wingen-vignette

```
library(wingen)
library(terra)
library(raster)
library(viridis)
library(sf)
library(SpatialKDE)
```

Background

wingen uses a moving window approach to create maps of genetic diversity. The method and its rationale are described in Bishop et al. (2023).

Example

To demonstrate how wingen works, we will use a subset of the data from the Bishop et al. (2023) simulation example. These simulations were created using Geonomics (Terasaki Hart et al., 2022) to generate a realistic landscape genomic dataset. In this simulation, spatial variation in genetic diversity is produced by varying population size and gene flow across the landscape via heterogeneous carrying capacity and conductance surfaces. These surfaces are based on an example digital elevation model of Tolkien's Middle Earth produced by the Center for Geospatial Analysis at William & Mary (Robert, 2020).

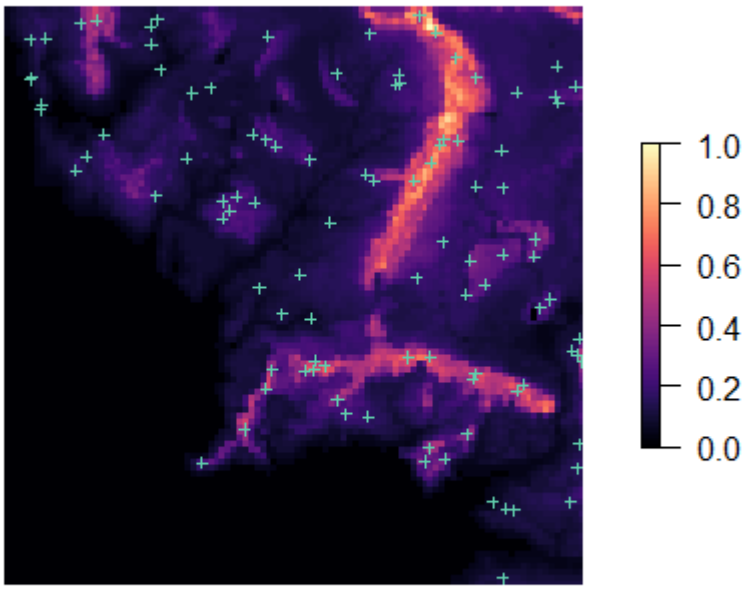
Load Middle Earth example

The small Middle Earth example dataset used here contains four objects which are loaded by `load_middle_earth_ex()`:

1. `lotr_vcf` - a vcfR object containing the genetic data
2. `lotr_coords` - a dataframe object containing sample coordinates
3. `lotr_lyr` - a raster object of the landscape (higher values indicate greater connectivity/carrying capacity)
4. `lotr_range` - a polygon outlining the "range" of the simulated species

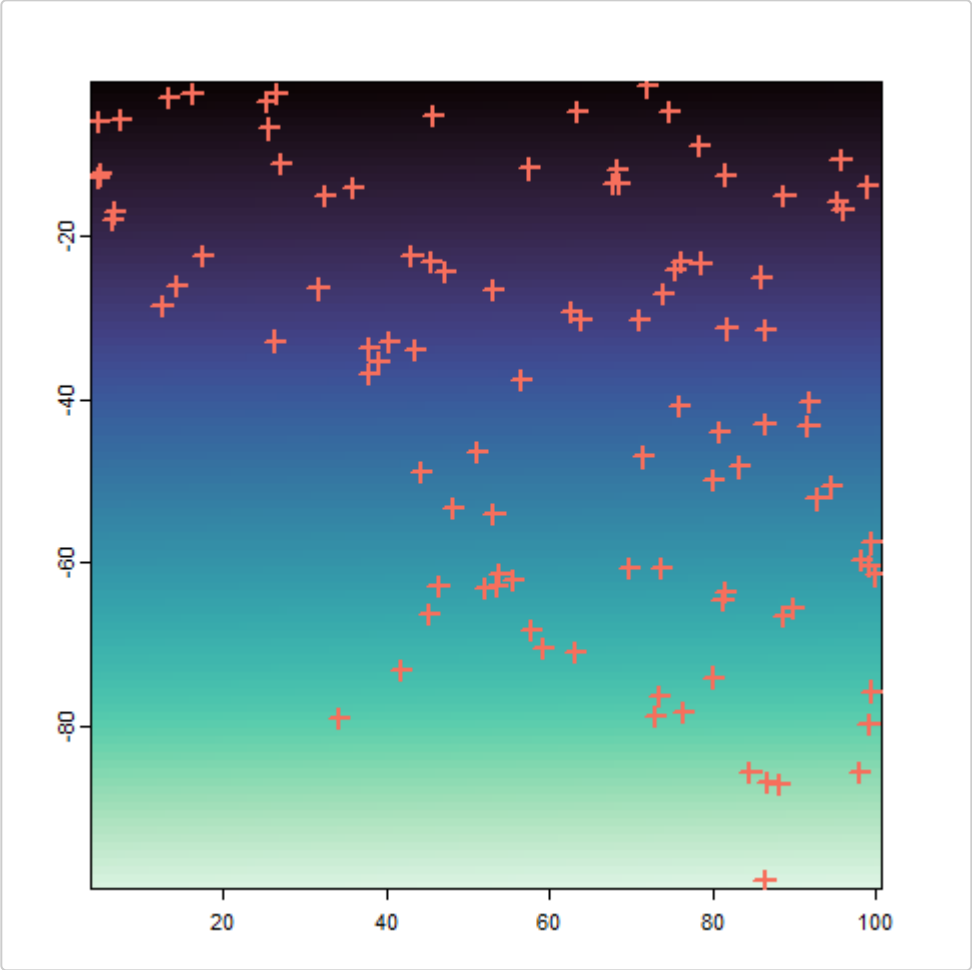
```
load_middle_earth_ex()
#>
#> -------------- middle earth example --------------
#>
#> Objects loaded:
#> *lotr_vcf* vcfR object (100 variants x 100 samples)
#> *lotr_coords* dataframe with x and y coordinates
#> *lotr_lyr* middle earth RasterLayer (100 x 100)
#> *lotr_range* SpatialPolygonsDataFrame of spp range
#>
#> -------------------------------------------------
```

```
#>

# Genetic data
lotr_vcf
#> Loading required package: vcfR
#> Warning: package 'vcfR' was built under R version 4.1.3
#>
#>    *****       ***   vcfR   ***       *****
#>    This is vcfR 1.14.0
#>      browseVignettes('vcfR') # Documentation
#>      citation('vcfR') # Citation
#>    *****       *****      *****       *****
#> ***** Object of Class vcfR *****
#> 100 samples
#> 1 CHROMs
#> 100 variants
#> Object size: 0.1 Mb
#> 0 percent missing data
#> *****        *****         *****

# Coordinates
head(lotr_coords)
#>             x         y
#> 538  88.73547 -66.61610
#> 1397 78.50479 -23.24048
#> 1200 14.32163 -25.99363
#> 952  89.86373 -65.49860
#> 1177 45.42427 -23.13054
#> 383  98.89395 -13.88943

# Raster data
lotr_lyr
#> class      : RasterLayer
#> dimensions : 100, 100, 10000  (nrow, ncol, ncell)
#> resolution : 1, 1  (x, y)
#> extent     : 0, 100, -100, 0  (xmin, xmax, ymin, ymax)
#> crs        : NA
#> source     : memory
#> names      : lyr.1
#> values     : 0, 1  (min, max)

# Map of data
plot(lotr_lyr, col = magma(100), axes = FALSE, box = FALSE)
points(lotr_coords, col = mako(1, begin = 0.8), pch = 3, cex = 0.5)
```

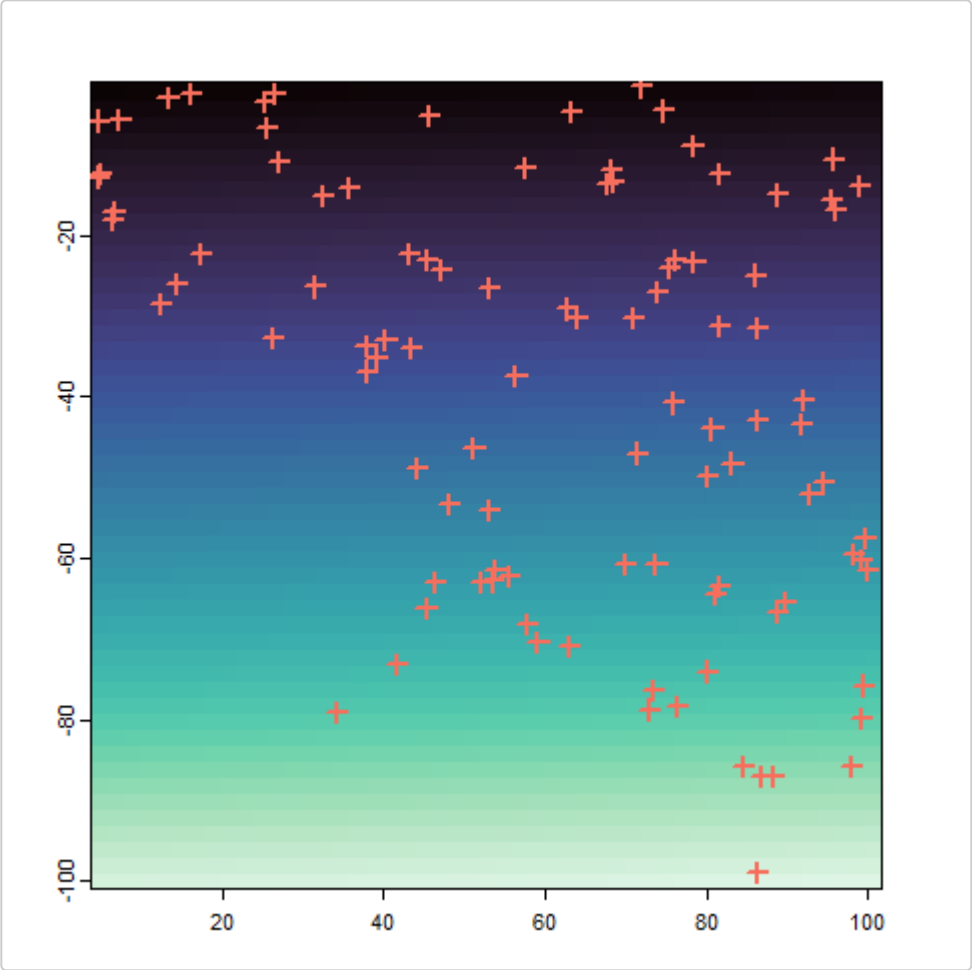


If users don't have a raster layer of their landscape, they can generate one using their sample coordinates with the `coords_to_raster()` function. The resolution of this raster can be either tuned with the `agg` (to aggregate) or `disagg` (to disaggregate) arguments, or defined using the `res` argument. The `res` argument can either be a single value (e.g., 0.00833) or a vector of two values with the x and y resolutions. The `buffer` argument can be used to add an edge to the raster (i.e., buffer away from the coordinates).

