Package 'snapKrig'

January 9, 2023

Title Fast Kriging and Geostatistics on Grids with Kronecker Covariance

Version 0.0.1

Description Geostatistical modeling and kriging with gridded data using spatially separable covariance functions (Kronecker covariances). Kronecker products in these models provide shortcuts for solving large matrix problems in likelihood and conditional mean, making 'snapKrig' computationally efficient with large grids. The package supplies its own S3 grid object class, and a host of methods including plot, print, Ops, square bracket replace/assign, and more. Our computational methods are described in Koch, Lele, Lewis (2020) <doi:10.7939/r3-g6qb-bq70>.

License MIT + file LICENSE

URL https://github.com/deankoch/snapKrig

BugReports https://github.com/deankoch/snapKrig/issues

Imports graphics, grDevices, methods, stats, utils

Suggests knitr, rmarkdown, terra, raster, sf, units, sp,

VignetteBuilder knitr

Encoding UTF-8

Language en-US

RoxygenNote 7.2.3

NeedsCompilation no

Author Dean Koch [aut, cre, cph] (<https://orcid.org/0000-0002-8849-859X>)

Maintainer Dean Koch <dkoch@ualberta.ca>

Repository CRAN

Date/Publication 2023-01-09 09:50:02 UTC

R topics documented:

`anyNA.sk` *Check for presence of grid points with missing data (NAs)*

Description

Returns a logical indicating if any of the grid points are NA

Usage

```
## S3 method for class 'sk'
anyNA(x, recursive)
```

Arguments

x	a sk object
recursive	ignored

Value

logical

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
anyNA(g)
g[1] = NA
anyNA(g)
```

---

as.double.sk *Coerce grid values to numeric (double type)*

Description

This also adds support for as.numeric

Usage

```
## S3 method for class 'sk'
as.double(x, ...)
```

Arguments

x	a sk object
...	further arguments to as.double

Value

an "sk" object with numeric data values

Examples

```
g = sk_validate(list(gval=sample(c(FALSE, TRUE), 4^2, replace=TRUE), gdim=4, gres=0.5))
g[]
as.numeric(g)[]
```

as.integer.sk *Coerce grid values to integer*

Description

Coerce grid values to integer

Usage

```
## S3 method for class 'sk'
as.integer(x, ...)
```

Arguments

x	a sk object
...	further arguments to as.integer

Value

an "sk" object with integer data values

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
g[]
as.integer(g)[]
```

as.logical.sk *Coerce grid values to logical*

Description

Coerce grid values to logical

Usage

```
## S3 method for class 'sk'
as.logical(x, ...)
```

Arguments

x	a sk object
...	further arguments to as.logical

Value

a "sk" object with logical data values

Examples

```
g = sk_validate(list(gval=sample(c(0,1), 4^2, replace=TRUE), gdim=4, gres=0.5))
g[]
as.logical(g)[]

# "range" for logical is reported as integer
summary(as.logical(g))
```

`as.matrix.sk` *convert to matrix*

Description

Returns a matrix representation of the grid data. This is shorthand for extracting the data using `x[]` (single layer) or `x[,j]` (multi-layer), then passing the result to `matrix` along with `dim(x)`.

Usage

```
## S3 method for class 'sk'
as.matrix(x, rownames.force = NA, layer = 1, ...)
```

Arguments

`x`	a sk object
`rownames.force`	ignored
`layer`	integer, for multi-layer grids, the layer number to return
`...`	further arguments to as.matrix

Value

the grid data as a matrix

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
plot(g)
as.matrix(g)
```

as.vector.sk *Convert grid data to vector of specified mode*

Description

Returns a vector of the specified mode, representing the vectorized grid data. For multi-layer x, the first layer is returned.

Usage

```
## S3 method for class 'sk'
as.vector(x, mode = "any")
```

Arguments

x	a sk object
mode	passed to as.vector

Details

For single layer x, and with default mode='any', this is the same as x[]

Value

a vector of the specified mode

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
as.vector(g)
```

dim.sk *Grid dimensions*

Description

Returns gdim, the number of y and x grid lines, in that order.

Usage

```
## S3 method for class 'sk'
dim(x)
```

Arguments

x	a sk object

Value

integer vector

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
dim(g)
```

---

is.na.sk *Indices of grid points with missing data (NAs)*

Description

Returns a logical vector indicating which grid points have NA values assigned

Usage

```
## S3 method for class 'sk'
is.na(x)
```

Arguments

x	a sk object

Value

a logical vector the same length as x

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
g[c(1,3)] = NA
is.na(g)
```

---

length.sk *The number of grid-points*

Description

Returns the total number of points in the grid, which is the product of the number of y and x grid lines (gdim).

Usage

```
## S3 method for class 'sk'
length(x)
```

Arguments

x	a sk object

Value

integer

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
length(g)
```

Math.sk *Math group generics*

Description

All except the cumsum family are supported

Usage

```
## S3 method for class 'sk'
Math(x, ...)
```

Arguments

x	a sk object
...	further arguments to Math

Value

the "sk" object with data values transformed accordingly

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
summary(g)
summary(abs(g))
summary(exp(g))
```

mean.sk *Calculate the mean value in a grid*

Description

This calculates the mean over all layers (if any)

Usage

```
## S3 method for class 'sk'
mean(x, ...)
```

Arguments

x	a sk object
...	further arguments to default mean method

Value

numeric

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
mean(g) == mean(g[])
```

Ops.sk *Operations group generics*

Description

Applies the operation point-wise to grid data values.

Usage

```
## S3 method for class 'sk'
Ops(e1, e2)
```

Arguments

e1	a "sk" object, vector or matrix
e2	a "sk" object, vector or matrix

Details

The function extracts the grid data `x[]` from all sk class arguments `x`, prior to calling the default method. Before returning, the result is copied back to the grid object of the second argument (or the first, if the second is not of class "sk").

Note that the compatibility of the two arguments is not checked beyond matching dimension (with vectors recycled as needed). This means for example you can do operations on two grids representing different areas, so long as they have the same `gdim`.

Value

a "sk" object (a copy of `e1` or `e2`) with modified data values

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))

# verify a trig identity using Ops and Math
summary( cos(g)^2 + sin(g)^2 )

# create a logical grid indicating points satisfying a condition
plot(g < 0)
all( !(g > 0) == (g[] < 0) )

# test negation
all( (-g)[] == -(g[]) )
```

---

plot.sk *Heatmap plots*

Description

A wrapper for sk_plot

Usage

```
## S3 method for class 'sk'
plot(x, ...)
```

Arguments

`x`	a sk object
`...`	other arguments passed to sk_plot

Value

nothing

See Also

sk_plot

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
plot(g)
```

`print.sk` *Auto-printing*

Description

Prints dimensions and indicates if the grid has values assigned

Usage

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

Arguments

x	a sk object
...	ignored

Details

This prints "(not validated)" if the sk object has no `is_na` entry, to remind users to run `sk_validate`.

Value

nothing

Examples

```
sk_make(list(gdim=10, gres=0.5))
sk_validate(sk_make(list(gdim=10, gres=0.5)))
```

sk *Make a snapKrig grid list object*

Description

Constructs snapKrig ("sk") class list, representing a 2-dimensional regular spatial grid

Usage

```
sk(..., vals = TRUE)
```

Arguments

`...`	raster, matrix, numeric vector, or list of named arguments (see details)
`vals`	logical indicating to include the data vector `gval` in return list

Details

This function accepts 'RasterLayer' and 'RasterStack' inputs from the `raster` package, 'SpatRaster' objects from `terra`, as well as any non-complex matrix, or a set of arguments defining the vectorization of one. It returns a sk class list containing at least the following three elements:

- `gdim`: vector of two positive integers, the number of grid lines (n = their product)
- `gres`: vector of two positive scalars, the resolution (in distance between grid lines)
- `gyx`: list of two numeric vectors (lengths matching gdim), the grid line intercepts

and optionally,

- `crs`: character, the WKT representation of the CRS for the grid (optional)
- `gval`: numeric vector or matrix, the grid data
- `is_obs`: logical vector indicating non-NA entries in the grid data
- `idx_grid`: length-n numeric vector mapping rows of `gval` to grid points

Supply some/all of these elements (including at least one of `gdim` or `gyx`) as named arguments to `...`. The function will fill in missing entries wherever possible.

If `gres` is missing, it is computed from the first two grid lines in `gyx`; If `gyx` is missing, it is assigned the sequence `1:n` (scaled by `gres`, if available) for each `n` in `gdim`; and if `gdim` is missing, it is set to the number of grid lines specified in `gyx`. `gyx` should be sorted (ascending order), regularly spaced (with spacing `gres`), and have lengths matching `gdim`.

Scalar inputs to `gdim`, `gres` are duplicated for both dimensions. For example the call `sk(gdim=c(5,5))` can be simplified to `sk(gdim=5)`, or `sk(5)`.

For convenience, arguments can also be supplied together in a named list passed to `...`. If a single unnamed argument is supplied (and it is not a list) the function expects it to be either a numeric vector (`gdim`), a matrix, or a raster object.

Empty grids - with all data `NA` - can be initialized by setting `vals=FALSE`, in which case `gval` will be absent from the returned list). Otherwise `gval` is the column-vectorized grid data, either as a

numeric vector (single layer case only) or as a matrix with grid data in columns. `gval` is always accompanied by `is_obs`, which supplies an index of `NA` entries (or rows)

A sparse representation is used when `gval` is a matrix, where only the non-`NA` entries (or rows) are stored. `idx_grid` in this case contains `NA`'s were `is_obs` is `FALSE`, and otherwise contains the integer index of the corresponding row in `gval`. In the matrix case it is assumed that each layer (ie column) has the same `NA` structure. `idx_grid` is only computed for the first layer. If a point is missing from one layer, it should be missing from all layers.

Value

a "sk" class list object

See Also

Other sk constructors: `sk_rescale()`, `sk_snap()`, `sk_sub()`

Examples

```
# simple grid construction from dimensions
gdim = c(12, 10)
g = sk(gdim)
summary(g)

# pass result to sk and get the same thing back
identical(g, sk(g))

# supply grid lines as named argument instead and get the same result
all.equal(g, sk(gyx=lapply(gdim, function(x) seq(x)-1L)))

# display coordinates and grid line indices
plot(g)
plot(g, ij=TRUE)

# same dimensions, different resolution, affecting aspect ratio in plot
gres_new = c(3, 4)
plot(sk(gdim=gdim, gres=gres_new))

# single argument (unnamed) can be grid dimensions, with shorthand for square grids
all.equal(sk(gdim=c(2,2)), sk(c(2,2)))
all.equal(sk(2), sk(gdim=c(2,2)))

# example with matrix data
gdim = c(25, 25)
gyx = as.list(expand.grid(lapply(gdim, seq)))
eg_vec = as.numeric( gyx[[2]] %% gyx[[1]] )
eg_mat = matrix(eg_vec, gdim)
g = sk(eg_mat)
plot(g, ij=TRUE, zlab='j mod i')

# y is in descending order
plot(g, xlab='x = j', ylab='y = 26 - i', zlab='j mod i')
```

```
# this is R's default matrix vectorization order
all.equal(eg_vec, as.vector(eg_mat))
all.equal(g, sk(gdim=gdim, gval=as.vector(eg_mat)))

