11*, 373
perdays(), *517*
perminf, 374, *385*, *403*, *519*
perminf(), *461*
pgsig, 376
piperate, 380
plmfld, *153*, 382, *402*
plmfld(), *462*
plndid, *36*, *216*, *319*, *364*, *375*, *381*, 383, *389*, *391*, *392*, *491*, *516*, *573*
plndid(), *461*
plndml, *17*, *18*, 386
plnpois, *385*, 388
plnpwr, 390, *402*
plnpwr(), *461*
plot(), *127*, *193*
plot-methods *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_analysis *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_benford *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_correlations *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_cortable *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_drift *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_falsification *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_freq *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_missingness *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_outliers *(*plot.rmbl_census_cwb*)*, 393
plot.bricklayer_power *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_aipw *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_alloc *(*ralloc*)*, 438
plot.rmbl_bacon *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_balance *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_band_sensitivity *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_bounds *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_burden *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_case_flow *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_census_cwb, 393
plot.rmbl_clearance_decomposition *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_cohort_trends *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_community_index *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_connection *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_count_trend *(*plot.rmbl_census_cwb*)*, 393
plot.rmbl_crf *(*plot.rmbl_census_cwb*)*, 393

plot.rmbl_datummap
(plot.rmbl_census_cwb), 393
plot.rmbl_density
(plot.rmbl_census_cwb), 393
plot.rmbl_disposition_mix
(plot.rmbl_census_cwb), 393
plot.rmbl_distinctions
(plot.rmbl_census_cwb), 393
plot.rmbl_drift_calibration
(plot.rmbl_census_cwb), 393
plot.rmbl_dsr (plot.rmbl_census_cwb),
393
plot.rmbl_dstfit, 407
plot.rmbl_dstfit(), *201*, *202*
plot.rmbl_dual_continuum
(plot.rmbl_census_cwb), 393
plot.rmbl_equity
(plot.rmbl_census_cwb), 393
plot.rmbl_event_study
(plot.rmbl_census_cwb), 393
plot.rmbl_exposure_trend
(plot.rmbl_census_cwb), 393
plot.rmbl_field (plot.rmbl_census_cwb),
393
plot.rmbl_footprint
(plot.rmbl_census_cwb), 393
plot.rmbl_healing_change
(plot.rmbl_census_cwb), 393
plot.rmbl_holistic_balance
(plot.rmbl_census_cwb), 393
plot.rmbl_hotspot
(plot.rmbl_census_cwb), 393
plot.rmbl_indigenous_gap
(plot.rmbl_census_cwb), 393
plot.rmbl_knox (plot.rmbl_census_cwb),
393
plot.rmbl_life_table
(plot.rmbl_census_cwb), 393
plot.rmbl_loo (plot.rmbl_census_cwb),
393
plot.rmbl_ltrlrng
(plot.rmbl_census_cwb), 393
plot.rmbl_opteffort
(plot.rmbl_census_cwb), 393
plot.rmbl_ordinal_compare
(plot.rmbl_census_cwb), 393
plot.rmbl_particles
(plot.rmbl_census_cwb), 393
plot.rmbl_permutation
(plot.rmbl_census_cwb), 393
plot.rmbl_pollution_report
(plot.rmbl_census_cwb), 393
plot.rmbl_power (plot.rmbl_census_cwb),
393
plot.rmbl_prevalence
(plot.rmbl_census_cwb), 393
plot.rmbl_profile_share
(plot.rmbl_census_cwb), 393
plot.rmbl_rate (plot.rmbl_census_cwb),
393
plot.rmbl_rate_change
(plot.rmbl_census_cwb), 393
plot.rmbl_region_coverage
(plot.rmbl_census_cwb), 393
plot.rmbl_share (plot.rmbl_census_cwb),
393
plot.rmbl_srchpod
(plot.rmbl_census_cwb), 393
plot.rmbl_staggered_att
(plot.rmbl_census_cwb), 393
plot.rmbl_stock_flow
(plot.rmbl_census_cwb), 393
plot.rmbl_svy_compare
(plot.rmbl_census_cwb), 393
plot.rmbl_svy_estimate
(plot.rmbl_census_cwb), 393
plot.rmbl_synth (plot.rmbl_census_cwb),
393
plot.rmbl_synthdid
(plot.rmbl_census_cwb), 393
plot.rmbl_trc_gaps
(plot.rmbl_census_cwb), 393
plot.rmbl_trc_progress
(plot.rmbl_census_cwb), 393
plot.rmbl_wellness_profile
(plot.rmbl_census_cwb), 393
plot.rmbl_yoy (plot.rmbl_census_cwb),
393
plotbump (plotseries), 414
plotbump(), *461*
plotcal (plotseries), 414
plotcal(), *461*
plotecdf (plotmore), 412
plotecdf(), *461*
plotevgrid, *403*, 408
plotfnl (plotlrnz), 410