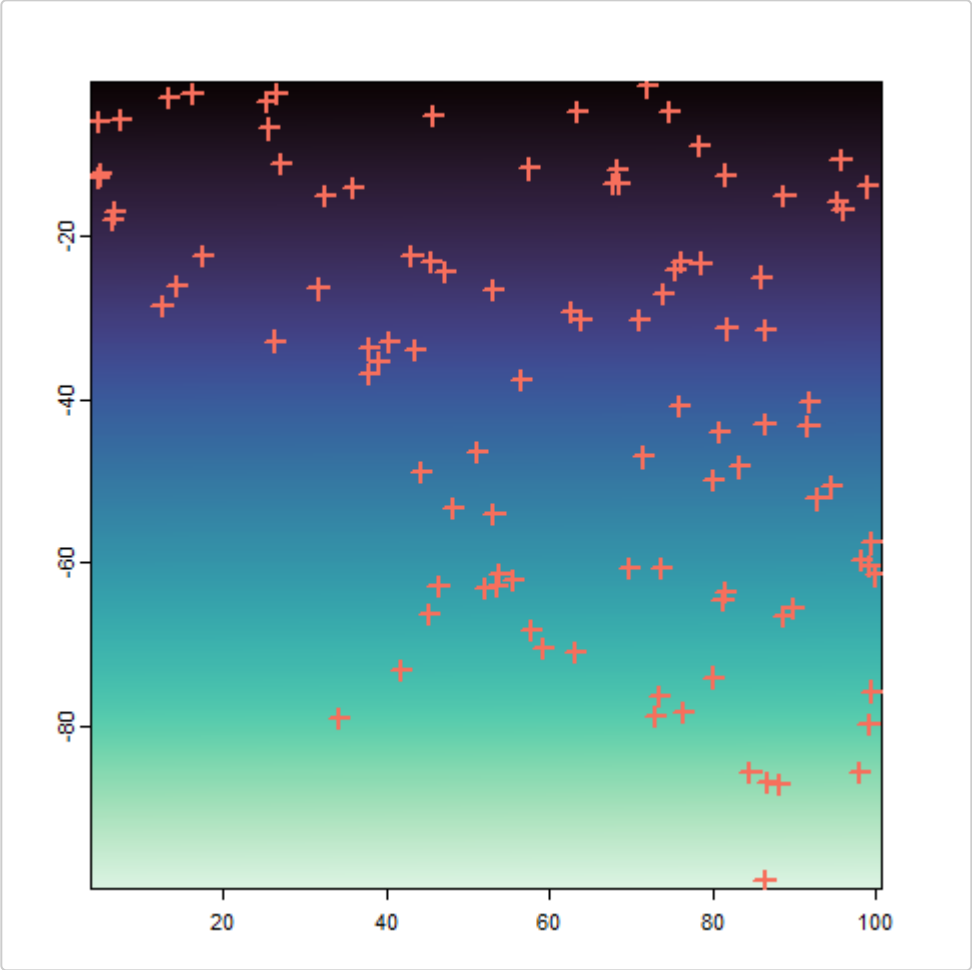
```
ex_raster1 <- coords_to_raster(lotr_coords, buffer = 1, plot = TRUE)
```



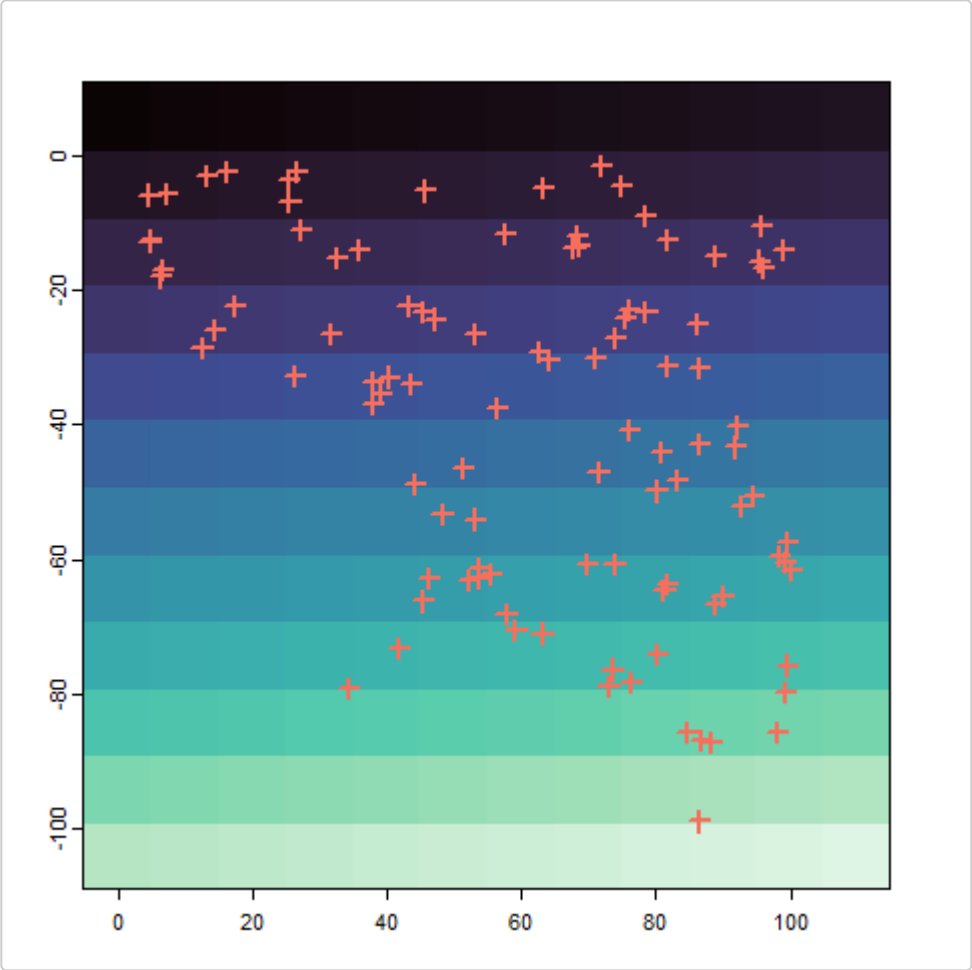

```
ex_raster2 <- coords_to_raster(lotr_coords, buffer = 1, agg = 2, plot = TRUE)
```




```
ex_raster3 <- coords_to_raster(lotr_coords, buffer = 1, disagg = 4, plot = TRUE)
```




```
ex_raster4 <- coords_to_raster(lotr_coords, buffer = 1, res = 10, plot = TRUE)
```



Workflow

The workflow of wingen uses three main functions:

1. `window_gd()` to generate moving window maps of genetic diversity
2. `krig_gd()` to use kriging to interpolate the moving window maps
3. `mask_gd()` to mask areas of the maps from (1) and (2) (e.g., to exclude areas outside the study region)

Run moving window calculations

The main arguments to `window_gd()` are:

1. `gen` - An object of type vcfR or a path to a vcf file with genotype data. The order of this file matters! The coordinate and genetic data should be in the same order, as there are currently no checks for this.
2. `coords` - sf points, or a matrix or dataframe with two columns representing the coordinates of the samples. The first column should be x and the second should be y.
3. `lyr` - A SpatRaster or RasterLayer which the window will move across to create the final map. In most cases, this will take the form of a raster of the study area.
4. `stat` - The genetic diversity summary statistic to calculate. We provide options for calculating nucleotide diversity (`stat = pi`), average allelic richness across all sites (`stat = allelic_richness` or `stat = biallelic_richness`), or average observed heterozygosity across all sites (`stat = Ho`). Currently, the

option to calculate nucleotide diversity only works for bi-allelic data. In addition, there are two functions for calculating allelic richness: (1) "`allelic_richness`" which uses the `allelic.richness()` function from the hierfstat package, and (2) "`biallelic_richness`", which provides a much faster calculation of allelic richness, but only works for bi-allelic data. When calculating "`biallelic_richness`", users can choose to rarify allele counts (as in `hierfstat::allelic.richness()`) by setting `rarify_alleles = TRUE` (the default) or to use the raw allele counts by setting `rarify_alleles = FALSE`. We recommend performing allele count rarefaction (`rarify_alleles = TRUE`) if there are missing values in the genetic data, but for datasets with no missing data, it is faster to use the raw counts (`rarify_alleles = FALSE`).

5. `fact` - To decrease computational time, we provide the option to aggregate the input raster layer by some factor defined using the `fact` argument. Increasing `fact` will decrease the number of cells and thereby decrease the number of calculations, with the trade-off that the resolution of the output layers will decrease. Users should keep in mind that if they increase `fact`, they may simultaneously want to decrease `wdim` because the proportion of the landscape covered by the neighborhood matrix could otherwise increase substantially.
6. `wdim` - Used to create the neighborhood matrix for the moving window based on the dimensions provided. This argument can either be set to one value (e.g., 3) which will create a square window (e.g., 3 x 3), or two values, which will create a rectangular window (e.g., 3 x 5). We encourage users to experiment with different values of `wdim` to determine the sensitivity of their results to this parameter. Ideally, `wdim` would be set with some knowledge of the study system in mind (e.g., the dispersal patterns and/or neighborhood size of the study organism). A preview of the window size relative to the landscape can be obtained using the `preview_gd()` function.
7. `rarify` - Users have the option to perform rarefaction by setting the `rarify` argument to `TRUE`. If `rarify = TRUE`, users then define `rarify_n` as the number of samples to rarify to and `rarify_nit` as the number of iterations for rarefaction (e.g., if `rarify_n = 4` and `rarify_nit = 5`, for each sample set, four random samples will be drawn five times). Users can also set `rarify_nit = "all"`to use all possible combinations of samples of size `rarify_n` within the window (for example, if `rarify_n = 4` and the number of samples in the window is 5, all 20 possible combinations of samples will be used). As the window moves across the landscape, three things can occur based on the number of samples in the window: (1) if the number of samples is lower than `rarify_n`, genetic diversity is not calculated and a raster value of `NA` is assigned, (2) if the number of samples is equal to `rarify_n` the genetic diversity statistic is calculated for those samples, (3) if the number of samples is greater than `rarify_n`, rarefaction is implemented and that set of samples is subsampled `rarify_nit` times to a size of `rarify_n` and the mean (or another summary statistic set using `fun`) of those `rarify_nit` iterations is used.