# multi-layer example from matrix
n_pt = prod(gdim)
n_layer = 3
mat_multi = matrix(stats::rnorm(n_pt*n_layer), n_pt, n_layer)
g_multi = sk(gdim=gdim, gval=mat_multi)
summary(g_multi)

# repeat with missing data (note all columns must have consistent NA structure)
mat_multi[sample.int(n_pt, 0.5*n_pt),] = NA
g_multi_miss = sk(gdim=gdim, gval=mat_multi)
summary(g_multi_miss)

# only observed data points are stored, idx_grid maps them to the full grid vector
max(abs( g_multi[['gval']] - g_multi_miss[['gval']][g_multi_miss[['idx_grid']],] ), na.rm=TRUE)

# single bracket indexing magic does the mapping automatically
max(abs( g_multi[] - g_multi_miss[] ), na.rm=TRUE)

# vals=FALSE drops multi-layer information
sk(gdim=gdim, gval=mat_multi, vals=FALSE)

# raster import examples skipped to keep CMD check time < 5s on slower machines

if( requireNamespace('raster') ) {

# open example file as RasterLayer
r_path = system.file('external/rlogo.grd', package='raster')
r = raster::raster(r_path)

# convert to sk (notice only first layer was loaded by raster)
g = sk(r)
summary(g)
plot(g)

# open a RasterStack - gval becomes a matrix with layers in columns
r_multi = raster::stack(r_path)
g_multi = sk(r_multi)
summary(g_multi)
plot(g_multi, layer=1)
plot(g_multi, layer=2)
plot(g_multi, layer=3)

# repeat with terra
if( requireNamespace('terra') ) {

# open example file as SpatRaster (all layers loaded by default)
r_multi = terra::rast(r_path)
g_multi = sk(r_multi)
```

```
summary(g_multi)

# open first layer only
g = sk(r[[1]])
summary(g)

}
}
```

---

sk_cmean *Compute kriging predictor (or variance) for an sk grid*

Description

Evaluates the kriging prediction equations to find the expected value (mean) of the spatial process for g at all grid points, including unobserved ones.

Usage

```
sk_cmean(
  g,
  pars,
  X = NA,
  what = "p",
  out = "s",
  fac_method = "chol",
  fac = NULL,
  quiet = FALSE
)
```

Arguments

g	an sk grid or list accepted by sk (with entries 'gdim', 'gres', 'gval')
pars	list of form returned by sk_pars (with entries 'y', 'x', 'eps', 'psill')
X	sk grid, numeric, vector, matrix, or NA: the mean, or its linear predictors
what	character, what to compute: one of 'p' (predictor), 'v' (variance), or 'm' (more)
out	character, the return object, either 's' (sk grid) or 'v' (vector)
fac_method	character, either 'chol' or 'eigen'
fac	(optional) pre-computed factorization of covariance matrix scaled by partial sill
quiet	logical indicating to suppress console output

Details

This predicts a noiseless version of the random process from which grid g was sampled, conditional on the observed data, and possibly a set of covariates. It is optimal in the sense of minimizing mean squared prediction error under the covariance model specified by pars, and assuming the predictions are a linear combination of the observed data.

The estimation method is determined by X. Set this to 0 and supply a de-trended g to do simple kriging. Set it to NA to estimate a spatially uniform mean (ordinary kriging). Or pass covariates to X, either as multi-layer sk grid or matrix, to do universal kriging. See sk_GLS for details on specifying X in this case.

Set what='v' to return the point-wise kriging variance. This usually takes much longer to evaluate than the prediction, but the computer memory demands are similar. A progress bar will be printed to console in this case unless quiet=TRUE.

Technical notes

All calculations returned by sk_cmean are exact. Our implementation is based on the variance decomposition suggested in section 3.4 (p. 153-155) of Cressie (1993), and uses a loop over eigenvectors (for observed locations) to compute variance iteratively.

In technical terms, sk_cmean estimates the mean of the signal process behind the data. The nugget effect (eps) is therefore added to the diagonals of the covariance matrix for the observed points, but NOT to the corresponding entries of the cross-covariance matrix. This has the effect of smoothing (de-noising) predictions at observed locations, which means sk_cmean is not an exact interpolator (except in the limit eps -> 0). Rather it makes a correction to the observed data to make it consistent with the surrounding signal.

This is a good thing - real spatial datasets are almost always noisy, and we are typically interested in the signal, not some distorted version of it. For more on this see section 3 of Cressie (1993), and in particular the discussion in 3.2.1 on the nugget effect.

The covariance factorization fac can be pre-computed using sk_var with arguments scaled=TRUE (and, if computing variance, fac_method='eigen'). This will speed up subsequent calls where only the observed data values have changed (same covariance structure pars, and same NA structure in the data). The kriging variance does not change in this case and only needs to be computed once.

reference: "Statistics for Spatial Data" by Noel Cressie (1993)

Value

numeric matrix, the predicted values (or their variance)

See Also

sk sk_pars

Other estimators: sk_GLS()

Other variance-related functions: sk_GLS(), sk_LL(), sk_nLL(), sk_sim(), sk_var()

Examples

```
## set up very simple example problem
```

```
# make example grid and covariance parameters
g_all = sk_sim(10)
pars = sk_pars(g_all)
plot(g_all)

# remove most of the data
n = length(g_all)
p_miss = 0.90
is_miss = seq(n) %in% sample.int(n, round(p_miss*n))
is_obs = !is_miss
g_miss = g_all
g_miss[is_miss] = NA
plot(g_miss)


## simple kriging

# estimate the missing data conditional on what's left over
g_simple = sk_cmean(g_miss, pars, X=0)
plot(g_simple)

# variance of the estimator
g_simple_v = sk_cmean(g_miss, pars, X=0, what='v', quiet=TRUE)
plot(g_simple_v)

# get the same results with pre-computed variance
var_pc = sk_var(g_miss, pars, scaled=TRUE, fac_method='eigen')
g_simple_v_compare = sk_cmean(g_miss, pars, X=0, what='v', fac=var_pc, quiet=TRUE)
max(abs(g_simple_v_compare - g_simple_v))

## ordinary kriging

# estimate spatially uniform mean - true value is 0
sk_GLS(g_miss, pars, out='b')

# ordinary kriging automatically adjusts for the trend
g_ok = sk_cmean(g_miss, pars, X=NA)

# additional uncertainty in estimation means variance goes up a bit
g_ok_v = sk_cmean(g_miss, pars, X=NA, what='v', quiet=TRUE)
range(g_ok_v - g_simple_v)


## universal kriging

# introduce some covariates
n_betas = 3
betas = stats::rnorm(n_betas, 0, 10)
g_X = sk_sim(g_all, pars, n_layer=n_betas-1L)
g_lm_all = g_all + as.vector(cbind(1, g_X[]) %*% betas)
g_lm_miss = g_lm_all
g_lm_miss[is_miss] = NA
```

```
# prediction
g_uk = sk_cmean(g_lm_miss, pars, g_X)
g_uk_v = sk_cmean(g_lm_miss, pars, g_X, what='v', quiet=TRUE)


## repeat with special subgrid case (faster!)

# make g_all a subgrid of a larger example
g_super = sk_rescale(g_all, down=2)

# re-generate the covariates for the larger extent
g_X_super = sk_sim(g_super, pars, n_layer=n_betas-1L)
g_lm_super = g_super + as.vector(cbind(1, g_X_super[]) %*% betas)

# prediction
g_super_uk = sk_cmean(g_lm_super, pars, g_X_super)
g_super_uk_v = sk_cmean(g_lm_super, pars, g_X_super, what='v', quiet=TRUE)


## verification

# get observed variance and its inverse
V_full = sk_var(g_all, pars)
is_obs = !is.na(g_miss)
Vo = V_full[is_obs, is_obs]
Vo_inv = solve(Vo)

# get cross covariance
is_diag = as.logical(diag(nrow=length(g_all))[is_obs,])
Vc = V_full[is_obs,]

# nugget adjustment to diagonals (corrects the output at observed locations)
Vc[is_diag] = Vc[is_diag] - pars[['eps']]

# get covariates matrix and append intercept column
X = g_X[]
X_all = cbind(1, X)
X_obs = X_all[is_obs,]

# simple kriging the hard way
z = g_miss[is_obs]
z_trans = Vo_inv %*% z
z_pred_simple = c(t(Vc) %*% z_trans)
z_var_simple = pars[['psill']] - diag( t(Vc) %*% Vo_inv %*% Vc )

# ordinary kriging the hard way
x_trans = ( 1 - colSums( Vo_inv %*% Vc ) )
m_ok = x_trans / sum(Vo_inv)
z_pred_ok = c( (t(Vc) + m_ok) %*% z_trans )
z_var_ok = z_var_simple + diag( x_trans %*% t(m_ok) )

# universal kriging the hard way
z_lm = g_lm_miss[is_obs]
```

```
z_lm_trans = Vo_inv %*% z_lm
x_lm_trans = X_all - t( t(X_obs) %*% Vo_inv %*% Vc )
m_uk = x_lm_trans %*% solve(t(X_obs) %*% Vo_inv %*% X_obs)
z_pred_uk = c( (t(Vc) + t(X_obs %*% t(m_uk)) ) %*% z_lm_trans )
z_var_uk = z_var_simple + diag( x_lm_trans %*% t(m_uk) )

# check that these results agree with sk_cmean
max(abs(z_pred_simple - g_simple), na.rm=TRUE)
max(abs(z_var_simple - g_simple_v))
max(abs(z_pred_ok - g_ok), na.rm=TRUE)
max(abs(z_var_ok - g_ok_v))
max(abs(z_pred_uk - g_uk), na.rm=TRUE)
max(abs(z_var_uk - g_uk_v))


## repeat verification for sub-grid case

# rebuild matrices
V_full = sk_var(g_super_uk, pars)
is_obs = !is.na(g_super)
Vo = V_full[is_obs, is_obs]
Vo_inv = solve(Vo)
is_diag = as.logical(diag(nrow=length(g_super))[is_obs,])
Vc = V_full[is_obs,]
Vc[is_diag] = Vc[is_diag] - pars[['eps']]
X = g_X_super[]
X_all = cbind(1, X)
X_obs = X_all[is_obs,]
z = g_miss[is_obs]

# universal kriging the hard way
z_var_simple = pars[['psill']] - diag( t(Vc) %*% Vo_inv %*% Vc )
z_lm = g_lm_super[is_obs]
z_lm_trans = Vo_inv %*% z_lm
x_lm_trans = X_all - t( t(X_obs) %*% Vo_inv %*% Vc )
m_uk = x_lm_trans %*% solve(t(X_obs) %*% Vo_inv %*% X_obs)
z_pred_uk = c( (t(Vc) + t(X_obs %*% t(m_uk)) ) %*% z_lm_trans )
z_var_uk = z_var_simple + diag( x_lm_trans %*% t(m_uk) )

# verification
max(abs(z_pred_uk - g_super_uk), na.rm=TRUE)
max(abs(z_var_uk - g_super_uk_v))
```

---

sk_coords *Return coordinates of a grid of points in column-vectorized order*

Description

Expands a set of y and x grid line numbers in the column-vectorized order returned by sk. This is similar to base::expand.grid but with the first dimension (y) descending instead of ascending.

Usage