plotgain *(plotmore)*, 412
plotgain(), *461*
plothex *(plotmore)*, 412
plothex(), *461*
plothrz *(plotseries)*, 414
plothrz(), *461*
plothwk, 409
plothwk(), *84*, *462*
plotlrnz, 410
plotlrnz(), *111*
plotmore, 412
plotopt, *393*
plotopt *(blpal)*, 50
plotrdg *(plotmore)*, 412
plotrdg(), *461*
plotseries, 414
plotsm *(plotseries)*, 414
plotsm(), *461*, *462*
plotvln *(plotmore)*, 412
plotvln(), *461*
plrcf, *18*, *223*, *224*, *388*, 415
polbrdarea, 417, *418*
polbrdn, *30*, *31*, *150*, *360*, *402*, *417*, *418*, 418, *562*, *563*
polbrdn(), *462*
polex, 420
polrpttxt *(vrfpol)*, 561
pqcbknd, 421
pqcbknd(), *233*, *240*
pqckg, *44*, *45*, *138*, 422, *446*
pqckg(), *117*, *118*, *120*, *140*, *141*, *233*, *234*, *240*, *275*, *276*, *421*, *465*, *495*, *585*, *588*
print.bricklayer_attestation, 424
print.bricklayer_attestation_check *(print.bricklayer_attestation)*, 424
print.bricklayer_benford *(print.bricklayer_attestation)*, 424
print.bricklayer_bundle *(print.bricklayer_attestation)*, 424
print.bricklayer_bundle_check *(print.bricklayer_attestation)*, 424
print.bricklayer_certificate *(print.bricklayer_attestation)*, 424
print.bricklayer_certpath_check *(print.bricklayer_attestation)*, 424
print.bricklayer_chain *(print.bricklayer_attestation)*, 424
print.bricklayer_drbg *(print.bricklayer_attestation)*, 424
print.bricklayer_drift *(print.bricklayer_attestation)*, 424
print.bricklayer_env_diff *(print.bricklayer_attestation)*, 424
print.bricklayer_falsification *(print.bricklayer_attestation)*, 424
print.bricklayer_falsify_family *(print.bricklayer_attestation)*, 424
print.bricklayer_fips_key *(print.bricklayer_attestation)*, 424
print.bricklayer_fips_public_key *(print.bricklayer_attestation)*, 424
print.bricklayer_hqc_capsule *(print.bricklayer_attestation)*, 424
print.bricklayer_hqc_key *(print.bricklayer_attestation)*, 424
print.bricklayer_hqc_public_key *(print.bricklayer_attestation)*, 424
print.bricklayer_kem_capsule *(print.bricklayer_attestation)*, 424
print.bricklayer_kem_key *(print.bricklayer_attestation)*, 424
print.bricklayer_kem_public_key *(print.bricklayer_attestation)*, 424
print.bricklayer_mcar *(print.bricklayer_attestation)*,