   We suggest that users select `rarify_n` and `rarify_nit` such that the number of possible ways to choose `rarify_n` from a sample of size `rarify_n` plus one is greater than `rarify_nit` in order for there to always be `rarify_nit` number of unique combinations of size `rarify_n`. If `rarify = FALSE`, rarefaction is not performed and only steps (1) and (2) occur such that: (1) if the number of samples in the window is less than the `min_n` argument, genetic diversity is not calculated and a raster value of `NA` is assigned, and (2) if the number of samples is equal to or greater than `min_n`, the genetic diversity statistic is calculated for those samples. We highly encourage users to perform rarefaction as genetic diversity statistics are sensitive to sample size. The main benefit of not performing rarefaction is decreased computational time; however, this is not worth the trade-off in inaccuracy unless you are confident that there is no effect of rarefaction after performing your analysis with and without rarefaction.
8. `parallel` - In order to increase computational efficiency, we provide the option for parallelization by setting the `parallel` argument to `TRUE`. Parallelization is performed using the furrr package. Use the `ncores` argument to set the number of cores to use for parallelization.
9. `crop_edges` - Whether to crop out the cells along the edge of the raster that are not surrounded by a full window. Users may want to do so to avoid "edge effects" caused by incomplete windows along the

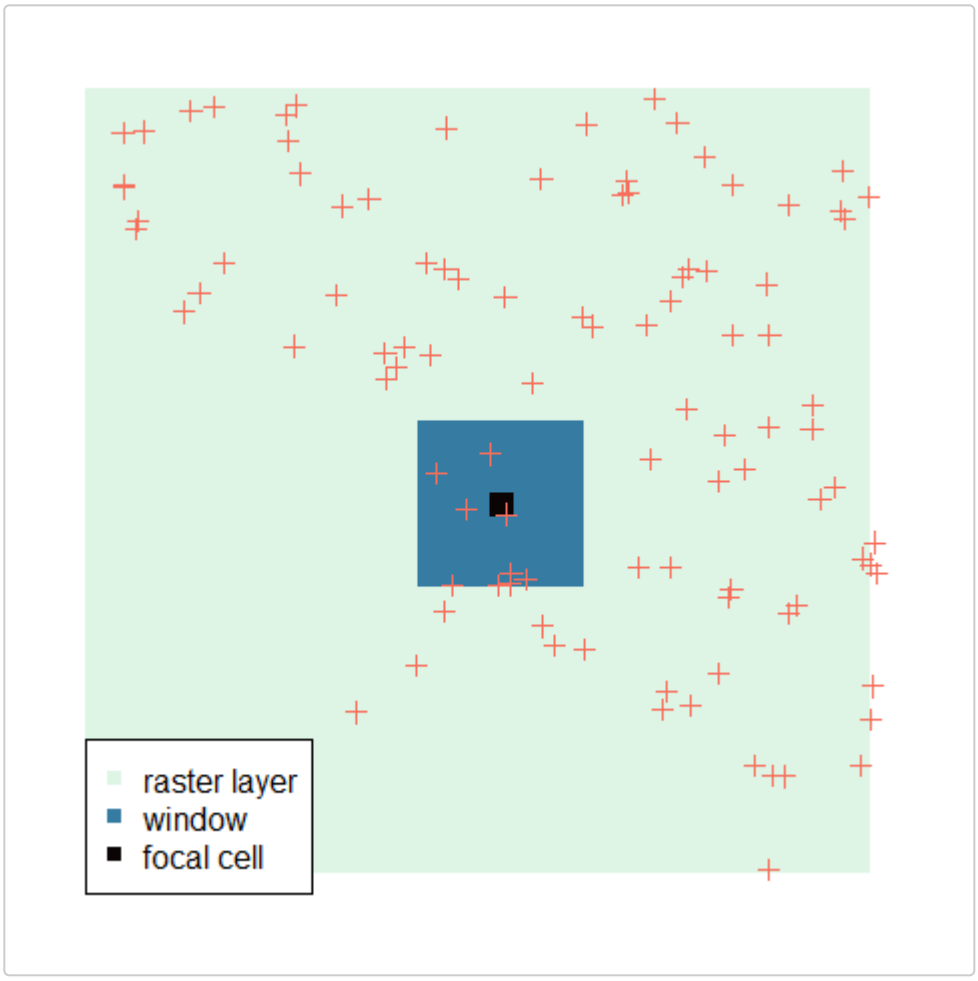
borders of the raster. As wingen is relatively insensitive to sample size, edge effects are not likely to have a very strong effect on diversity estimates, so by default `crop_edges = FALSE`.

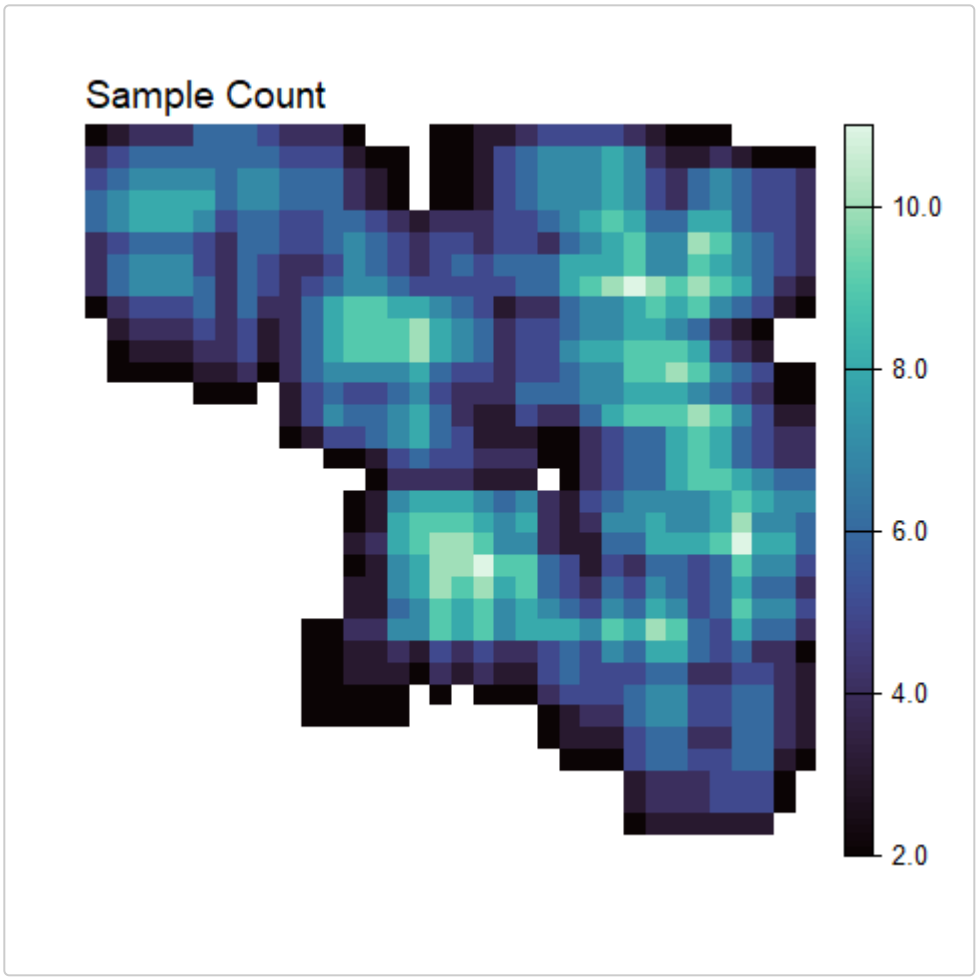
Note: Coordinates and rasters used in wingen should be in a projected (planar) coordinate system such that raster cells are of equal sizes. Therefore, spherical systems (including latitude-longitude coordinate systems) should be projected prior to use. An example of how this can be accomplished is shown below. If no CRS is provided, a warning will be given and wingen will assume the data are provided in a projected system.

```
# First, we create example latitude-longitude coordinates and rasters

## Example raster:
lyr_longlat <- rast(ncols = 40, nrows = 40, xmin = -110, xmax = -90, ymin = 40, ymax = 60,
                    crs = "+proj=longlat +datum=WGS84")

## Example coordinates:
coords_df <- data.frame(x = c(-110, -90), y = c(40, 60))
coords_longlat <- st_as_sf(coords_df, coords = c("x", "y"), crs = "+proj=longlat")

# Next, the coordinates and raster can be projected to an equal area projection, in this case the
         Goode Homolosine projection (https://proj.org/operations/projections/goode.html):
coords_eq <- st_transform(coords_longlat, crs = "+proj=goode")
lyr_eq <- project(lyr_longlat, "+proj=goode")

# Coordinates can be switched back to latitude-longitude the same way by replacing "goode" with
         "longlat"
```

Before running `window_gd()`, users can preview the moving window and the counts within each cell of the raster to get a sense of how big the window is and what the density of counts looks like across their landscape. Here, we provide the raster layer (`lotr_lyr`), the coordinates (`lotr_coords`), the window dimensions (7), the aggregation factor (3), and the minimum sample number (`min_n`). `min_n` will be used to mask the sample count layer to show how much of the landscape will be excluded due to low sample counts.

```
preview_gd(lotr_lyr,
  lotr_coords,
  wdim = 7,
  fact = 3,
  sample_count = TRUE,
  min_n = 2
)
```


raster layer
window
focal cell




```
#> class       : SpatRaster
#> dimensions  : 34, 34, 1  (nrow, ncol, nlyr)
#> resolution  : 3, 3  (x, y)
#> extent      : 0, 102, -102, 0  (xmin, xmax, ymin, ymax)
#> coord. ref. :
#> source(s)   : memory
#> name        : sample_count
#> min value   :            2
#> max value   :           11
```