```
sk_coords(g, out = "matrix", corner = FALSE, na_omit = FALSE, quiet = FALSE)
```

Arguments

g	any object accepted by sk
out	character indicating return value type, either 'list', 'matrix', or 'sf'
corner	logical, indicates to return only the four corner points
na_omit	logical, indicates to return only locations of observed points
quiet	logical, suppresses console output

Details

out='sf' returns an sf simple features object containing points in the same order, with data (if any) copied from g[['gval']] into column 'gval'. Note that length(g) points are created, which can be slow for large grids.

If na_omit is TRUE the function omits points with NA data (in gval) and only returns the coordinates for observations. This argument is ignored when corners=TRUE (which always returns the four corner points) or when the grid contains no observations (all points returned).

Value

a matrix, list, or sf point collection in column vectorized order

See Also

sk sk_snap base::expand.grid

Other exporting functions: sk_export()

Examples

```
gdim = c(5,3)
g_complete = sk(gdim=gdim, gres=c(0.5, 0.7), gval=seq(prod(gdim)))
sk_coords(g_complete)
sk_coords(g_complete, out='list')

# missing data example
idx_obs =  sort(sample.int(length(g_complete), 5))
g = sk(gdim=gdim, gres=c(0.5, 0.7))
g[idx_obs] = g_complete[idx_obs]
all.equal(sk_coords(g, na_omit=TRUE), sk_coords(g_complete)[idx_obs,])

# corner points
sk_coords(g, corner=TRUE)
sk_coords(g, corner=TRUE, out='list')

# repeat with multi-layer example
g_multi = sk(utils::modifyList(g, list(gval = cbind(g[], 2*g[]))))
all.equal(sk_coords(g_multi, na_omit=TRUE), sk_coords(g_complete)[idx_obs,])
```

```
sk_coords(g_multi, corner=TRUE)

# sf output type
if( requireNamespace('sf') ) {

# gather all points but don't copy data
sf_coords_all = sk_coords(sk(g, vals=FALSE), out='sf')

# extract non-NA data
sf_coords = sk_coords(g, out='sf', na_omit=TRUE)

# plot everything together
plot(g, reset=FALSE)
plot(sf_coords_all, add=TRUE)
plot(sf_coords, pch=16, cex=2, add=TRUE)

}
```

---

sk_export *Convert "sk" grid to SpatRaster*

Description

Convert "sk" grid to SpatRaster

Usage

```
sk_export(g, template = "terra")
```

Arguments

g	any object accepted or returned by sk
template	character or RasterLayer/SpatRaster to set output type
	Converts a vector or matrix to a SpatRaster or RasterLayer. Multi-layer outputs are supported for terra but not raster.

Value

a RasterLayer or SpatRaster containing the data from g (or a sub-grid)

See Also

sk

Other exporting functions: sk_coords()

Examples

```
if( requireNamespace('raster') ) {

# open example file as RasterLayer
r_path = system.file('external/rlogo.grd', package='raster')
r = raster::raster(r_path, band=1)
g = sk(r)

# convert back to RasterLayer and compare
r_from_g = sk_export(g, 'raster')
print(r)
print(r_from_g)

# NOTE: layer name, band number, and various other metadata are lost
all.equal(r_from_g, r)

}

# same with terra
if( requireNamespace('terra') ) {

# convert all layers
r = terra::rast(r_path)
g = sk(r)
r_from_g = sk_export(g)

# NOTE: various metadata are lost
all.equal(r_from_g, r)

}
```

---

sk_fit *Fit a covariance model to an sk grid by maximum likelihood*

Description

This uses `stats::optim` to minimize the log-likelihood function for a grid dataset g over the space of unknown parameters for the covariance function specified in `pars`. If only one parameter is unknown, the function instead uses `stats::optimize`.

Usage

```
sk_fit(
  g,
  pars = NULL,
  X = NA,
  iso = TRUE,
  n_max = 1000,
```

```
  quiet = FALSE,
  lower = NULL,
  initial = NULL,
  upper = NULL,
  log_scale = TRUE,
  method = "L-BFGS-B",
  control = list()
)
```

Arguments

g	an sk grid (or list with entries 'gdim', 'gres', 'gval')
pars	covariance parameter list, with NAs indicating parameters to fit
X	numeric (or NA), matrix, or sk grid of linear predictors, passed to sk_LL
iso	logical, indicating to constrain the y and x kernel parameters to be the same
n_max	integer, the maximum number of observations allowed
quiet	logical, indicating to suppress console output
lower	numeric vector, lower bounds for parameters
initial	numeric vector, initial values for parameters
upper	numeric vector, upper bounds for parameters
log_scale	logical, indicating to log-transform parameters for optimization
method	character, passed to stats::optim (default is 'L-BFGS-B')
control	list, passed to stats::optim

Details

NA entries in pars are treated as unknown parameters and fitted by the function, whereas non-NA entries are treated as fixed parameters (and not fitted). If none of the parameters in pars are NA, the function copies everything as initial values, then treats all parameters as unknown. pars can also be a character vector defining a pair of correlograms (see ?sk_pars) in which case all covariance parameters are treated as unknown.

Bounds and initial values are set automatically using sk_bds, unless they are otherwise specified in arguments lower, initial, upper. These should be vectors containing only the unknown parameters, *ie.* they must exclude fixed parameters. Call sk_update_pars(pars, iso=iso) to get a template with the expected names and order.

All parameters in the covariance models supported by snapKrig are strictly positive. Optimization is (by default) done on the parameter log-scale, and users can select a non-constrained method if they wish (?stats::optim). As the default method 'L-BFGS-B' is the only one that accepts bounds (lower, initial, upper are otherwise ignored) method is ignored when log_scale=FALSE.

Note that the 'gxp' and 'mat' correlograms behave strangely with very small or very large shape parameters, so for them we recommended 'L-BFGS-B' only.

When there is only one unknown parameter, the function uses stats::optimize instead of stats::optim. In this case all entries of control with the exception of 'tol' are ignored, as are bounds and initial values, and arguments to method.

As a sanity check n_max sets a maximum for the number of observed grid points. This is to avoid accidental calls with very large datasets that would cause R to hang or crash. Set n_max=Inf (with caution) to bypass this check. Similarly the maximum number of iterations is set to 1e3 but this can be changed by manually setting 'maxit' in control.

Value

A parameter list in the form returned by sk_pars containing both fixed and fitted parameters. The data-frame of bounds and initial values is also included in the attribute 'bds'

See Also

sk sk_LL sk_nLL stats::optim stats::optimize

Other parameter managers: sk_bds(), sk_kp(), sk_pars_make(), sk_pars_update(), sk_pars(), sk_to_string()

Examples

```
# define a grid
gdim = c(50, 51)
g_empty = sk(gdim)
pars_src = sk_pars(g_empty)
pars_src = utils::modifyList(pars_src, list(eps=runif(1, 0, 1e1), psill=runif(1, 0, 1e2)))
pars_src[['y']][['kp']] = pars_src[['x']][['kp']] = runif(1, 1, 50)

# generate example data
g_obs = sk_sim(g_empty, pars_src)
sk_plot(g_obs)

# fit (set maxit low to keep check time short)
fit_result = sk_fit(g_obs, pars='gau', control=list(maxit=25), quiet=TRUE)

# compare estimates with true values
rbind(true=sk_pars_update(pars_src), fitted=sk_pars_update(fit_result))

# extract bounds data frame
attr(fit_result, 'bds')

# non-essential examples skipped to stay below 5s exec time on slower machines

# check a sequence of other psill values
pars_out = fit_result
psill_test = ( 2^(seq(5) - 3) ) * pars_out[['psill']]
LL_test = sapply(psill_test, function(s) sk_LL(utils::modifyList(pars_out, list(psill=s)), g_obs) )
plot(psill_test, LL_test)
lines(psill_test, LL_test)
print(data.frame(psill=psill_test, likelihood=LL_test))

# repeat with most data missing
n = prod(gdim)
n_obs = 25
```

```
g_obs = sk_sim(g_empty, pars_src)
idx_miss = sample.int(length(g_empty), length(g_empty) - n_obs)
g_miss = g_obs
g_miss[idx_miss] = NA
sk_plot(g_miss)

# fit (set maxit low to keep check time short) and compare
fit_result = sk_fit(g_miss, pars='gau', control=list(maxit=25), quiet=TRUE)
rbind(true=sk_pars_update(pars_src), fitted=sk_pars_update(fit_result))
```

---

sk_GLS *Generalized least squares (GLS) with Kronecker covariances for sk grids*

Description

Computes coefficients b of the linear model E(Z) = Xb using the GLS equation for sk grid g and covariance model pars. By default the function returns the linear predictor as an sk object

Usage

```
sk_GLS(g, pars, X = NA, out = "s", fac_method = "eigen", fac = NULL)
```

Arguments

g	a sk grid object (or list with entries 'gdim', 'gres', 'gval')
pars	list of form returned by sk_pars (with entries 'y', 'x', 'eps', 'psill')
X	sk grid, matrix or NA, the linear predictors (in columns) excluding intercept
out	character, either 'b' (coefficients), 'z' or 's' (Xb), or 'a' (all)
fac_method	character, factorization method: 'eigen' (default) or 'chol' (see sk_var)
fac	matrix or list, (optional) pre-computed covariance matrix factorization

Details

This is the maximum likelihood estimator for the linear trend Xb if we assume the covariance parameters (in pars) are specified correctly.

The GLS solution is: b = (X^T V^-1 X)^-1 X^T V^-1 z,

where V is the covariance matrix for data vector z (which is g[!is.na(g)]), and X is a matrix of covariates. V is generated from the covariance model pars with grid layout g.

Operations with V^-1 are computed using the factorization fac (see sk_var), or else as specified in fac_method.

Argument X can be an sk grid (matching g) with covariates in layers; or it can be a matrix of covariates. DO NOT include an intercept layer (all 1's) in argument X or you will get collinearity

errors. Matrix X should have independent columns, and its rows should match the order of g[] or g[!is.na(g)].

Use X=NA to specify an intercept-only model; ie to fit a spatially constant mean. This replaces X in the GLS equation by a vector of 1's.

By default out='s' returns the linear predictor in an sk grid object. Change this to 'z' to return it as a vector, or 'b' to get the GLS coefficients only. Set it to 'a' to get the second two return types (in a list) along with matrix X and its factorization.

The length of the vector output for out='z' will match the number of rows in X. This means that if NA grid points are excluded from X, they will not appear in the output (and vice versa). In the X=NA case, the length is equal to the number of non-NA points in g. Note that if a point is observed in g, the function will expect its covariates to be included X (ie X should have no NAs corresponding to non-NA points in g).

If g[] is a matrix (a multi-layer grid), the covariates in X are recycled for each layer. Layers are assumed mutually independent and the GLS equation is evaluated using the corresponding block-diagonal V. This is equivalent to (but faster than) calling sk_GLS separately on each layer with the same X and averaging the resulting b estimates.

Value

linear predictor Xb as an sk grid, or numeric vector, or coefficients (see details)

See Also

sk

Other estimators: sk_cmean()

Other variance-related functions: sk_LL(), sk_cmean(), sk_nLL(), sk_sim(), sk_var()

Examples

```
# set up example grid and covariance parameters
gdim = c(45, 31)
g_empty = sk(gdim)
n = length(g_empty)
pars = utils::modifyList(sk_pars(g_empty, 'gau'), list(psill=2))