424
print.bricklayer_oqs_key
(print.bricklayer_attestation), 424
print.bricklayer_oqs_public_key
(print.bricklayer_attestation), 424
print.bricklayer_power
(print.bricklayer_attestation), 424
print.bricklayer_prereg
(print.bricklayer_attestation), 424
print.bricklayer_prereg_check
(print.bricklayer_attestation), 424
print.bricklayer_public_key
(print.bricklayer_attestation), 424
print.bricklayer_recode_manifest
(print.bricklayer_attestation), 424
print.bricklayer_recompute
(print.bricklayer_attestation), 424
print.bricklayer_report
(print.bricklayer_attestation), 424
print.bricklayer_schema
(print.bricklayer_attestation), 424
print.bricklayer_signature
(print.bricklayer_attestation), 424
print.bricklayer_signing_key
(print.bricklayer_attestation), 424
print.bricklayer_timestamp
(print.bricklayer_attestation), 424
print.rmbl_band_sensitivity (bandsens), 38
print.rmbl_bounds
(print.bricklayer_attestation), 424
print.rmbl_cpschk (cpschk), 122
print.rmbl_dstfit (dstfit), 200
print.rmbl_fetch
(print.bricklayer_attestation), 424
print.rmbl_ledger (ldgrread), 301
print.rmbl_ledger_check (ldgrread), 301
print.rmbl_margin_check
(print.bricklayer_attestation), 424
print.rmbl_mnfdiff (mnfdiff), 352
print.rmbl_or_check
(print.bricklayer_attestation), 424
print.rmbl_recode_manifest_check
(print.bricklayer_attestation), 424
print.rmbl_transfer_check
(print.bricklayer_attestation), 424
print.rmbl_yoy (yoy), 588
printmeth
(print.bricklayer_attestation), 424
profcols, 428
profcols(), *22*, *136, 137*, *198*, *245*, *284, 285*, *335*, *338*, *461*
profshr, *99*, *402*, 430
prrgchk (prrgdecl), 432
prrgchk(), *432*
prrgdecl, 432
prrgdecl(), *227*, *439*, *549*
prsbands, *38–40*, *217*, *303, 304*, 433
pswppod (ltrlrng), 321
pswppod(), *462*
pubbnd, *402*, 435
pubbnd(), *21, 22*, *70*, *593, 594*

qltyrcd, *205*, *278, 279*, *306*, *371*, 437, *510*

ralloc, 438
ralloc(), *440, 441*, *548, 549*
rallocvrf, 440
rallocvrf(), *438, 439*
raplok, *104*, 441
ratechg, *102*, *401*, 442, *476*, *599*
ratechg(), *21, 22*, *461*, *483*
rategap, *16*, 445
rcdmnf, *255*, 446, *564*, *579*
recevid, 447
recevid(), *131*, *143*
recrd, 448

recrd(), *339*, *340*, *343–345*, *354*, *355*, *448*, *578*, *581*
relbl, *169*, *255*, 450, *460*
rgncov, *401*, 451
rgncov(), *457*
rgnmapalt, 452
rgnmapalt(), *452*, *455*, *457*, *458*
rgnmapcmp, 454
rgnmapcmp(), *452*, *453*, *457*, *458*
rgnmapintg, 456
rgnmapintg(), *452*, *453*, *455*, *462*
rgnmappts, 457
rgnmappts(), *454*
rlblfrns, *370*, *450*, 459, *560*, *583*, *584*
rmblapp, *42*, 461
rmblapp(), *347*, *463*, *464*
rmbldemo, 463
rmbldemo(), *461*, *462*
rndbytes, 464
rndbytes(), *182*, *183*, *199*, *200*, *268*, *294*, *295*, *422*, *439*
rptanl, 466
rptanl(), *22*, *347*
rptmd, 467
rptmd(), *135*, *137*
rslvarc, 468
rslvarc(), *318*
rslvckan, 469
rslvckan(), *173*, *318*, *471*
rslvckns, 471
rslvckns(), *470*
rslvsoc, 472
rslvsoc(), *318*
rsv, 473
rsvidx *(*rsv*)*, 473
rsvsmp *(*rsv*)*, 473
rtcalc, *401*, *444*, 475, *493*, *494*
rtcalc(), *21*, *22*, *461*
rul, 477
rul(), *136*, *478*, *480*, *488*, *553–555*
rulebtw *(*rulelib*)*, 478
rulecmpl *(*rulelib*)*, 478
rulecolcnt *(*rulelib*)*, 478
ruledstn *(*rulelib*)*, 478
ruledstn(), *210*
ruleincr *(*rulelib*)*, 478
ruleinset *(*rulelib*)*, 478
rulelib, 478
rulemad *(*rulelib*)*, 478
rulemad(), *328*
rulenull *(*rulelib*)*, 478
rulerx *(*rulelib*)*, 478
ruleuniq *(*rulelib*)*, 478
ruleuniq(), *210*
rvkfetch, 481