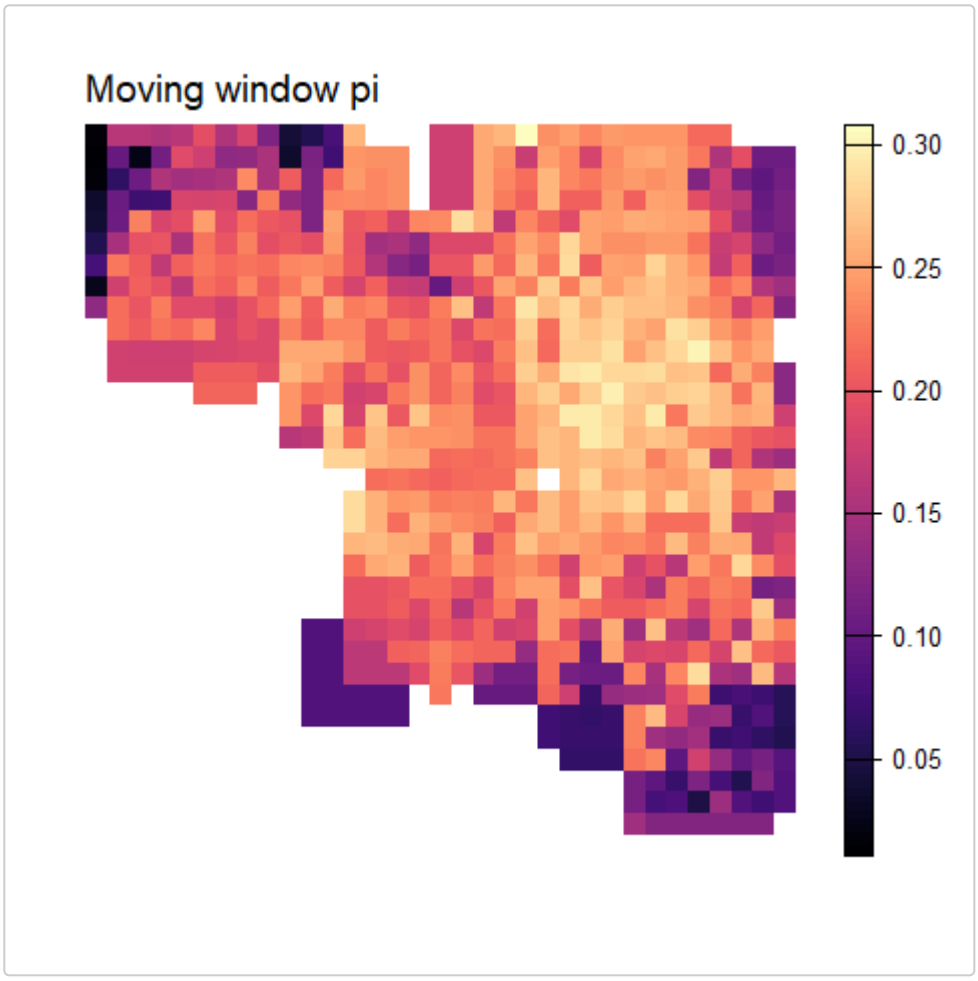
Next, we run the moving window function with our vcf, coordinates, and raster layer. Here, we set the parameters to calculate pi, use a window size of 7 x 7, an aggregation factor of 3, and rarefaction with a rarefaction size of 2 (i.e., minimum sample size of 2), and 5 iterations.

The `L` argument is used in the calculation of pi. If `L = NULL` (default), the function returns the sum over SNPs of nucleotide diversity. Otherwise, the function returns the average nucleotide diversity per nucleotide given the length L of the sequence. Users may want to set `L` differently depending on amounts of missing data, for example, so that missing sites are not considered in the calculation.

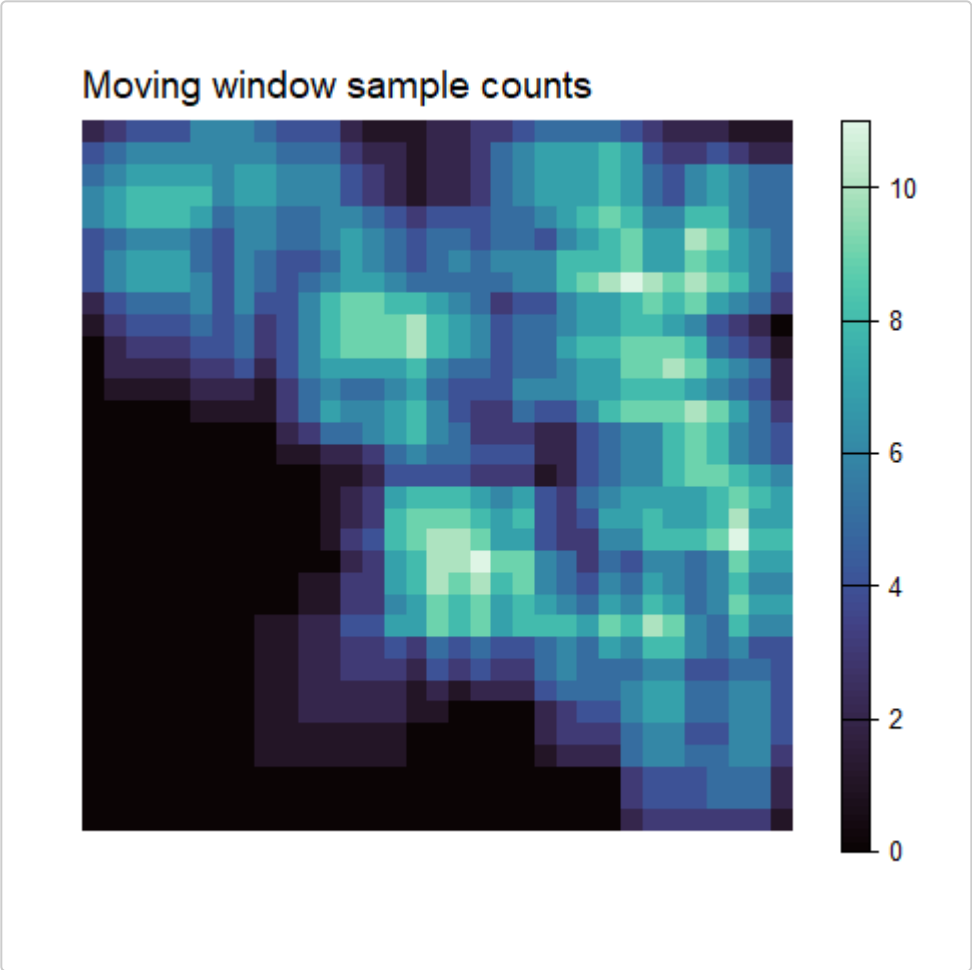
We then plot the genetic diversity layer (the first layer of the produced raster stack) and the sample counts layer (the second layer).

Note: you will get a warning that no CRS is found for the provided coordinates or raster and to check that the CRS for these objects match. This is because these simulated data doesn't have a CRS, but we know they match, so we can ignore this warning

```
wgd <- window_gd(lotr_vcf,
  lotr_coords,
  lotr_lyr,
  stat = "pi",
  wdim = 7,
  fact = 3,
  rarify = TRUE,
  rarify_n = 2,
  rarify_nit = 5,
  L = 100
)
#> Loading required namespace: adegenet
#> Warning in crs_check_window(lyr, coords): No CRS found for the provided
#> coordinates. Make sure the coordinates and the raster have the same projection
#> (see function details or wingen vignette)
#> Warning in crs_check_window(lyr, coords): No CRS found for the provided raster.
#> Make sure the coordinates and the raster have the same projection (see function
#> details or wingen vignette)


par(pty = "s")
# The plot_gd function plots the genetic diversity layer
plot_gd(wgd, main = "Moving window pi")
```



```
# The plot_count function plots the sample count layer
plot_count(wgd, main = "Moving window sample counts")
```



Krige results

To produce smoother maps of genetic diversity, we provide the function `krig_gd()` which creates a spatially interpolated raster from the moving window raster produced by `window_gd()`. This function uses the `autoKrige()` function from the R package automap to perform kriging on the moving window raster using an automatically generated variogram. Note that the raster stack from `window_gd()`, including both the genetic diversity layer and the sample count layer, can be used to generate kriged maps of both genetic diversity and sample count.

Kriging is performed by first transforming the moving window layer into a set of coordinates with corresponding genetic diversity (or sample count) values and then interpolating using these coordinates across the grid provided. Because of this, it is important to keep in mind how the coordinates from the moving window raster and the grid align. If the resolution of the moving window raster is much lower than that of the grid, there are fewer points for interpolation which can result in grid-like artifacts during kriging.

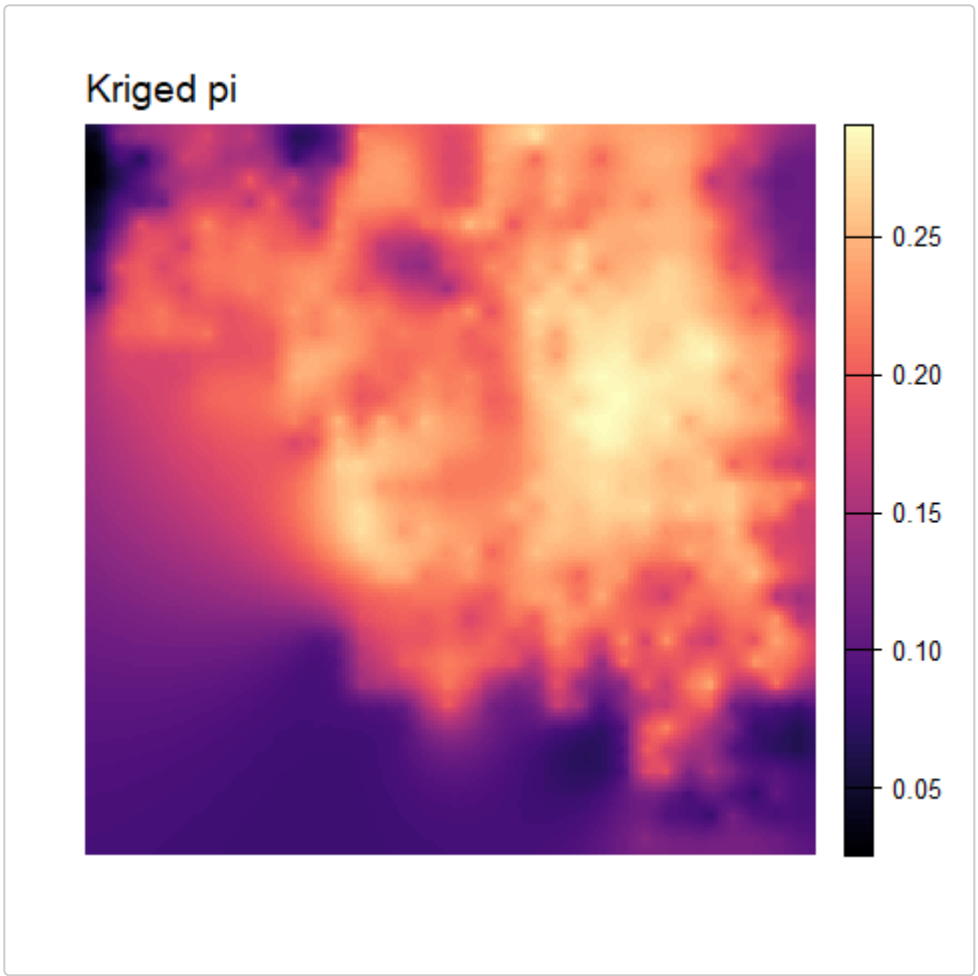
To deal with this issue, users can either (1) resample their moving window raster layers and grid layers to the same resolution using the `resample` argument, or (2) manually disaggregate or aggregate either the moving window or grid layers using the `r_agg`, `r_disagg`, `grd_agg`, or `grd_disagg` arguments. Generally, if users want a smoother resulting surface, a higher resolution grid layer should be used (this can be accomplished by using the `grd_disagg` argument to disaggregate the grid layer). The resampling, aggregation, and disaggregation options currently only work if the object provided to create the grid is a raster layer. Keep in mind that increasing the resolution of the moving window layer (i.e., either by resampling or disaggregating) can

increase computational time substantially as this increases the number of coordinates being used for kriging. This is also the case for increasing the resolution of the grid layer, though to a lesser extent.