# generate spatial noise
g_noise = sk_sim(g_empty, pars)
plot(g_noise)

# generate more spatial noise to use as covariates
n_betas = 3
betas = stats::rnorm(n_betas, 0, 10)
g_X = sk_sim(g_empty, pars, n_layer=n_betas-1L)
X = g_X[]
X_all = cbind(1, X)
g_lm = g_empty
g_lm[] = as.vector(X_all %*% betas)
plot(g_lm)
```

```
# combine with noise to make "observed" data
g_obs = g_lm + g_noise
plot(g_obs)

# By default (out='s') the function returns the linear predictor
g_lm_est = sk_GLS(g_obs, pars, g_X, out='s')
g_lm_est
plot(g_lm_est)

# equivalent, but slightly faster to get vector output
max(abs( sk_GLS(g_obs, pars, g_X, out='z') - g_lm_est[] ))

# repeat with matrix X
max(abs( sk_GLS(g_obs, pars, g_X[], out='z') - g_lm_est[] ))

# return the GLS coefficients
betas_est = sk_GLS(g_obs, pars, g_X, out='b')
print(betas_est)
print(betas)

# compute trend manually as product of betas with X and intercept
lm_est = X_all %*% betas_est
max( abs(lm_est - g_lm_est[] ) )

# de-trend observations by subtracting linear predictor
plot(g_obs - g_lm_est)

# repeat with pre-computed eigen factorization (same result but faster)
fac_eigen = sk_var(g_obs, pars, fac_method='eigen', sep=TRUE)
betas_est_compare = sk_GLS(g_obs, pars, g_X, fac=fac_eigen, out='b')
max( abs( betas_est_compare - betas_est ) )

# missing data example
n_obs = 10
g_miss = g_obs
idx_miss = sort(sample.int(n, n-n_obs))
g_miss[idx_miss] = NA
is_obs = !is.na(g_miss)
plot(g_miss)

# coefficient estimates are still unbiased but less precise
betas_est = sk_GLS(g_miss, pars, g_X, out='b')
print(betas_est)
print(betas)

# set X to NA to estimate the spatially constant trend
b0 = sk_GLS(g_miss, pars, X=NA, out='b')

# matrix X does not need to include unobserved points, but output is filled to match X
X_obs = X[is_obs,]
sk_GLS(g_miss, pars, X=X_obs)
sk_GLS(g_miss, pars, X=X)
```

```
# verify GLS results manually
X_all_obs = cbind(1, X_obs)
V = sk_var(g_miss, pars)
z = g_miss[!is.na(g_miss)]
X_trans = t(X_all_obs) %*% solve(V)
betas_compare = solve( X_trans %*% X_all_obs ) %*% X_trans %*% z
betas_compare - betas_est

# multi-layer examples skipped to stay below 5s exec time on slower machines


# generate some extra noise for 10-layer example
g_noise_multi = sk_sim(g_empty, pars, n_layer=10)
g_multi = g_lm + g_noise_multi
betas_complete = sk_GLS(g_multi, pars, g_X, out='b')
print(betas_complete)
print(betas)

# multi-layer input shares covariates matrix X, and output is to a single layer
summary(sk_GLS(g_multi, pars, g_X))
summary(sk_GLS(g_multi, pars, X))
# note that X cannot be missing data where `g` is observed

# repeat with missing data
g_multi[!is_obs,] = NA
g_X_obs = g_X
g_X_obs[!is_obs,] = NA
betas_sparse = sk_GLS(g_multi, pars, X, out='b')
print(betas_sparse)
print(betas)
summary(sk_GLS(g_multi, pars, g_X))
summary(sk_GLS(g_multi, pars, X))
summary(sk_GLS(g_multi, pars, g_X_obs))
summary(sk_GLS(g_multi, pars, X_obs))
```

---

sk_LL *Likelihood of covariance model* pars *given the data in sk grid* g

---

Description

This computes the log-likelihood for the Gaussian process covariance model pars, given 2-dimensional grid data g, and, optionally, linear trend data in X.

Usage

```
sk_LL(pars, g, X = 0, fac_method = "chol", fac = NULL, quiet = TRUE, out = "l")
```

Arguments

pars	list of form returned by sk_pars (with entries 'y', 'x', 'eps', 'psill')
g	an sk grid (or list with entries 'gdim', 'gres', 'gval' and/or 'idx_grid')
X	numeric, vector, matrix, or NA, a fixed mean value, or matrix of linear predictors
fac_method	character, the factorization to use: 'chol' (default) or 'eigen'
fac	matrix or list, (optional) pre-computed covariance factorization
quiet	logical indicating to suppress console output
out	character, either 'l' (likelihood), 'a' (AIC), 'b' (BIC), or 'more' (see details)

Details

The function evaluates:

`-log( 2 * pi ) - ( 1/2 ) * ( log_det + quad_form )`,

where log_det is the logarithm of the determinant of covariance matrix V, and quad_form is z^T V^-1 z, for the observed response vector z, which is constructed by subtracting the trend specified in X (if any) from the non-NA values in g.

If the trend spatially uniform and known, it can be supplied in argument X as a numeric scalar. The default is zero-mean model X=0, which assumes users has subtracted the trend from g beforehand.

If the trend is unknown, the function will automatically use GLS to estimate it. This is profile likelihood on the covariance function parameters (not REML). To estimate a spatially constant mean, set X=NA. To estimate a spatially variable mean, supply linear predictors as columns of a matrix argument to X (see sk_GLS). Users can also pass a multi-layer bk grid X with covariates in layers.

fac_method specifies how to factorize V; either by using the Cholesky factor ('chol') or eigen-decomposition ('eigen'). A pre-computed factorization fac can be supplied by first calling sk_var(..., scaled=TRUE) (in which case fac_method is ignored).

When out='a', the function instead returns the AIC value and when out='b' it returns the BIC (see stats::AIC). This adjusts the likelihood for the number of covariance and trend parameters (and in the case of BIC, the sample size), producing an index that can be used for model comparison (lower is better).

When out='more', the function returns a list containing the log-likelihood and both information criteria, along with several diagnostics: the number of observations, the number of parameters, the log-determinant log_det, and the quadratic form quad_form.

Value

numeric, the likelihood of pars given g_obs and X, or list (if more=TRUE)

See Also

sk sk_GLS sk_var stats::AIC

Other likelihood functions: sk_nLL()

Other variance-related functions: sk_GLS(), sk_cmean(), sk_nLL(), sk_sim(), sk_var()

Examples

```
# set up example grid, covariance parameters
gdim = c(25, 12)
n = prod(gdim)
g_empty = g_lm = sk(gdim)
pars = utils::modifyList(sk_pars(g_empty, 'gau'), list(psill=0.7, eps=5e-2))

# generate some coefficients
n_betas = 3
betas = stats::rnorm(n_betas)

# generate covariates and complete data in grid and vector form
g_X = sk_sim(g_empty, pars, n_layer=n_betas-1L)
X = g_X[]
X_all = cbind(1, X)
g_lm = g_empty
g_lm[] = c(X_all %*% betas)

# add some noise
g_all = sk_sim(g_empty, pars) + g_lm
z = g_all[]

# two methods for likelihood
LL_chol = sk_LL(pars, g_all, fac_method='chol')
LL_eigen = sk_LL(pars, g_all, fac_method='eigen')

# compare to working directly with matrix inverse
V = sk_var(g_all, pars, fac_method='none', sep=FALSE)
V_inv = chol2inv(chol(V))
quad_form = as.numeric( t(z) %*% crossprod(V_inv, z) )
log_det = as.numeric( determinant(V, logarithm=TRUE) )[1]
LL_direct = (-1/2) * ( n * log( 2 * pi ) + log_det + quad_form )

# relative errors
abs( LL_direct - LL_chol ) / max(LL_direct, LL_chol)
abs( LL_direct - LL_eigen ) / max(LL_direct, LL_eigen)

# get AIC or BIC directly
sk_LL(pars, g_all, out='a')
sk_LL(pars, g_all, out='b')

# repeat with pre-computed variance factorization
fac_eigen = sk_var(g_all, pars, fac_method='eigen', sep=TRUE)
sk_LL(pars, g_all, fac=fac_eigen) - LL_eigen

# repeat with multi-layer example
n_layer = 10
g_noise_multi = sk_sim(g_empty, pars, n_layer)
g_multi = g_lm + g_noise_multi
LL_chol = sk_LL(pars, g_multi, fac_method='chol')
LL_eigen = sk_LL(pars, g_multi, fac_method='eigen')
LL_direct = sum(sapply(seq(n_layer), function(j) {
```

```
 quad_form = as.numeric( t(g_multi[,j]) %*% crossprod(V_inv, g_multi[,j]) )
 (-1/2) * ( n * log( 2 * pi ) + log_det + quad_form )
}))

# relative errors
abs( LL_direct - LL_chol ) / max(LL_direct, LL_chol)
abs( LL_direct - LL_eigen ) / max(LL_direct, LL_eigen)

# repeat with most data missing
is_obs = seq(n) %in% sort(sample.int(n, 50))
n_obs = sum(is_obs)
g_obs = g_empty
z_obs = g_all[is_obs]
g_obs[is_obs] = z_obs

# take subsets of covariates
g_X_obs = g_X
g_X_obs[!is_obs,] = NA
X_obs = X[is_obs,]

LL_chol_obs = sk_LL(pars, g_obs, fac_method='chol')
LL_eigen_obs = sk_LL(pars, g_obs, fac_method='eigen')

# working directly with matrix inverse
V_obs = sk_var(g_obs, pars, fac_method='none')
V_obs_inv = chol2inv(chol(V_obs))
quad_form_obs = as.numeric( t(z_obs) %*% crossprod(V_obs_inv, z_obs) )
log_det_obs = as.numeric( determinant(V_obs, logarithm=TRUE) )[1]
LL_direct_obs = (-1/2) * ( n_obs * log( 2 * pi ) + log_det_obs + quad_form_obs )
abs( LL_direct_obs - LL_chol_obs ) / max(LL_direct_obs, LL_chol_obs)
abs( LL_direct_obs - LL_eigen_obs ) / max(LL_direct_obs, LL_eigen_obs)

# again using a pre-computed variance factorization
fac_chol_obs = sk_var(g_obs, pars, fac_method='chol', scaled=TRUE)
fac_eigen_obs = sk_var(g_obs, pars, fac_method='eigen', scaled=TRUE)
sk_LL(pars, g_obs, fac=fac_chol_obs) - LL_chol_obs
sk_LL(pars, g_obs, fac=fac_eigen_obs) - LL_eigen_obs

# detrend the data by hand, with and without covariates then compute likelihood
g_obs_dtr = g_obs - sk_GLS(g_obs, pars)
g_obs_X_dtr = g_obs - sk_GLS(g_obs, pars, g_X)
LL_dtr = sk_LL(pars, g_obs_dtr, X=0)
LL_X_dtr = sk_LL(pars, g_obs_X_dtr, X=0)

# or pass a covariates grid (or matrix) to de-trend automatically
LL_dtr - sk_LL(pars, g_obs, X=NA)
LL_X_dtr - sk_LL(pars, g_obs, X=g_X)