saveRDS(), *440*
scanadj, *187*, 483, *509*, *598*, *599*
scanadj(), *21*, *22*, *127*
scests, *261*, *371*, *372*, 484, *541*, *576*, *577*
schemapath, *43*, *122*, 487
schemapath(), *303*
schval, 487
schval(), *477*, *553–556*
sctidy, *485*, *486*, 489
set.seed(), *61*, *465*
sevidx, 490
sha256file, *230*, 491, *552*
sha256file(), *120*, *231*, *344*, *565*
sha512file *(*filedgst*)*, 231
sha512file(), *74*, *158*
shiny::runApp(), *186*, *462*
shiny::shinyApp(), *461*
shrs, *172*, *401*, *476*, 493
shrs(), *461*
signpubkey, 495
signpubkey(), *117*, *145*, *423*
siudate, 496
siudate(), *500*
siufetch, 497
siufetch(), *498*, *500*
siufprs, *497*, *498*, 498
siuprs, 499
siuprs(), *462*, *498*, *499*, *501*, *503*
siusch, 501
siusch(), *499*
siuso, 502
siuso(), *462*, *500*, *503*
siutxt, 503
siutxt(), *500*, *502*
skchmrg *(*dstn*)*, 202
smrymrg *(*onl*)*, 365
smrystat *(*onl*)*, 365
smryupd *(*onl*)*, 365
sprd, 504
srchfig, *325*, *367*, 506
srchfig(), *462*

srchpod, *367*
srchpod *(*ltrlrng*)*, 321
srchpod(), *462*
srchscs *(*ltrlrng*)*, 321
stats::chisq.test(), *189–191*
stats::cor(), *148*
stats::cov(), *62*
stats::density(), *413*
stats::dnorm(), *95*
stats::ks.test(), *194*
stats::mad(), *155*
stats::na.omit(), *159*
stats::optim(), *81*
stats::pgamma(), *69*
stats::quantile(), *155*
stats::sd(), *159*
stats::var(), *93*, *166*
staysmry, 507
staysmry(), *19*
stdinc, *210, 211*, *218, 219*, *242*, *476*, 508
stdprev, *100, 101*, *402*, *485*, 510
stdprev(), *462*
stepchg, 512, *545*
stepchg(), *461*
stgatt, *36*, *216*, *385*, *388*, *402*, *408*, 514
stgatt(), *461*
stkflow, *58*, *401*, 516
stkflow(), *10–12*, *20*, *373*, *462*, *508*
stpdnpv, *375*, 518
sum(), *590*
summary.bricklayer_benford *(*print.bricklayer_attestation*)*, 424
summary.bricklayer_drift *(*print.bricklayer_attestation*)*, 424
summary.bricklayer_report *(*print.bricklayer_attestation*)*, 424
svcs, *174*, *176*, *178, 179*, 520
svycmp, *403*, 521
svycmp(), *461*, *523*, *526*, *528*
svydes, *17, 18*, *226*, *505*, 523
svydes(), *461*, *522*, *527, 528*
svyest, *403*, 526
svyest(), *461*, *522, 523*, *525, 526*
swpwdth *(*ltrlrng*)*, 321
swtdid, 528
swtdid(), *461*
synctl, *402*, 530
syndid, *402*, 532
syndid(), *461*