To run this function, we provide the raster stack output from `window_gd()`, the indices of the layers we want to krige, and the raster layer to interpolate across. We also disaggregate the original layer by a factor of two to get a smoother output surface (users should play around with this parameter). The output of this function is a raster stack of the kriged input layers.

```
# Note: this step can take a little while
# index = 1 kriges the first layer in wgd (the genetic diversity layer)
kgd <- krig_gd(wgd, index = 1, lotr_lyr, disagg_grd = 2)
#> [using ordinary kriging]
```

```
par(pty = "s")
plot_gd(kgd, main = "Kriged pi")
```

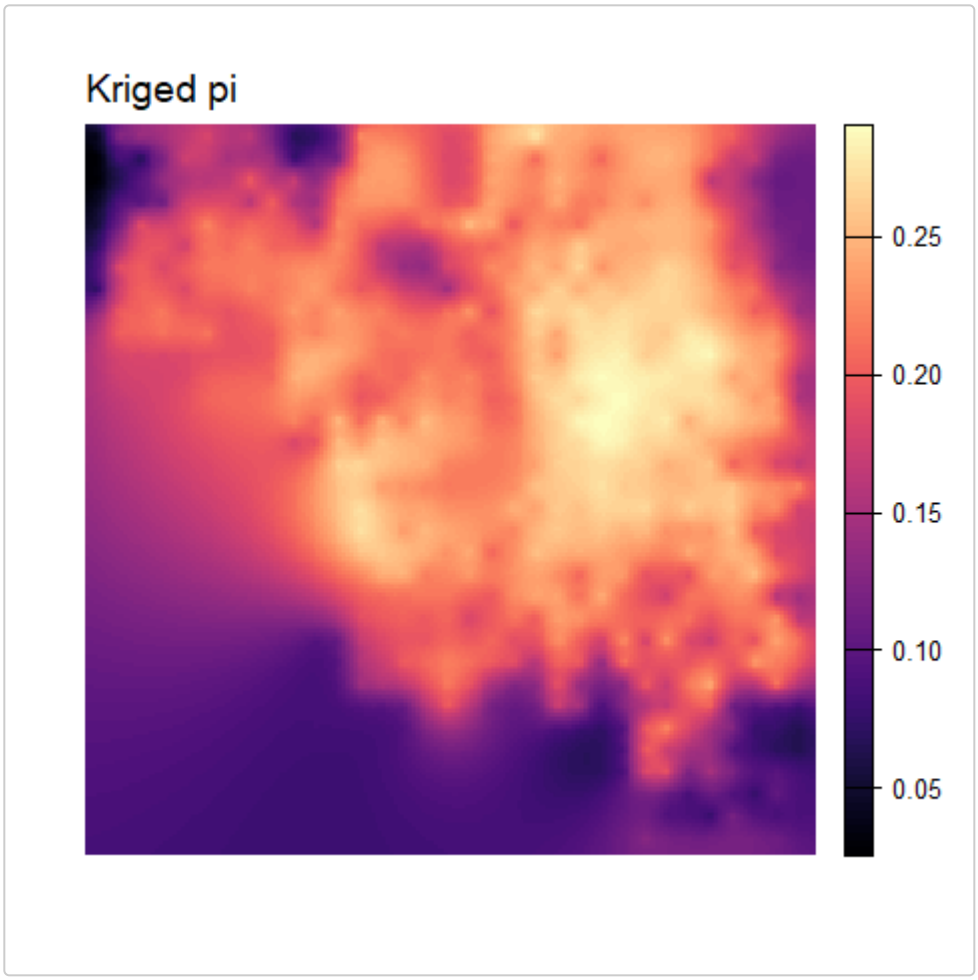


Note: By default, the kriging method is set to `krig_method = "universal"` for universal kriging (model formula: ~ x + y), which allows the mean to vary across locations, while variance is held constant. Alternatively, `krig_method` can be set to `"ordinary"` for ordinary/simple kriging (model formula: ~ 1) which assumes that the mean and variance are constant across space.

For information on the differences between universal and ordinary kriging, we recommend these articles:

- What are the different kriging models?
- Kriging interpolation

```
kgd_ordinary <- krig_gd(wgd, index = 1, lotr_lyr, disagg_grd = 2, krig_method = "ordinary")
#> [using ordinary kriging]

par(pty = "s")
plot_gd(kgd_ordinary, main = "Kriged pi")
```



Users can optionally get the full output from `autoKrige()`

```
kgd_autoKrige <- krig_gd(wgd, index = 1, lotr_lyr, disagg_grd = 2, autoKrige_output = TRUE)
#> [using ordinary kriging]

summary(kgd_autoKrige)
#>                  Length Class      Mode
#> raster           1      SpatRaster S4
#> autoKrige_output 4      autoKrige  list
```

Note: `autoKrige()` does not work for non-projected systems (i.e., latitude-longitude) and will throw an error. See above example of how latitude-longitude coordinates and rasters can be transformed.

Note: Kriging can produce values that fall outside the range of possible values (e.g., genetic diversity values less than 0 or greater than the possible maximum). By default, in order to ensure that the values are bounded within the range of possible values, `krig_gd` converts all values greater than the maximum of the input raster to that maximum (`upper_bound = TRUE`), and all values less than the minimum of the input raster to that minimum (`lower_bound = TRUE`). Users can turn off this functionality by setting `lower_bound` and `upper_bound` to `FALSE`.

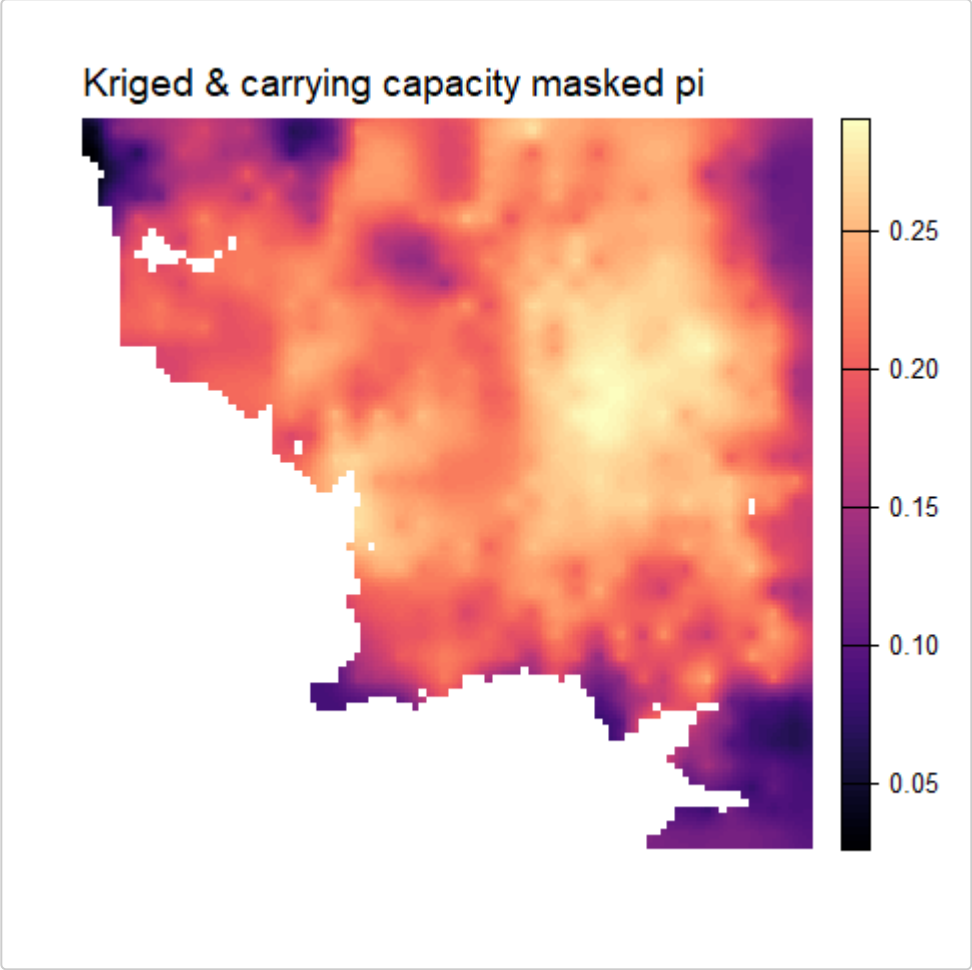
Users can also set their own custom `lower_bound` and `upper_bound` values (e.g., for heterozygosity or pi, you may want to set `lower_bound = 0` and `upper_bound = 1`).

Mask results

Next, we mask the resulting kriged layers. Masking can be performed using a variety of methods.

Method 1: Mask using the carrying capacity layer to exclude any areas where the carrying capacity is lower than 0.01. Alternatively, one could use a species distribution model or habitat suitability model to exclude areas where the probability of presence is very low:

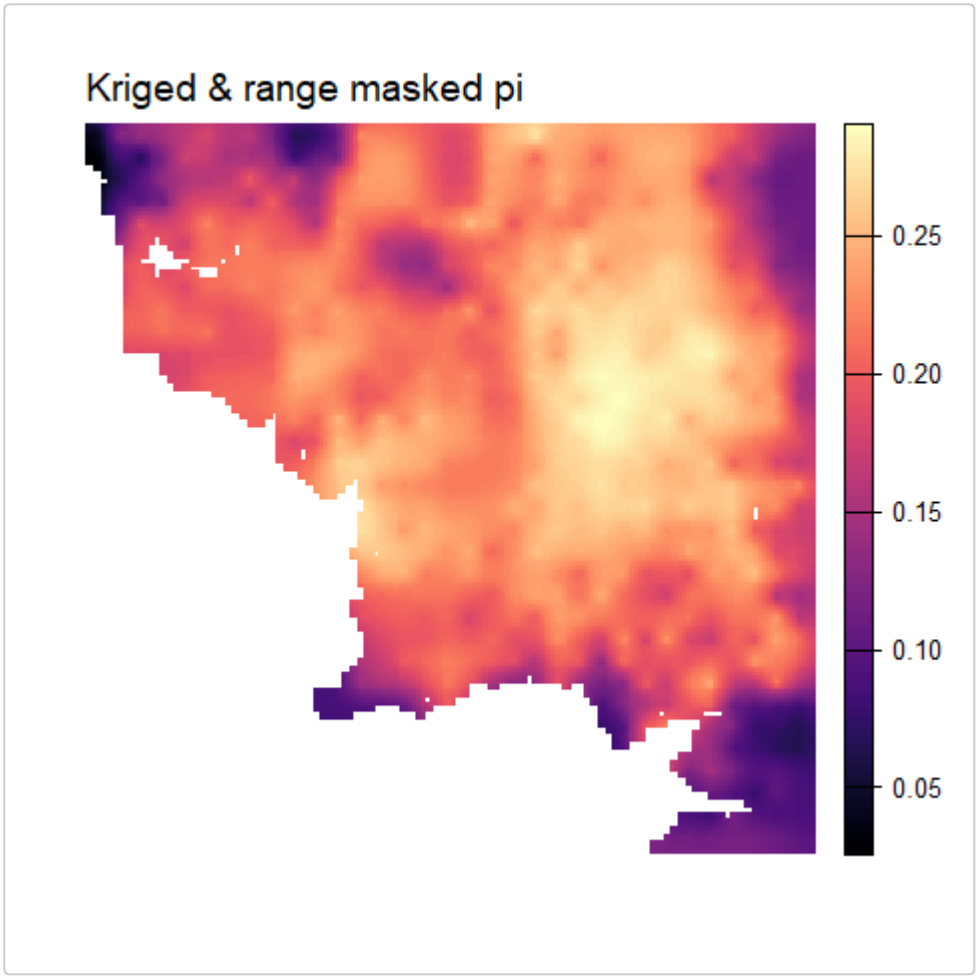
```
# disaggregate lotr_lyr to make it the same resolution as kgd before masking
## note: lotr_lyr is a RasterLayer which we convert to a SpatRaster with rast()
mask_lyr <- disagg(rast(lotr_lyr), 2)
mgd <- mask_gd(kgd, mask_lyr, minval = 0.01)

par(pty = "s")
plot_gd(mgd, main = "Kriged & carrying capacity masked pi")
```