# note that this introduce new unknown parameter(s), so AIC and BIC increase (worsen)
sk_LL(pars, g_obs, X=NA, out='a') > sk_LL(pars, g_obs_dtr, X=0, out='a')
sk_LL(pars, g_obs, X=g_X, out='a') > sk_LL(pars, g_obs_X_dtr, X=0, out='a')

# X can be the observed subset, or the full grid (as sk grid or as matrix)
```

```
sk_LL(pars, g_obs, X=X)
sk_LL(pars, g_obs, X=X_obs)
sk_LL(pars, g_obs, X=g_X)
sk_LL(pars, g_obs, X=g_X_obs)

# equivalent sparse input specification
g_sparse = g_all
g_sparse[] = matrix(g_obs[], ncol=1)
g_sparse = sk(gval=matrix(g_obs[], ncol=1), gdim=gdim)
LL_chol_obs - sk_LL(pars, g_sparse)
LL_eigen_obs - sk_LL(pars, g_sparse)
LL_dtr - sk_LL(pars, g_sparse, X=NA)
LL_X_dtr - sk_LL(pars, g_sparse, X=g_X)

## repeat with complete data

# the easy way to get likelihood
LL_X_chol = sk_LL(pars, g_all, g_X)
LL_X_eigen = sk_LL(pars, g_all, g_X, fac_method='eigen')

# the hard way
V = sk_var(g_all, pars, sep=FALSE)
V_inv = chol2inv(chol(V))
X_tilde_inv = chol2inv(chol( crossprod(crossprod(V_inv, X_all), X_all) ))
betas_gls = X_tilde_inv %*% crossprod(X_all, (V_inv %*% z))
z_gls = z - (X_all %*% betas_gls)
z_gls_trans = crossprod(V_inv, z_gls)
quad_form = as.numeric( t(z_gls) %*% z_gls_trans )
log_det = as.numeric( determinant(V, logarithm=TRUE) )[1]
LL_direct = (-1/2) * ( n * log( 2 * pi ) + log_det + quad_form )
abs( LL_direct - LL_X_chol ) / max(LL_direct, LL_X_chol)
abs( LL_direct - LL_X_eigen ) / max(LL_direct, LL_X_eigen)

# return detailed list of components with out='more'
LL_result = sk_LL(pars, g_all, X=X, out='more')
LL_result[['LL']] - LL_X_chol
LL_result[['quad_form']] - quad_form
LL_result[['log_det']] - log_det
LL_result[['n_obs']] - n
```

---

sk_nLL *Negative log-likelihood for parameter vector* p

Description

Returns the negative log-likelihood of covariance model `pars_fix`, given the observations in data grid `g_obs`. Parameter values are copied from the first argument, vector `p`, so that the function can be passed to numerical optimizers (etc).

Usage

```
sk_nLL(p, g_obs, pars_fix, X = 0, iso = FALSE, quiet = TRUE, log_scale = FALSE)
```

Arguments

p	numeric vector of covariance parameters accepted by sk_pars_update
g_obs	sk object or list with entries 'gdim', 'gres', 'gval'
pars_fix	list of form returned by sk_pars (with entries 'y', 'x', 'eps', 'psill')
X	numeric, vector, matrix, or NA, the mean or its linear predictors, passed to sk_LL
iso	logical, indicates to use identical kernels for x and y (pars$x is ignored)
quiet	logical indicating to suppress console output
log_scale	logical, indicates that pars_fix contains log-parameter values

Details

This is a wrapper for sk_LL (times -1) that allows parameters to be passed as a numeric vector instead of a list. Parameters in p are copied to pars_fix and passed to the likelihood computer.

p is the vector of covariance parameters to test. Names in p are ignored; Its length and order should correspond with the pattern of NAs in pars_fix. Users should check that the desired parameter list is being constructed correctly by testing with: sk_pars_update(pars_fix, p, iso=iso, na_omit=TRUE).

Value

numeric, the negative log-likelihood of p given data g_obs

See Also

sk sk_GLS sk_var sk_pars_update

Other likelihood functions: sk_LL()

Other variance-related functions: sk_GLS(), sk_LL(), sk_cmean(), sk_sim(), sk_var()

Examples

```
# set up example grid and data
g = sk(gdim=10, gval=stats::rnorm(10^2))

# get some default parameters and vectorize them
pars = sk_pars(g, 'gau')
p = sk_pars_update(pars)
sk_nLL(p, g, pars)

# change a parameter in the numeric vector and re-evaluate
p_compare = p
p_compare[1] = 2 * p_compare[1]
sk_nLL(p_compare, g, pars)
```

```
# repeat by calling sk_LL directly with modified parameters list
pars_compare = pars
pars_compare[['eps']] = 2 * pars_compare[['eps']]
-sk_LL(pars_compare, g)

# set up a subset of parameters to replace - eg when fitting those parameters
pars_fix = pars
pars_fix[['eps']] = NA
pars_fix[['y']][['kp']] = NA

# names in p_fit are for illustration only (only the order matters)
p_fit = c(eps=1, y.rho=1)

# replace NA parameter values in pars_fix to get completed parameters list
sk_pars_update(pars_fix, p_fit, na_omit=TRUE)

# make the replacement and evaluate likelihood in one call
sk_nLL(p_fit, g, pars_fix)

# equivalently:
pars_fit = pars
pars_fit[['eps']] = p_fit[1]
pars_fit[['y']][['kp']] = p_fit[2]
-sk_LL(pars_fit, g)
```

---

sk_pars *Initialize Kronecker covariance function parameters for a sk grid*

Description

Returns a parameter list defining the Kronecker covariance model for g, with default initial values assigned to parameters based on the grid dimensions and sample variance of observed data.

Usage

```
sk_pars(g, pars = "gau", fill = "initial")
```

Arguments

g	list, a sk grid list (or any other object accepted by sk)
pars	character or list defining kernels accepted by sk_pars_make
fill	character, either 'initial', 'lower' or 'upper'

Details

Swap `fill='initial'` with 'lower' and 'upper' to get default lower and upper bounds.

Value

a list defining the Kronecker covariance parameters

See Also

sk sk_corr

Other parameter managers: sk_bds(), sk_fit(), sk_kp(), sk_pars_make(), sk_pars_update(), sk_to_string()

Examples

```
sk_pars(g=10)
sk_pars(c(10,15))
sk_pars(c(10,15), 'mat')
sk_pars(c(10,15), 'mat', 'upper')
```

---

sk_plot *Plot grid data*

Description

Plots a matrix or raster as a heatmap with an optional color bar legend. This is a wrapper for graphics::image similar to terra::plot but with tighter margins to increase the image area on screen; and with a different layout for heat-maps of matrices, which are plotted with i and j axes (row and column numbers) replacing y and x.

Usage

```
sk_plot(g, gdim = NULL, ...)
```

Arguments

g	vector, matrix, or any object understood by sk
gdim	numeric vector, (optional) grid dimensions of the data when g is a vector
...	plotting parameters (see details)

Details

This method is called automatically when a "sk" object is passed to plot.

g can be a vector, in which case gdim supplies the y, x dimensions (ie number of rows, number of columns), in that order. g can also can be a matrix, raster or any other object understood by sk (in which case gdim can be omitted).

The data in g can be of numeric, integer, logical, or factor class. The numeric class is plotted with a continuous color bar legend, while the others get a categorical legend.

Category names (ie tick labels) for the legend can be supplied in argument `breaks`, with one character string for each unique non-NA value in `g`, as integer, in ascending order. If the data are continuous, `breaks` can either be the desired number of bins for coloring, or a vector of break points delineating bins (passed to `graphics::image`). Note that a set of `n` bins has `n+1` break points.

`pal` should be one of the palette names returned by `grDevices::hcl.pals`, or else a vector of color names with length one fewer than the number of break points.

If the data are all `NA`, the function omits the heatmap and legend, and draws grid lines instead. `col_grid` can be specified to enable/disable grid lines more generally. These lines can sometimes appear misaligned due to anti-aliasing. If this is a problem, try switching to a different graphics device back-end (eg. in Windows Rstudio, try changing from the default to Cairo with `options(RStudioGD.backend = 'cairo')`).

The function sets the graphical parameters 'oma' (to `c(0,0,0,0)`) and 'mar' (to values between 1.1 and 5.1, depending on whether titles, axes, and legend are plotted), then calls `graphics::image`, which sets other graphical parameters such as 'usr'. By default all graphical parameters are reset to their original values at the end of the function call. `reset=FALSE` prevents this, so that additional elements can be added to the plot later (such as by calling `sf::st_plot(.., add=TRUE)` or `graphics:lines`).

Value

The function returns a vector of suggested plot height and width values in units of inches which minimize the unused margin space. For example to save a trim version of your plot as png, call `sk_plot` first to get the suggested height and width, say `y` and `x`, then pass the result to `png(filename, height=m*y, width=m*x, pointsize=m*12, ...)`, where `m` is any positive scaling factor.

Plotting parameters

The following style parameters are optional:

adj, leg_just numeric in [0,1]: respectively, the horizontal justification of the title and vertical justification of the color bar legend (default 0.5 for both)

asp numeric or NA: the aspect ratio parameter passed to graphics::image (default 1)

axes, leg logical: respectively indicates to draw axes (y and x, or i and j), the color bar legend (default TRUE)

axes_w, leg_w, lab_w, main_w, oma_w numeric: respectively, the number of lines of margin to reserve for axes (default 2), legend (default 5), axis labels (default 2), main title (default 2), and outer white-space (default 0.5)

breaks numeric (vector) or character vector: the color break points (see details)

cex, cex.main, cex.x, cex.y, cex.z numeric: controls the size of text elements in the plot (default 1), those for title, x/y labels and ticks, and legend title and ticks all inherit the value assigned to cex (unless otherwise specified).

col_box, col_grid character: respectively, the colors to use for drawing a box around the image border and for drawing grid cell boundaries (NA to omit)

col_invert, col_rev logical: respectively, inverts (default FALSE), and reverses the color scale (default TRUE

ij logical: enables/disables matrix style plot with j axis annotations on top (default TRUE for vector and matrix input, otherwise FALSE)

layer integer: the layer (column) to plot (default 1)

lwd_axis, lwd_ticks, lwd_grid numeric: respectively, line widths for the axis lines, ticks, and grid lines (default 1)

main, zlab, ylab, xlab character: respectively, a title to put on top in bold, a legend title to put over the color bar, and axis titles for dimensions y and x. Setting to '' omits both the label and its margin space

minimal logical: produces a stripped down plot showing only the grid data. This omits all annotation (unless otherwise specified by `axes` and/or `leg`) and removes all margin space (unless otherwise specified by `leg_w` and/or `mar_w`)

pal character (vector): one of `graphics::hcl.pals` (default 'Spectral') or a vector of colors

pxmax integer: the maximum number of pixels to draw along each dimension (default 2000). If either x or y dimension exceeds this limit, the grid is up-scaled before plotting

reset logical: indicates to restore original graphical parameters after plot is finished (default TRUE)

zlim numeric vector: range in the data to plot (ignored for discrete plots)

x_ontop logical: toggles the placement of the horizontal dimension axis on top vs bottom

See Also

sk

Other plotting functions: `sk_plot_pars()`

Examples

```
# example grid
gdim = c(50, 100)
n = prod(gdim)
g = sk(gdim)