toascii, 534
toascii(), *29, 30*, *582*
tojs, 535
tojs(), *248*, *287–289*, *578*
topcors, *401*, 537
topcors(), *115, 116*, *327*
topshr *(*concentration*)*, 111
trccmpl, 539, *543*
trchlthgap, 540, *542*
trcind, *540, 541*, 542
trcprog, *402*, *539*, *542*, 543
trcprog(), *462*
trndtst, *102*, *513*, 544
trndtst(), *22*, *461*
tse, 546
tse(), *461*
tsinfo *(*tsvrf*)*, 549
tsstmp, 548
tsstmp(), *439*
tsvrf, *24, 25*, 549
tsvrf(), *66*, *68*, *439*, *483*, *548, 549*

usecpstmpl, *134, 135*, 552

validUTF8(), *30*
valrules, 553
valrules(), *137*, *477–480*, *487, 488*, *555*
valsch, *124*, *135*, 555
valsch(), *127*, *137*, *282–284*, *318, 319*, *478*, *487, 488*, *553, 554*, *558*
var, *160*
vrfcps, *43*, *122*, *134, 135*, *222*, *350*, *487*, *552, 553*, 557
vrfcps(), *121*, *302*, *354, 355*, *462*
vrfmrg, *370*, *446*, *460*, 559, *583, 584*
vrfpol, *150*, *360*, *402*, *419*, 561
vrfpol(), *462*
vrfrcd, *255*, *446*, 564, *580*
vrfrcdmnf, *446*
vrfrcdmnf *(*wrrcdmnf*)*, 579
vrfsha256, 565
vrfsha256(), *173*, *318*, *558*

wbrtry, *43*, 566, *570*

wbrtry(), *29*
wbsave, *24*, *25*, *43*, *124*, *230*, *566*, *567*, 568
wbsave(), *29*
wbsnpnat, *570*, 571
wbsnpnat(), *49*
wbsnpurl, *49*, *124*, *571*, 572
wbsnpurl(), *228*, *571*
wcb, *216*, *320*, *363*, *375*, *384*, *387*, *389*, *391*, *515*, *529*, 573, *574*
wcb(), *461*
welprof, *207*, *208*, *245*, *246*, *257*, *262*, *332*, *402*, *541*, 575
welprof(), *462*
wrmnfjs, 578
wrmnfjs(), *56*, *339*, *344*, *345*, *352*, *448*, *558*, *580*, *581*
wrrcdmnf, *446*, 579
wrsmrytxt, 580
wrsmrytxt(), *344*, *345*, *448*, *449*, *578*
wrtxtfb, 582
wrtxtfb(), *534*

xfrvrf, *33*, *169*, *560*, 583
xmssadv, 585
xmssadv(), *141*, *422*, *423*
xmssfgt *(*xmssadv*)*, 585
xmssfgt(), *585*, *586*
xmssidxseen, *45*, *123*, 587
xmssidxseen(), *120*
xmssregkey *(*xmssadv*)*, 585
xmssregkey(), *141*, *422*, *585*, *586*

yoy, *401*, *443*, *444*, *494*, 588, *597–599*, *602*, *603*
yoy(), *21*, *22*, *346*, *443*, *461*, *483*, *593–596*, *599–601*, *603*, *604*
yoybnd, 593
yoybnd(), *22*, *70*, *435*, *436*
yoycsv *(*yoydelim*)*, 594
yoycsv(), *604*
yoydelim, 594
yoyhtml, *597*, *598*
yoyhtml *(*yoyren*)*, 599
yoyhtml(), *461*, *462*, *592*, *596*, *597*, *603*, *604*
yoyjs *(*yoydelim*)*, 594
yoyjs(), *604*
yoylbl, 596
yoylbl(), *249*, *601*
yoymd *(*yoydelim*)*, 594
yoymd(), *604*
yoypals, *591*, 597, *603*
yoypals(), *600*, *601*
yoypdf, *597*, *598*
yoypdf *(*yoyren*)*, 599
yoypdf(), *592*, *596*, *597*, *603*, *604*
yoypv, *484*, 598
yoyren, 599
yoysmry, *598*, 602
yoysmry(), *591*, *592*, *594*, *599*
yoytsv *(*yoydelim*)*, 594
yoytsv(), *604*
yoywr, 603
yoywr(), *591*