Method 2: Mask the layer using a species range map (in this case, an sf polygon) to exclude areas falling outside the species range.

```
mgd <- mask_gd(kgd, lotr_range)
```

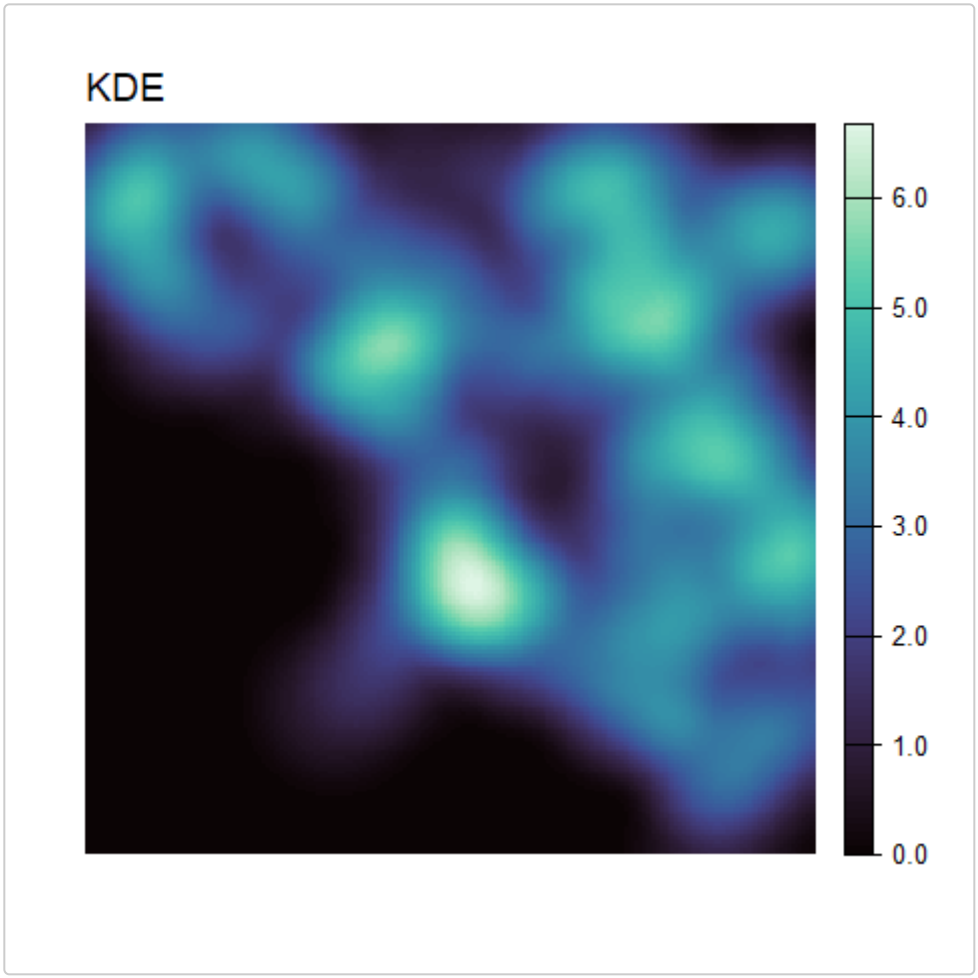
```
par(pty = "s")
plot_gd(mgd, main = "Kriged & range masked pi")
```



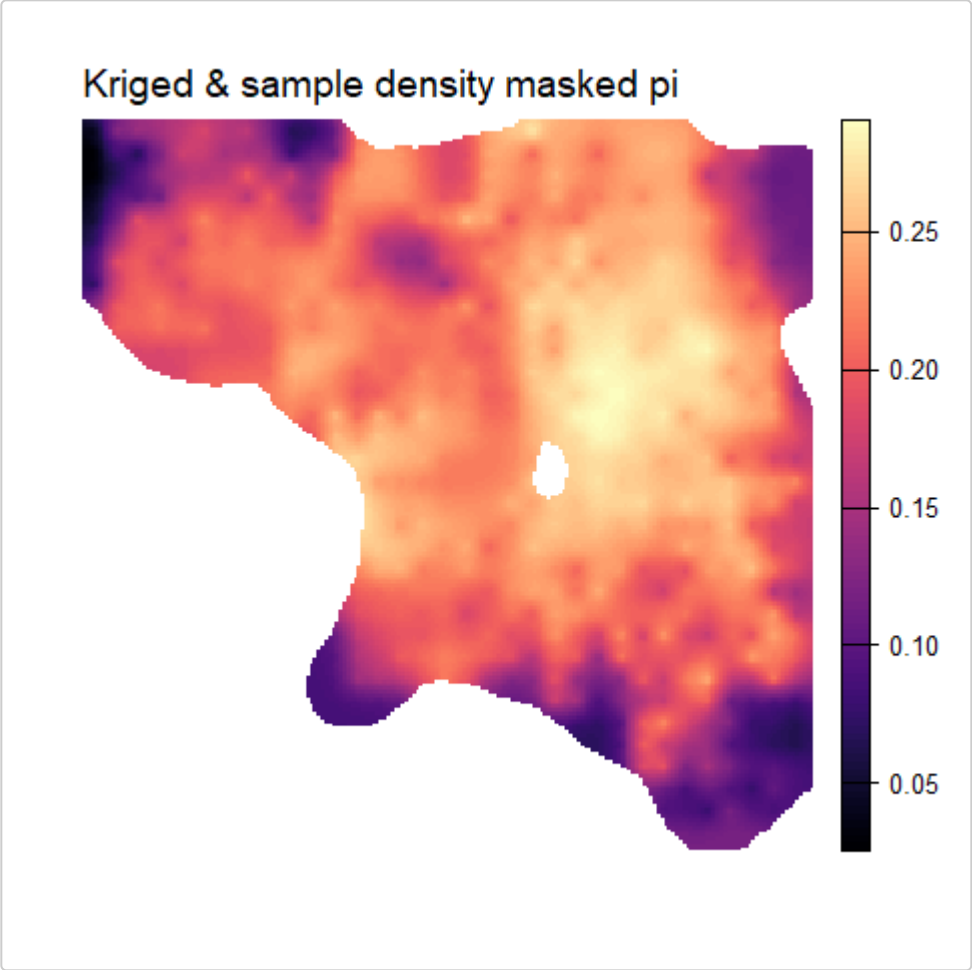
Method 3: Mask the layer using a spatial Kernel Density Estimation (KDE) of sample density to exclude areas with low sampling density using the SpatialKDE package:

```
# Spatial KDE requires an sf data.frame containing only POINTS that is in a projected coordinate
        system.
# The simulated coordinates are not projected, but for the purpose of this example we pretend they
        are projected under the Goode Homolosine projection,
# This kind of arbitrary setting of crs should not be done for real data (see above example for how
        to properly project coordinates)
lotr_sf <- st_as_sf(lotr_coords, coords = c("x", "y")) %>% st_set_crs("+proj=goode")

# The grid layer must also be a RasterLayer for SpatialKDE
# Here, we use the kriged raster as our grid to get a KDE layer of the same spatial extent and
        resolution
grid <- raster(kgd)

# See the SpatialKDE package for more options and details about using KDE
kde_lyr <- kde(
  lotr_sf,
  kernel = "quartic",
  band_width = 15,
  decay = 1,
  grid = grid,
```

```
)

par(pty = "s")
# Visualize KDE layer
plot_count(kde_lyr, main = "KDE")
```

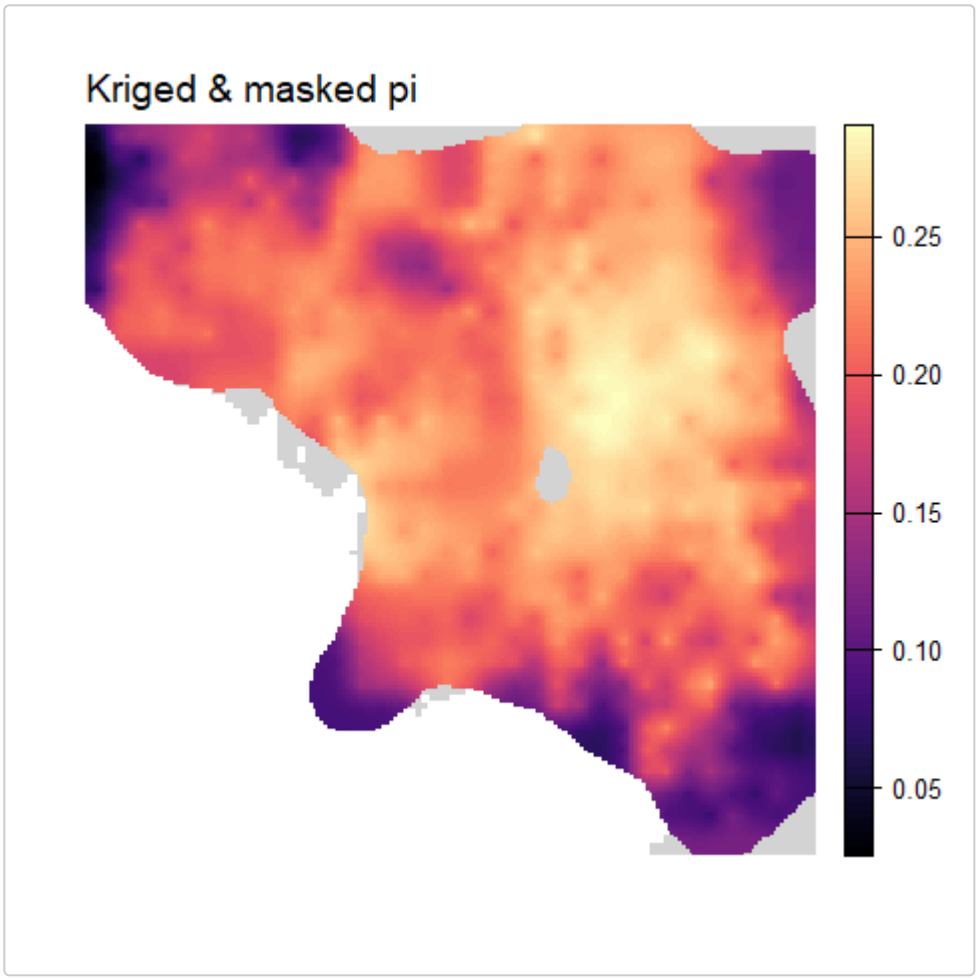



```
# Mask with mask_gd
mgd <- mask_gd(kgd, kde_lyr, minval = 1)

plot_gd(mgd, main = "Kriged & sample density masked pi")
```