# plot the grid layout as raster then as matrix
plot(g)
plot(g, ij=TRUE)

# example data: cosine of squared distance to top left corner
z = apply(expand.grid(g$gyx), 1, \(z) cos( 2*sum(z^2) ) )
g_example = utils::modifyList(g, list(gval=z))
plot(g_example)

# plot as matrix (changes default palette)
plot(g_example, ij=TRUE)

# alignment
plot(g_example, ij=TRUE, main='Centered title and legend by default')
plot(g_example, ij=TRUE, main='adj: left-right justification of title', adj=0)
plot(g_example, ij=TRUE, main='leg_just: top-bottom justification of color bar', leg_just=0)

# set the palette - see hcl.pals() for valid names
```

```
pal = 'Zissou 1'
plot(g_example, pal=pal, main=pal)
plot(g_example, pal=pal, main=pal, col_invert=TRUE)
plot(g_example, pal=pal, main=pal, col_invert=TRUE, col_rev=TRUE)

# example data: cosine of distance to top left corner
g[] = apply(expand.grid(g$gyx), 1, \(z) cos( sqrt(sum(z^2))/50 ) )
plot(g)

# specify the layer for multi-layer objects (default is first layer)
g_multi = sk(list(gdim=gdim, gval=cbind(z, z^2)))
plot(g_multi)
plot(g_multi, layer=2)

# reduce number of color breaks or specify a factor for discrete value plots
plot(g, breaks=50)
plot(g, breaks=3)
g[] = cut(g[], breaks=3, dig.lab=1)
plot(g)

# pass color bar labels for discrete plots in breaks (in order low to high)
plot(g, breaks=c('a', 'b', 'c'), zlab='group')

# select some "observed" points and make a covariance matrix
idx_obs = match(seq(n), sort(sample.int(prod(gdim), 1e2)))
g[] = idx_obs
plot(g)
v = sk_var(g)

# matrix display mode is automatic when first argument is a matrix or vector
sk_plot(v, zlab=expression(V[ij]))
sk_plot(c(v), dim(v), zlab=expression(V[ij]))

# or pass the matrix to `sk` first to turn it into a sk grid object
g_v = sk(v)
plot(g_v, zlab=expression(V[ij]))

# minimal versions
plot(g_v, minimal=TRUE)
plot(g_v, minimal=TRUE, leg=TRUE)
plot(g_v, minimal=TRUE, col_grid='white', leg=TRUE)

# logical matrix plots are gray-scale by default
plot(g_v > 1e-2, main='logical matrix')

# logical, integer and factor class matrices get a discrete color-bar
interval = 1e-2 # try also 1e-3 to see behaviour with large number of bins
v_discrete = cut(v, seq(0, ceiling(max(v)), by=interval), dig.lab=2)
g_v[] = cut(v, seq(0, ceiling(max(v)), by=interval), dig.lab=2)
plot(g_v)

# labels are preserved for character matrices
z_char = rep(c('foo', 'bar'), n/2)
```

```
z_char[sample.int(n, n/2)] = NA
sk_plot(z_char, gdim)

# multi-pane plot
g_sim = sk_sim(c(100, 50), n_layer=3)
split.screen(c(1,3))
screen(1)
plot(g_sim, main='layer 1', layer=1, minimal=TRUE, col_box='black')
screen(2)
plot(g_sim, main='layer 2', layer=2, minimal=TRUE, col_box='black')
screen(3)
plot(g_sim, main='layer 3', layer=3, minimal=TRUE, col_box='black')
```

---

sk_plot_pars *Plot the covariance structure of a snapKrig model*

Description

Visualization of the footprint of a covariance kernel as a heatmap of approximate size `dim(g)`, where each grid cell is colored according to its covariance with the central grid point.

Usage

```
sk_plot_pars(pars, g = NULL, simple = FALSE, ...)
```

Arguments

`pars`	list of the form returned by `sk_pars` with entries 'y', 'x', ('eps', 'psill')
`g`	any object understood by `sk`
`simple`	logical, if FALSE the function adds some annotation
`...`	additional arguments passed to `sk_plot`

Details

If g is not supplied, the function sets a default with dimensions 100 x 100. A default resolution is computed such that the maximum nugget-free covariance along the outer edge of the plot is 5% of `pars$psill`.

When `simple=FALSE` (the default), covariance parameters are printed in the title and axis labels with values rounded to 3 decimal places. This can be customized by passing arguments 'main', 'ylab', 'xlab' (and any others accepted `sk_plot` apart from `gdim`).

Value

the same as `sk_plot`

See Also

sk sk_pars

Other plotting functions: sk_plot()

Examples

```
gdim = c(100, 100)
g = sk(gdim)
pars = sk_pars(g, 'mat')

# plot with default grid
sk_plot_pars(pars)

# plot with a predefined grid
sk_plot_pars(pars, g)

# zoom in/out by passing a grid object with suitably modified resolution
gres = g[['gres']]
sk_plot_pars(pars, sk(gdim=gdim, gres=0.5*gres))
sk_plot_pars(pars, sk(gdim=gdim, gres=2*gres))

# change plot style settings (all parameters of sk_plot accepted)
sk_plot_pars(pars, simple=TRUE)
sk_plot_pars(pars, minimal=TRUE)
```

---

sk_plot_semi *Plot a semi-variogram*

Description

Plots a sample semi-variogram using the point pair difference data in vg. Binned summary statistics are drawn as circles with size scaled to the sample sizes. A covariance model (pars) is optionally drawn over the sample data as a ribbon plot.

Usage

```
sk_plot_semi(vg, pars = NULL, add = FALSE, fun = "classical", ...)
```

Arguments

vg	data frame of sample absolute differences, with columns 'dabs', 'd' (and 'bin')
pars	list of the form returned by sk_pars with entries 'y', 'x', 'eps', 'psill'
add	logical, indicates to draw on an existing plot rather than create a new one
fun	character or function, the aggregation function (see details)
...	further plotting parameters (see below)

Details

If vg is a data frame, it should contain absolute differences (numeric 'dabs'), inter-point distances (numeric 'd'), and, optionally, an assignment into distance bins (integer 'bin') for a sample of point pairs. If 'bin' is missing, the function calls sk_add_bins to assign them automatically.

Function fun is the statistic to use for estimating the variogram (ie twice the semi-variogram) from the distance-binned absolute differences in vg. If fun is a function, it must accept sub-vectors of the numeric vg$dabs as its only argument, returning a non-negative numeric scalar. fun can also be set to one of the names 'root_median', 'root_mean' (the default), or 'classical', as shorthand for the robust fourth-root-based methods in section 2.4 of Cressie (1993), or the classical mean of squares method of Matheron.

Optional list pars defines a theoretical semi-variogram to draw over the sample data. When add=TRUE, the function overlays it on an existing plot (without changing the legend, plot limits etc). Anisotropic models, which may assume a range of semi-variances for any given distance, are drawn as a ribbon plot.

add=TRUE can only be used in combination with an earlier call to sk_plot_semi where reset=FALSE (which allows the function to change R's graphical parameters)

vg can be a grid object (anything understood by sk) rather than a variogram data frame. When add=FALSE, the function uses it to set the distance limits for an initial empty plot (the model semi-variance is then drawn if pars is supplied).

Value

nothing

Plotting parameters

The following style parameters are optional:

alpha_bin, alpha_model, alpha_bin_b, alpha_model_b numeric in [0,1]: respectively, the transparency of the fill color for circles and ribbons (default 0.3), and their borders (default 0.5)

bty character: plot frame border type, passed to base::plot (default 'n' for no border)

col_bin, col_model character: respectively, the color to use for circles (default 'black') and ribbons (default 'blue')

cex_bin numeric > 0: scaling factor for circles (default 1.5)

d_max numeric > 0: x (distance) limit for plotting in input units

leg logical: adds a sample bin legend

leg_main character: title for the sample bin legend (default 'model')

lwd numeric: line width for the model semi-variance

main character: a title

n_bin, n_test integer: respectively, the number of distance bins for the sample (optional if vg has a 'bin' column, and ignored if vg is a grid object), and the number of distances at which to evaluate the semi-variogram for model pars (default 5e3, ignored if pars not supplied)

reset logical: indicates to reset graphical parameters to their original values when finished (default TRUE)

xlab, ylab character: titles for the y and x axes. The default for y is 'semi-variance (gamma)', and for x 'distance'

Examples

```
# make example grid and reference covariance model
gdim = c(10, 15)
g_empty = sk(gdim)
pars = sk_pars(g_empty, 'mat')

# plot a semivariance model
sk_plot_semi(g_empty)
sk_plot_semi(g_empty, pars)

# change annotations, sharpen ribbon border
sk_plot_semi(g_empty, pars, main='title', xlab='x', ylab='y')
sk_plot_semi(g_empty, pars, alpha_model_b=1, main='example title', xlab='x', ylab='y')

# generate sample data and sample semivariogram
g_obs = sk_sim(g_empty, pars)
vg = sk_sample_vg(g_obs)
sk_plot_semi(vg)

# different aggregation methods produce variety of results
sk_plot_semi(vg, fun='root_median')
sk_plot_semi(vg, fun='root_mean')
sk_plot_semi(vg, fun='classical') # default
sk_plot_semi(vg, fun=function(x) mean(x^2)) # same as classical

# plot again with reference model and adjust distance limits, number of bins
sk_plot_semi(vg, pars)
sk_plot_semi(vg, pars, d_max=10)
sk_plot_semi(vg, pars, d_max=10, n_bin=1e2)

# add dashed line for half sample variance (this tends to underestimate the sill)
sk_plot_semi(vg, pars)
sample_var = var(g_obs[['gval']], na.rm=TRUE)
abline(h=sample_var/2, lty='dashed')

# initial call with reset=FALSE, then use add=TRUE to overlay the same model with a green fill
sk_plot_semi(vg, pars, lwd=2, reset=FALSE)
sk_plot_semi(vg, pars, add=TRUE, col_model='green', alpha_model_b=0)

# overlay several models with varying nugget effect
pars_vary = pars
for(i in seq(3))
{
  pars_vary$eps = 0.8 * pars_vary$eps
  sk_plot_semi(vg, pars_vary, add=TRUE, alpha_model_b=0)
}
dev.off()
```

sk_rescale *Up or down-scale a sk grid by an integer factor*

Description

Changes the resolution of a sk grid by a factor of up or down. For down-scaling, this introduces NAs at unobserved grid points (and does no interpolation).

Usage

```
sk_rescale(g, up = NULL, down = NULL)
```

Arguments

g	a sk grid or any grid object accepted by sk
up	integer > 0, or vector of two, the up-scaling factors
down	integer > 0, or vector of two, the down-scaling factors

Details

Users should specify a sk grid g to re-scale and an integer scaling factor; either up or down (and not both). This effects the scaling of resolution (g[['gres']]) by up or 1/down.

up (or down) should be a vector of two positive integers, the desired re-scaling factors in the y and x dimensions, in that order, or a single value to be used for both.

When up is supplied, a lower resolution grid is returned comprising every upth grid line of g along each dimension. All other grid lines, and any data values lying on them, are ignored. up should be no greater than dim(g) - 1. Note that if up does not evenly divide this number, the bounding box will shrink slightly.

When down is supplied, the function returns a higher resolution grid (say g_fine) with the same bounding box as g. Along each dimension, every downth grid line of g_fine coincides with a grid line of g. Any non-NA values found in g[] are copied to g_fine, and g can be recovered from g_fine with sk_rescale(g_fine, up=down).

Value

a sk grid of the requested resolution

See Also

sk sk_cmean

Other indexing functions: sk_mat2vec(), sk_sub_find(), sk_sub_idx(), sk_vec2mat()

Other sk constructors: sk_snap(), sk_sub(), sk()

Examples