Another nice visualization option is to add a “background” to your plots in the form of a raster or other sp or sf object (e.g., a country or range boundary) which can help provide geographic context:

```
par(pty = "s")
plot_gd(mgd, bkg = lotr_range, main = "Kriged & masked pi")
```



Parallelization

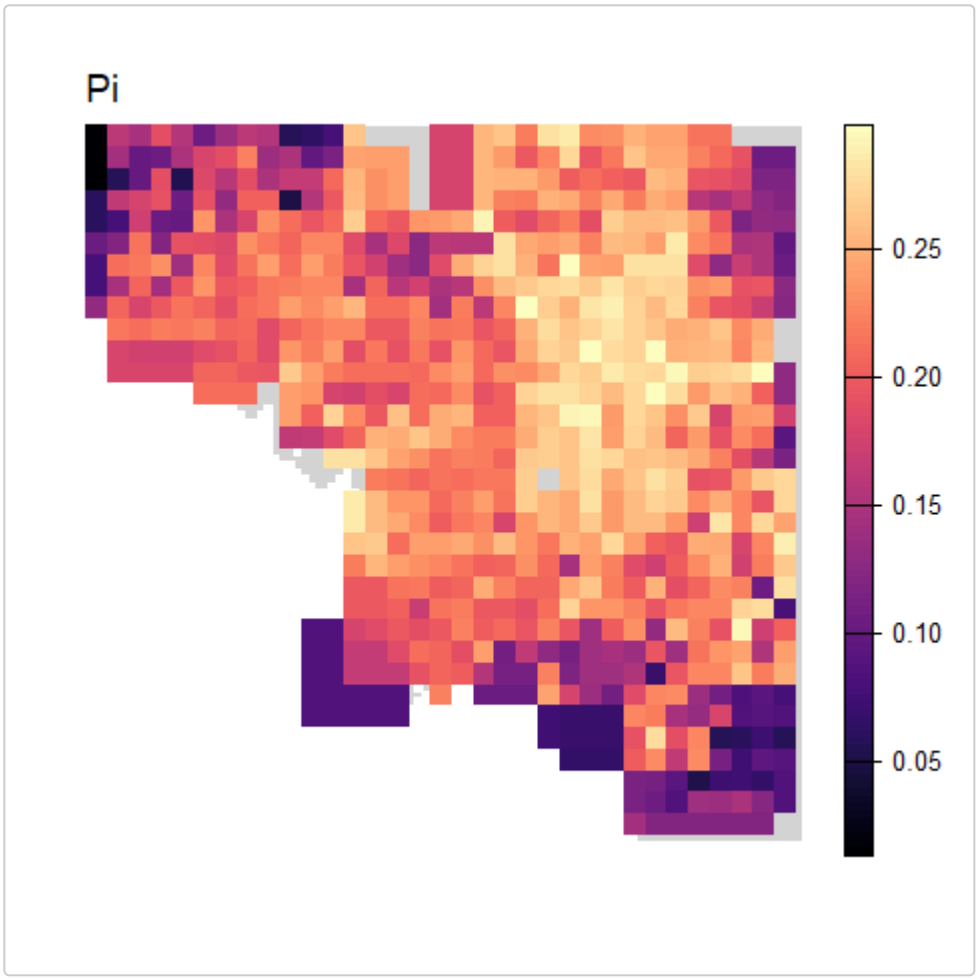
To increase computational speed, users can perform the `window_gd()` calculations with parallelization by setting `parallel = TRUE` and `ncores` to the number of cores to use:

```
# Note: this code is not evaluated when building the vignette as it spawns multiple processes
wgd <- window_gd(lotr_vcf,
  lotr_coords,
  lotr_lyr,
  stat = "pi",
  wdim = 7,
  fact = 3,
  rarify_n = 2,
  rarify_nit = 5,
  rarify = TRUE,
  parallel = TRUE,
  ncores = 2
)
```

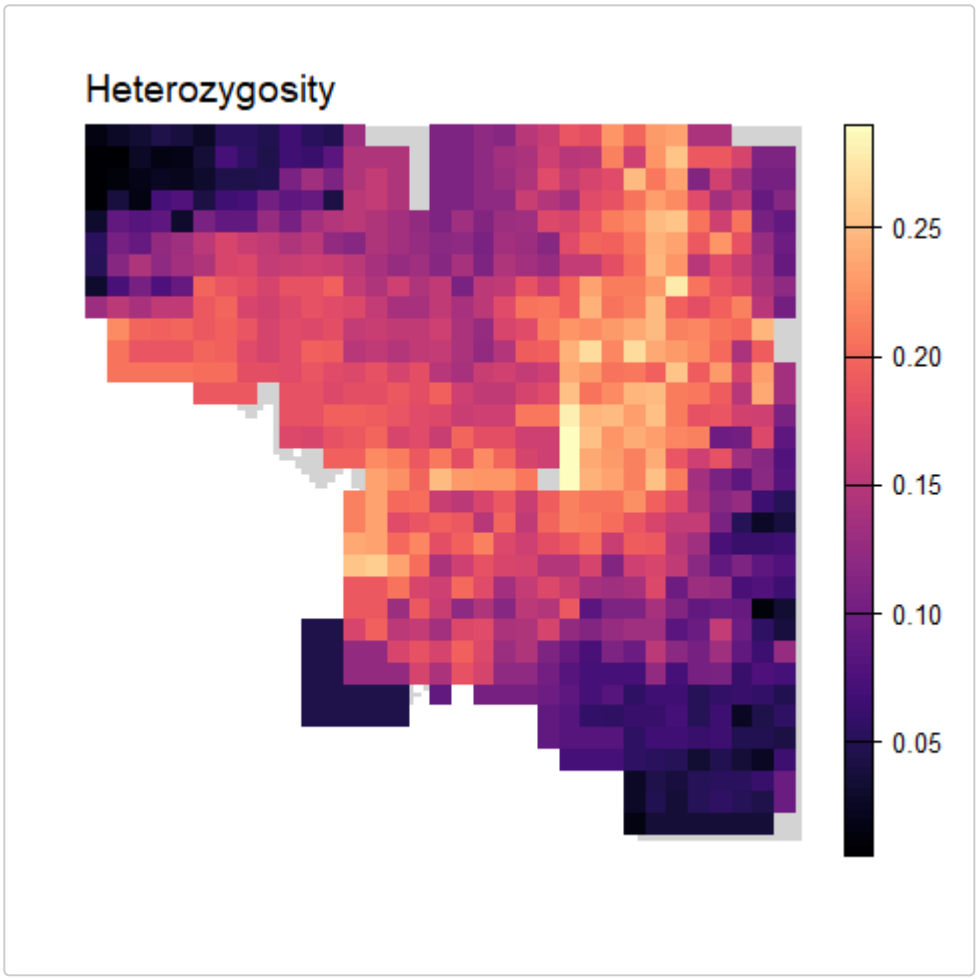
Other genetic diversity metrics

In addition to pi, users can calculate observed heterozygosity (`"Ho"`), allelic richness (either using `"biallelic_richness"` [faster but only works on biallelic data] or `"allelic_richness"`):

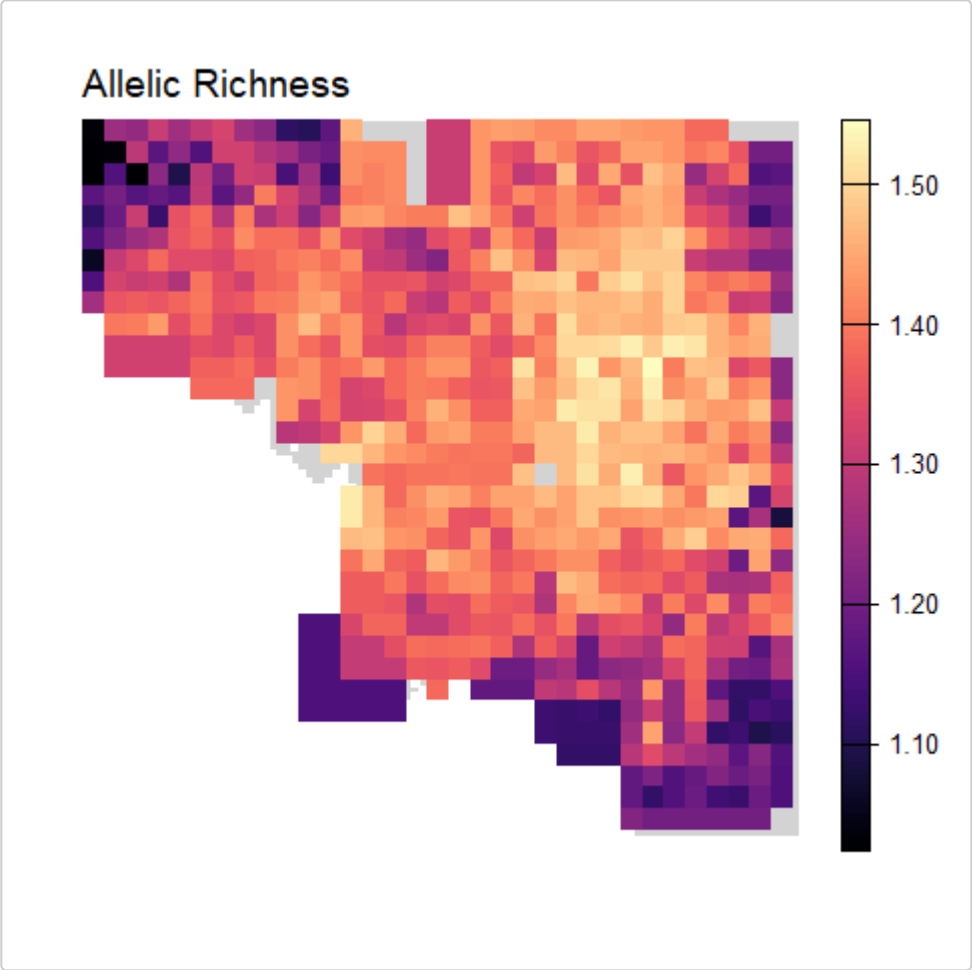
```
pi_wgd <- window_gd(lotr_vcf,
  lotr_coords,
  lotr_lyr,
  stat = "pi",
  wdim = 7,
  fact = 3,
  rarify_n = 2,
  rarify_nit = 5,
  rarify = TRUE,
  L = 100
)

het_wgd <- window_gd(lotr_vcf,
  lotr_coords,
  lotr_lyr,
  stat = "Ho",
  wdim = 7,
  fact = 3,
  rarify_n = 2,
  rarify_nit = 5,
  rarify = TRUE
)

AR_wgd <- window_gd(lotr_vcf,
  lotr_coords,
  lotr_lyr,
  stat = "biallelic_richness",
  wdim = 7,
  fact = 3,
  rarify_n = 2,
  rarify_nit = 5,
  rarify = TRUE
)


par(pty = "s")
plot_gd(pi_wgd, bkg = lotr_range, main = "Pi")
```




```
plot_gd(het_wgd, bkg = lotr_range, main = "Heterozygosity")
```




```
plot_gd(AR_wgd, bkg = lotr_range, main = "Allelic Richness")
```



General moving window

We provide a `window_general` function that can be used to make moving window maps for other types of data inputs and functions. Unlike `window_gd`, `window_general` does not require a `vcfR` or a path to a vcf file as input.