```
# example data
gdim = c(50, 53)
pars = utils::modifyList(sk_pars(gdim), list(eps=1e-2))
g = sk_sim(gdim, pars)
plot(g)

# upscale
plot(sk_rescale(g, up=1)) # does nothing
plot(sk_rescale(g, up=2))

# downscale
sk_plot(sk_rescale(g, down=1)) # does nothing
sk_plot(sk_rescale(g, down=2))

# length-2 vectors to rescale differently in x and y directions
plot(sk_rescale(g, up=c(2,3)))
plot(sk_rescale(g, down=c(2,3)))

# invert a down-scaling
g_compare = sk_rescale(sk_rescale(g, down=c(5,3)), up=c(5,3))
all.equal(g, g_compare)

# multi-layer example with about 50% of points missing
idx_miss = sample.int(length(g), round(0.5*length(g)))
g_multi = sk_sim(gdim, pars, n_layer=3)
g_multi[idx_miss,] = NA

# plot third layer, then down-scaled and up-scaled versions
sk_plot(g_multi, layer=3)
sk_plot(sk_rescale(g=g_multi, down=2), layer=3)
sk_plot(sk_rescale(g=g_multi, up=2), layer=3)
```

`sk_sample_vg` *Sample point pair absolute differences for use in semi-variogram estimation*

Description

Compute the absolute differences for point pairs in g, along with their separation distances. If no sample point index is supplied (in `idx`), the function samples points at random using `sk_sample_pt`.

Usage

```
sk_sample_vg(
  g,
  n_pp = 10000,
```

```
  idx = NULL,
  n_bin = 25,
  n_layer_max = NA,
  quiet = FALSE
)
```

Arguments

g	any grid object accepted or returned by sk
n_pp	integer maximum number of point pairs to sample
idx	optional integer vector indexing the points to sample
n_bin	integer number of distance bins to assign (passed to sk_add_bins)
n_layer_max	integer, maximum number of layers to sample (for multi-layer g)
quiet	logical, suppresses console output

Details

In a set of n points there are n_pp(n) = (n^2-n)/2 possible point pairs. This expression is inverted to determine the maximum number of sample points in g to use in order to satisfy the argument n_pp, the maximum number of point pairs to sample. A random sub-sample of idx is taken as needed. By default n_pp=1e4 which results in n=141.

The mean of the point pair absolute values ('dabs') for a given distance interval is the classical estimator of the variogram. This and two other robust methods are implemented in sk_plot_semi. These statistics are sensitive to the choice of distance bins. They are added automatically by a call to sk_add_bins (with n_bin) but users can also set up bins manually by adjusting the 'bin' column of the output.

For multi-layer g, the function samples observed point locations once and re-uses this selection in all layers. At most n_layer_max layers are sampled in this way (default is the square root of the number of layers, rounded up)

Value

A data frame with a row for each sampled point pair. Fields include 'dabs' and 'd', the absolute difference in point values and the separation distance, along with the vector index, row and column numbers, and component (x, y) distances for each point pair. 'bin' indicates membership in one of n_bin categories.

See Also

sk sk_sample_pt sk_add_bins

Examples

```
# make example grid and reference covariance model
gdim = c(22, 15)
n = prod(gdim)
g_empty = sk(gdim)
```

```
pars = sk_pars(g_empty, 'mat')

# generate sample data and sample semi-variogram
g_obs = sk_sim(g_empty, pars)
vg = sk_sample_vg(g_obs)
str(vg)

# pass to plotter and overlay the model that generated the data
sk_plot_semi(vg, pars)

# repeat with smaller sample sizes
sk_plot_semi(sk_sample_vg(g_obs, 1e2), pars)
sk_plot_semi(sk_sample_vg(g_obs, 1e3), pars)

# use a set of specific points
n_sp = 10
( n_sp^2 - n_sp ) / 2 # the number of point pairs
vg = sk_sample_vg(g_obs, idx=sample.int(n, n_sp))
sk_plot_semi(vg, pars)

# non-essential examples skipped to stay below 5s exec time on slower machines


# repeat with all point pairs sampled (not recommended for big data sets)
vg = sk_sample_vg(g_obs, n_pp=Inf)
sk_plot_semi(vg, pars)
( n^2 - n ) / 2 # the number of point pairs

## example with multiple layers

# generate five layers
g_obs_multi = sk_sim(g_empty, pars, n_layer=5)

# by default, a sub-sample of sqrt(n_layers) is selected
vg = sk_sample_vg(g_obs_multi)
sk_plot_semi(vg, pars)

# change this behaviour with n_layer_max
vg = sk_sample_vg(g_obs_multi, n_layer_max=5)
sk_plot_semi(vg, pars)
```

`sk_sim` *Random draw from multivariate normal distribution for sk grids*

Description

Generates a random draw from the multivariate Gaussian distribution for the covariance model `pars` on grid g, with mean zero.

Usage

```
sk_sim(g, pars = sk_pars(g), n_layer = 1, fac = NULL, sk_out = TRUE)
```

Arguments

g	an sk object or any grid object accepted by sk
pars	list, covariance parameters in form returned by sk_pars
n_layer	positive integer, the number of draws to return
fac	list, optional pre-computed factorization of component correlation matrices
sk_out	logical, if TRUE an sk grid is returned

Details

pars and g define the model's covariance matrix V. This function uses base::rnorm to get a vector of independent standard normal variates, which it multiplies by the square root of the covariance matrix, V, for the desired model (as defined by pars and g). The result has a multivariate normal distribution with mean zero and covariance V.

Multiple independent draws can be computed more efficiently by reusing the factorization of V. This can be pre-computed with sk_var and supplied in fac, or users can set n_layer and the function will do this automatically.

Value

sk grid or its vectorized form (vector for single-layer case, matrix for multi-layer case)

See Also

sk sk_pars base::rnorm

Other variance-related functions: sk_GLS(), sk_LL(), sk_cmean(), sk_nLL(), sk_var()

Examples

```
# example grid and covariance parameters
gdim = c(100, 200)
g = sk(gdim)
pars_gau = sk_pars(g)

# this example has a large nugget effect
g_sim = sk_sim(g, pars_gau)
plot(g_sim)

# repeat with smaller nugget effect for less noisy data
pars_smooth = utils::modifyList(pars_gau, list(eps=1e-2))
g_sim = sk_sim(g, pars_smooth)
plot(g_sim)

# the nugget effect can be very small, but users should avoid eps=0
pars_smoother = utils::modifyList(pars_gau, list(eps=1e-12))
```

```
g_sim = sk_sim(g, pars_smoother)
plot(g_sim)

# multi-layer example
g_sim_multi = sk_sim(g, pars_smoother, n_layer=3)
plot(g_sim_multi, layer=1)
plot(g_sim_multi, layer=2)
plot(g_sim_multi, layer=3)
```

---

sk_snap *Snap a set of points to a "sk" grid*

Description

Maps the input points in `from` to the closest grid points in the lattice of which g is a sub-grid. In cases of duplicate mappings, the function returns the first matches only.

Usage

```
sk_snap(from, g = nrow(from), crop_from = FALSE, crop_g = FALSE, quiet = FALSE)
```

Arguments

`from`	matrix, data frame, or points object from `sp` or `sf`, the source points
g	any object accepted or returned by `sk`, the destination grid
`crop_from`	logical, indicating to omit points not overlying g.
`crop_g`	logical, indicating to trim g to the extent of `from`.
`quiet`	logical, suppresses console output

Details

`from` can be a geometry collection from packages `sf` or `sp`, or a matrix or list of y and x coordinates. When `from` is a matrix, its first two columns should be the y and x coordinates (in that order), and the (optional) third column should be the data. When `from` is a list, the function expects (two or three) vectors of equal length, ordered as above.

When `from` is a geometry collection with a coordinates reference system (CRS) string, points are first transformed to the CRS of g. If one or both of `from` and g are missing a CRS definition, the function assumes the same one is shared in both.

g can be a raster geometry object (such as SpatRaster), in which case the function behaves like `terra::rasterize`, or an sk grid object. It can also be a matrix (supplying dimensions) or a list containing either `gdim` or`gres`, from which an appropriately spaced set of grid lines is derived, centered under the bounding box of the points. If g is not supplied, it is automatically set to equal `nrow(from)`, so that there there is one grid line along each dimension for each input point.

`crop_from` and `crop_g` control the extent of the output grid. If both are `FALSE` (the default) the function returns the smallest regular grid containing both g and the snapped `from` points. If

crop_from=TRUE and crop_g=FALSE the output grid will match g exactly. If crop_from=FALSE and crop_g=TRUE the output grid will include all snapped points, and possibly omit some or all of g. And if both are TRUE, the output grid encloses the intersection of the points with the bounding box of g.

Value

sk object, a grid containing the snapped points. These are assigned the corresponding data value in from, or if from has no data, an integer mapping to the points in from. Un-mapped grid points are set to NA.

See Also

sk sk_coords

Other sk constructors: sk_rescale(), sk_sub(), sk()

Examples

```
# functions to scale arbitrary inverval to (1, 2,... 100) and make color palettes
num_to_cent = function(x) 1L + floor(99*( x-min(x) ) / diff(range(x)))
my_pal = function(x) grDevices::hcl.colors(x, 'Spectral', rev=TRUE)
my_col = function(x) my_pal(1e2)[ num_to_cent(x) ]

# create a grid object
gdim = c(40, 30)
g = sk(gdim=gdim, gres=1.1)

# randomly position points within bounding box of g
n_pts = 10
from = lapply(g$gyx, function(yx) runif(n_pts, min(yx), max(yx)) )

# translate away from g (no overlap is required)
from[['y']] = from[['y']] + 5
from[['x']] = from[['x']] + 15

# add example data values and plot
from[['z']] = stats::rnorm(length(from[['y']]))
plot(g, reset=FALSE)
graphics::points(from[c('x', 'y')], pch=16, col=my_col(from[['z']]))
graphics::points(from[c('x', 'y')])

# snap only the points overlying the input grid
g_snap = sk_snap(from, g, crop_from=TRUE)
plot(g_snap, col_grid='black', reset=FALSE, leg=FALSE)
graphics::points(from[c('x', 'y')], pch=16, col=my_col(from[['z']]))
graphics::points(from[c('x', 'y')])

# snap all points to grid extension (default settings)
g_snap = sk_snap(from, g, crop_from=FALSE, crop_g=FALSE)
plot(g_snap, col_grid='black', reset=FALSE)
graphics::points(from[c('x', 'y')], pch=16, col=my_col(from[['z']]))
```

```
graphics::points(from[c('x', 'y')])

# find smallest subgrid enclosing all snapped grid points
g_snap = sk_snap(from, g, crop_g=TRUE)
plot(g_snap, col_grid='black', reset=FALSE)
graphics::points(from[c('x', 'y')], pch=16, col=my_col(from[['z']]))
graphics::points(from[c('x', 'y')])

# create a new grid of different resolution enclosing all input points
g_snap = sk_snap(from, g=list(gres=c(0.5, 0.5)))
plot(g_snap, reset=FALSE, col_grid='black')
graphics::points(from[c('x', 'y')], pch=16, col=my_col(from[['z']]))
graphics::points(from[c('x', 'y')])

if( requireNamespace('sf') ) {

# a different example, snapping mis-aligned subgrid
g_pts = sk(list(gdim=c(15, 8), gres=1.7), vals=FALSE)
g_pts[['gyx']][['y']] = g_pts[['gyx']][['y']] + 5
g_pts[['gyx']][['x']] = g_pts[['gyx']][['x']] + 5
from = sk_coords(g_pts, out='list')

# convert to sf
eg_sfc = sf::st_geometry(sk_coords(g_pts, out='sf'))
plot(g, reset=FALSE)
plot(eg_sfc, add=TRUE)

# generate example data and plot
eg_sf = sf::st_sf(data.frame(z=stats::rnorm(length(g_pts))), geometry=eg_sfc)
plot(g, reset=FALSE)
plot(eg_sf, pch=16, add=TRUE, pal=my_pal)
plot(eg_sfc, add=TRUE)

# snap points
g_snap = sk_snap(from=eg_sf, g)
plot(g_snap, reset=FALSE, col_grid='black')
plot(eg_sf, pch=16, add=TRUE, pal=my_pal)
plot(eg_sfc, add=TRUE)

# snapping points without data produces the mapping (non-NA values index "from")
g_snap = sk_snap(from=eg_sfc, g)
plot(g_snap, ij=TRUE, reset=FALSE, col_grid='black')
plot(eg_sfc, add=TRUE)

# with crop_g=TRUE)
g_snap = sk_snap(from=eg_sfc, g, crop_g=TRUE)
plot(g_snap, reset=FALSE, col_grid='black')
plot(eg_sfc, add=TRUE)

# test with sp class
eg_sp = as(eg_sf,'Spatial')
g_snap = sk_snap(from=eg_sp, g)
plot(g_snap, reset=FALSE, col_grid='black')
```

```
plot(eg_sf, pch=16, add=TRUE, pal=my_pal)
plot(eg_sfc, add=TRUE)

}
```

---

sk_var *Generate a covariance matrix or its factorization*

Description

Computes the covariance matrix V (or one of its factorizations) for the non-NA points in sk grid g, given the model parameters list pars

Usage

```
sk_var(
  g,
  pars = NULL,
  scaled = FALSE,
  fac_method = "none",
  X = NULL,
  fac = NULL,
  sep = FALSE
)
```

Arguments

g	a sk grid object or a list with entries 'gdim', 'gres', 'gval'
pars	list of form returned by sk_pars (with entries 'y', 'x', 'eps', 'psill')
scaled	logical, whether to scale by 1/pars$psill
fac_method	character, the factorization to return, one of 'none', 'chol', 'eigen'
X	numeric matrix, the X in t(X) %*% V %*% X (default is identity, see details)
fac	matrix or list of matrices, the variance factorization (only used with X)
sep	logical, indicating to return correlation components instead of full covariance matrix

Details

By default the output matrix is V. Alternatively, if X is supplied, the function returns the quadratic form X^T V^-1 X.

When fac_method=='eigen' the function instead returns the eigen-decomposition of the output matrix, and when fac_method=='chol' its lower triangular Cholesky factor is returned. Supplying this factorization in argument fac in a subsequent call with X can speed up calculations. fac is ignored when X is not supplied.

scaled=TRUE returns the matrix scaled by the reciprocal of the partial sill, 1/pars$psill, before factorization. This is the form expected by functions sk_var_mult and sk_LL in argument fac.

Numerical precision issues with poorly conditioned covariance matrices can often be resolved by using 'eigen' factorization method (instead 'chol') and making sure that pars$eps > 0.

If all grid points are observed, then the output V becomes separable. Setting sep=TRUE in this case causes the function to return the x and y component correlation matrices (or their factorizations, as requested in fac_method) separately, in a list. scaled has no effect in this output mode. Note also that sep has no effect when X is supplied.

If the function is passed an empty grid g (all points NA) it returns results for the complete case (no NAs). If it is passed a list that is not a sk grid object, it must include entries 'gdim', 'gres', 'gval' and/or 'idx_grid' (as they are specified in sk), all other entries are ignored in this case.

Value

either matrix V, or X^T V^-1 X, or a factorization ('chol' or 'eigen')

See Also

sk

Other variance-related functions: sk_GLS(), sk_LL(), sk_cmean(), sk_nLL(), sk_sim()

Examples

```
# define example grid with NAs and example predictors matrix
gdim = c(12, 13)
n = prod(gdim)
n_obs = floor(n/3)
idx_obs = sort(sample.int(n, n_obs))
g = g_empty = sk(gdim)
g[idx_obs] = stats::rnorm(n_obs)
plot(g)

# example kernel
psill = 0.3
pars = utils::modifyList(sk_pars(g), list(psill=psill))

# plot the covariance matrix for observed data, its cholesky factor and eigen-decomposition
V_obs = sk_var(g, pars)
V_obs_chol = sk_var(g, pars, fac_method='chol')
V_obs_eigen = sk_var(g, pars, fac_method='eigen')
sk_plot(V_obs)
sk_plot(V_obs_chol)
sk_plot(V_obs_eigen$vectors)

# empty and complete cases are treated the same

# get the full covariance matrix with sep=FALSE (default)
V_full = sk_var(g_empty, pars)

# check that the correct sub-matrix is there
```

```
max(abs( V_obs - V_full[idx_obs, idx_obs] ))

# get 1d correlation matrices with sep=TRUE...
corr_components = sk_var(g_empty, pars, sep=TRUE)
str(corr_components)
sk_plot(corr_components[['x']])

# ... these are related to the full covariance matrix through psill and eps
corr_mat = kronecker(corr_components[['x']], corr_components[['y']])
V_full_compare = pars$psill * corr_mat + diag(pars$eps, n)
max(abs(V_full - V_full_compare))

# ... their factorizations can be returned as (nested) lists
str(sk_var(g_empty, pars, fac_method='chol', sep=TRUE))
str(sk_var(g_empty, pars, fac_method='eigen', sep=TRUE))

# compare to the full covariance matrix factorizations (default sep=FALSE)
str(sk_var(g_empty, pars, fac_method='chol'))
str(sk_var(g_empty, pars, fac_method='eigen'))

# test quadratic form with X
nX = 3
X_all = cbind(1, matrix(stats::rnorm(nX * n), ncol=nX))
cprod_all = crossprod(X_all, chol2inv(chol(V_full))) %*% X_all
abs(max(sk_var(g_empty, pars, X=X_all) - cprod_all ))

# test products with inverse of quadratic form with X
mult_test = stats::rnorm(nX + 1)
cprod_all_inv = chol2inv(chol(cprod_all))
cprod_all_inv_chol = sk_var(g_empty, pars, X=X_all, scaled=TRUE, fac_method='eigen')
sk_var_mult(mult_test, pars, fac=cprod_all_inv_chol) - cprod_all_inv %*% mult_test

# repeat with missing data
X_obs = X_all[idx_obs,]
cprod_obs = crossprod(X_obs, chol2inv(chol(V_obs))) %*% X_obs

abs(max(sk_var(g, pars, X=X_obs) - cprod_obs ))
cprod_obs_inv = chol2inv(chol(cprod_obs))
cprod_obs_inv_chol = sk_var(g, pars, X=X_obs, scaled=TRUE, fac_method='eigen')
sk_var_mult(mult_test, pars, fac=cprod_obs_inv_chol) - cprod_obs_inv %*% mult_test

# `scaled` indicates to divide matrix by psill
print( pars[['eps']]/pars[['psill']] )
diag(sk_var(g, pars, scaled=TRUE)) # diagonal elements equal to 1 + eps/psill
( sk_var(g, pars) - psill * sk_var(g, pars, scaled=TRUE) ) |> abs() |> max()
( sk_var(g, pars, X=X_obs, scaled=TRUE) - ( cprod_obs/psill ) ) |> abs() |> max()

# in Cholesky factor this produces a scaling by square root of psill
max(abs( V_obs_chol - sqrt(psill) * sk_var(g, pars, fac_method='chol', scaled=TRUE) ))

# and in the eigendecomposition, a scaling of the eigenvalues
vals_scaled = sk_var(g, pars, fac_method='eigen', scaled=TRUE)$values
max(abs( sk_var(g, pars, fac_method='eigen')$values - psill*vals_scaled ))
```

summary.sk *Grid summary*

Description

Prints detailed information about a grid

Usage

```
## S3 method for class 'sk'
summary(object, ...)
```

Arguments

object	an sk object
...	ignored

Details

All dimensional information (gdim, gres, gyx) is printed in the order y, x

Value

nothing

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
summary(g)
g[1] = NA
summary(g)
```

[.sk *Extract a sk list element (single-bracket access)*

Description

Copies the specified list element or grid point value

Usage

```
## S3 method for class 'sk'
x[i = NULL, j = NULL, drop = FALSE, ...]
```

Arguments

x	a sk object
i	column-vectorized index
j	index of layer (only for multi-layer x)
drop	ignored
...	ignored

Details

Behavior depends on the class of i. For character vectors this extracts the named list entries of x. For numeric, it accesses the vectorized grid data values. For multi-layer objects, a layer can be specified in j.

the default NULL for i and j is treated as numeric, and is shorthand for all indices. For example if x has a single-layer x[] returns all grid data in a vector. If x is multi-layer x[,1] all grid data from the first layer, and x[] returns all layers, as a matrix.

Value

a list, vector, or matrix (see description)

Examples

```
# define a sk list and extract two of its elements
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
g[c('gdim', 'gres')]

# display all the grid data as a vector or a matrix
g[]
matrix(g[], dim(g))

# extract a particular grid point or a subset
g[1]
g[seq(5)]
```

[<-.sk *Single-bracket assign*

Description

Behavior depends on the class of i. For character vectors, this assigns to the named list entries of x (as usual). For numeric indices, it assigns vectorized grid data values. For multi-layer objects, specify the layer in j and supply a matrix for replacement

Usage

```
## S3 replacement method for class 'sk'
x[i = NULL, j = NULL] <- value
```

Arguments

x	an sk object
i	column-vectorized index
j	index of layer (only for multi-layer x)
value	the replacement values

Value

the "sk" object with the specified subset replaced by value

Examples

```
g = sk_validate(list(gval=stats::rnorm(4^2), gdim=4, gres=0.5))
print(g)
g[1] = NA
print(g)
```

[[.sk<- *sk_methods.R Dean Koch, 2022 S3 methods for sk grid list objects*

Description

Replace a sk list element (double-bracket assign)

Usage

```
## S3 method for class '`sk<-`'
x[[value]]
```

Arguments

x	a sk object
value	the replacement object

Details

Replaces entries in the sk list object. This does no validation. If it did, then `sk_validate` would have an infinite recursion problem (it uses `[[<-`). Users should pass the results to `sk_validate` afterwards unless they know what they're doing.

Value

a "sk" object

Examples

```
# sk list elements are interrelated - for example gres must match spacing in gyx
g = sk_validate(list(gval=stats::rnorm(10^2), gdim=10, gres=0.5))
g[['gres']] = 2 * g[['gres']]
g[['gyx']] = lapply(g[['gyx']], function(x) 2*x)
sk_validate(g)
```

Index