The required input (`x`) depends on the statistic (`stat`) being calculated.

For the standard genetic diversity statistics:

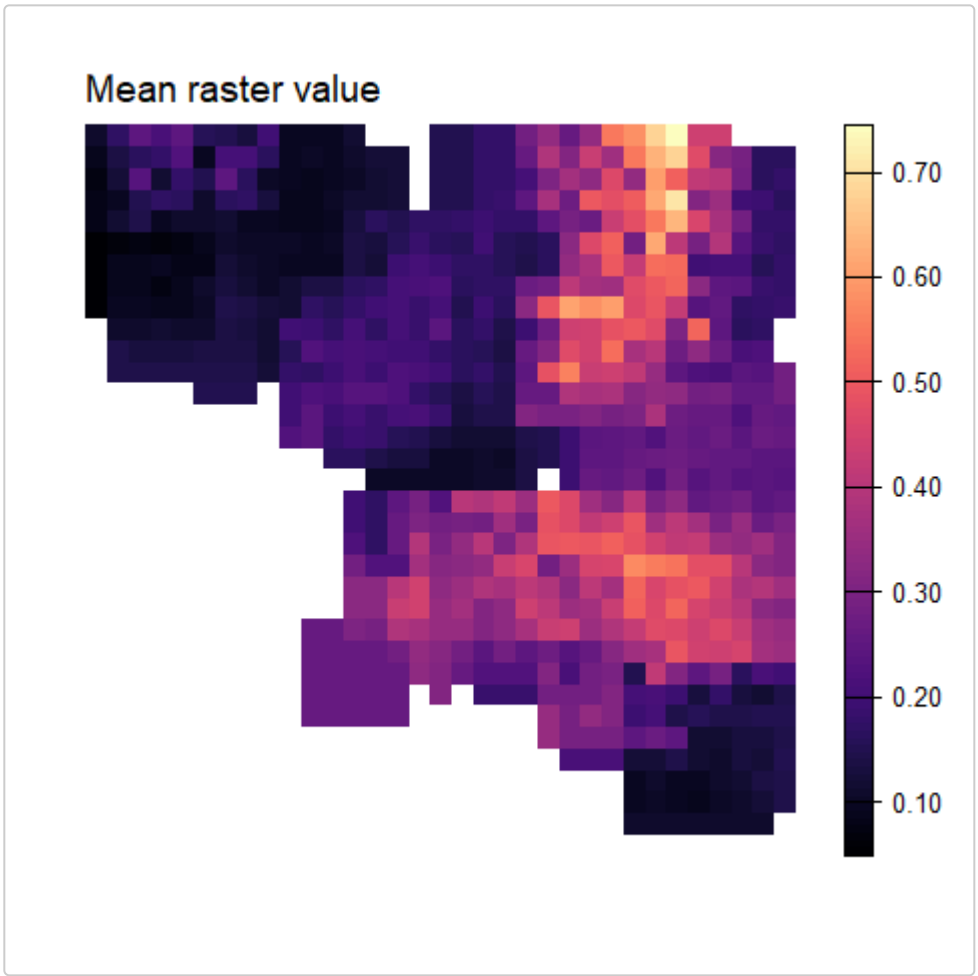
- If `stat = pi` or `biallelic_richness`, `x` must be a dosage matrix with values of 0, 1, or 2 (*note:* rows must be individuals).
- If `stat = Ho`, `x` must be a heterozygosity matrix where values of 0 = homozygosity and values of 1 = heterozygosity (*note:* rows must be individuals).
- If `stat = allelic_richness`, `x` must be a `genind` type object.

For other statistics:

- If `x` is a vector, `stat` can be any function that can be applied to a vector (e.g., `stat = mean, var, sum, etc.`).
- If `x` is a matrix or data frame (*note:* rows must be individuals), `stat` can be any function that takes a matrix or data frame and outputs a single numeric value (e.g., a function that produces a custom diversity index) *(note: this functionality has not have been tested extensively and may produce errors, so use with caution)*.

As an example, let's create a moving window map of our raster layer values (e.g., carrying capacity and conductance, in this case) at the sample coordinates:

```
# First, we extract the raster values at those coordinates
vals <- extract(lotr_lyr, lotr_coords)

# Next, we run the window_general function with the env vector and set the `stat` to mean
# Note: we can also provide additional arguments to functions, such as na.rm = TRUE
we <- window_general(vals,
  coords = lotr_coords,
  lyr = lotr_lyr,
  stat = mean,
  wdim = 7,
  fact = 3,
  rarify_n = 2,
  rarify_nit = 5,
  rarify = TRUE,
  na.rm = TRUE
)

par(pty = "s")
plot_gd(we, main = "Mean raster value")
```



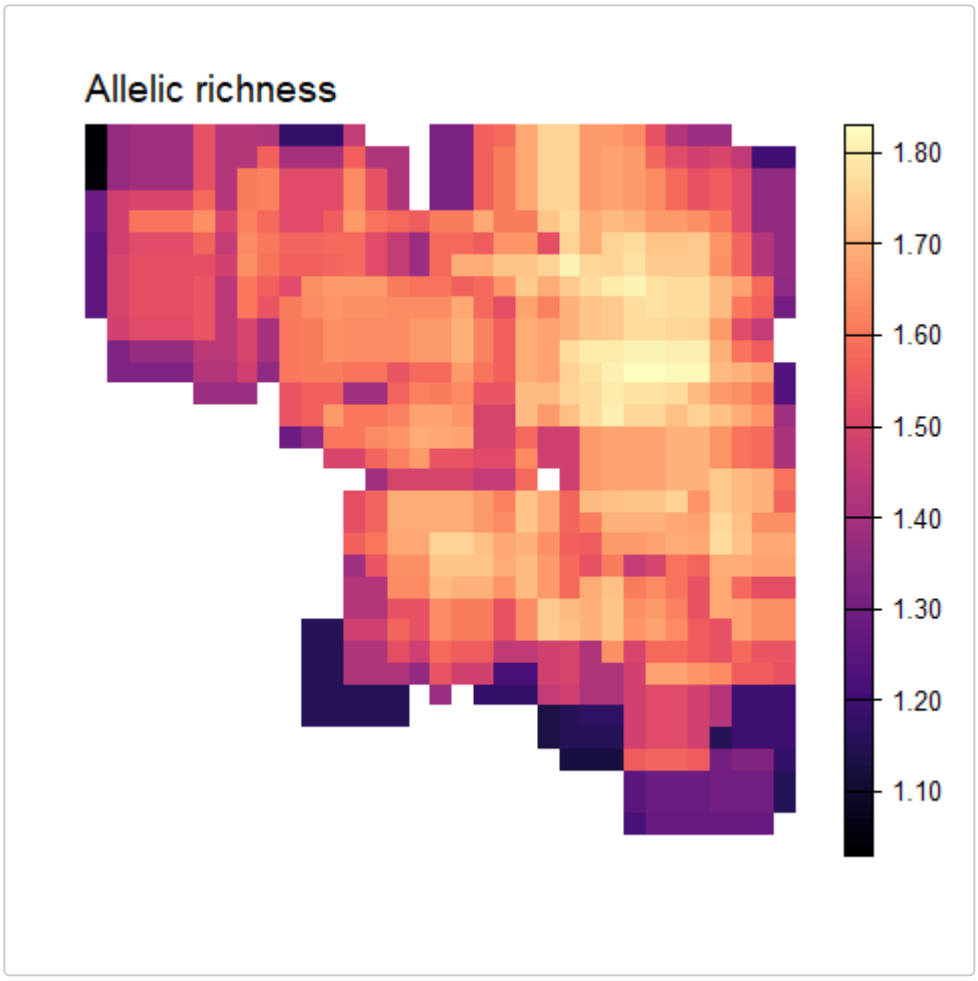
Other genetic data input file types in wingen

Although the default input data type for wingen is a VCF file, which is a standard file type to encode genomic data, wingen can accept a genind object as input to calculate allelic richness using `window_general()`. Given that some wingen functionality can be feasible using a genind object, users could easily convert various other genetic data file types to then run wingen.

For example, genepop (.gen) is a typical file format for encoding microsatellite data, in which each allele within a locus is coded using a two- or three-digit system (i.e., in a diploid organism, in a two-digit system, each locus would be assigned four digits specifying each of two alleles. In a three-digit system in the same organism, each locus would be assigned six digits encoding those same two alleles). Users can convert a genepop file into a genind object using the `read.genepop()` function in the adegenet package, which automatically reads in a .gen file as a genind object. In many cases, microsatellite data may not be biallelic, and may contain more than one observed allele at a given locus; in such cases, running `window_general()` with a genind object is still feasible.

An example of how one could use a genind object as input into `window_general()` is as follows:

```
# We use the vcfR package to convert vcf to genind for our example
library(vcfR)

# Convert existing vcf example file into genind object:
genind <- vcfR2genind(lotr_vcf)

# Run window_general with no rarefaction
we_gi <- window_general(genind,
                        coords = lotr_coords,
                        lyr = lotr_lyr,
                        stat = "allelic_richness",
                        wdim = 7,
                        fact = 3,
                        na.rm = TRUE
)

# Plot results
par(pty = "s")
plot_gd(we_gi, main = "Allelic richness")
```


Allelic richness
1.80
1.70
1.60
1.50
1.40
1.30
1.20
1.10