Introduction to the R package 'icosa' v0.10.0 for global triangular and hexagonal gridding

Adam T. Kocsis

2020-02-15

1. Introduction

The purpose of this vignette is to demonstrate the basic usage of the `icosa` package, explain object structures and basic functionalities. The primary targeted application of the package is in global biological sciences (e.g. in macroecological, biogeographical analyses), but other fields might find the structures and procedures relevant, given that they operate with point coordinate data. This is just a brief introduction to the package's capabilities. The complete documentation of the package, and relevant tutorials will be posten on the evolv-ED blog (http://www.evolv-ed.net). Previous versions of this document are available at the package's GitHub repository (https://github.com/adamkocsis/icosa/_archive/vignettes).

2. The grids

The primary problem with ecological samples is that due to density and uniformity issues, the data points are to be aggregated to distinct units. As coordinate recording is very efficient on the 2d surface of a polar coordinate system (i.e. latiude and longitude data), this was primarly achieved by rectangular gridding of the surface (for instance 1° times 1° grid cells). Unfortunately, this method suffers from systematic biasing effects: as the poles are approached, the cells become smaller in area, and come closer together.

The `icosa` package apptrroaches this problem from one of the most straightforward ways, by tessellation of a regular icosahedron to a given resolution. This procedure ends up with a polyhedral object of triangular faces of higly isometric properties: very similar shapes of cells which are roughly equally distanced, and similar in cell area.

To visualize the grids in 3 dimension, you have to have rgl package installed. This is optional, but to ensure maximum functionality when plotting with 'rgl', it is better to attach the 'rgl' package first, otherwise it will mask out some functionalities of 'icosa'. Attaching the package 'rgdal' is necessary for reading in shapefiles and transformation of projections, while the raster package is required for some lookup functions.

```r
library(rgdal)
library(raster)
library(rgl)
library(icosa)
```
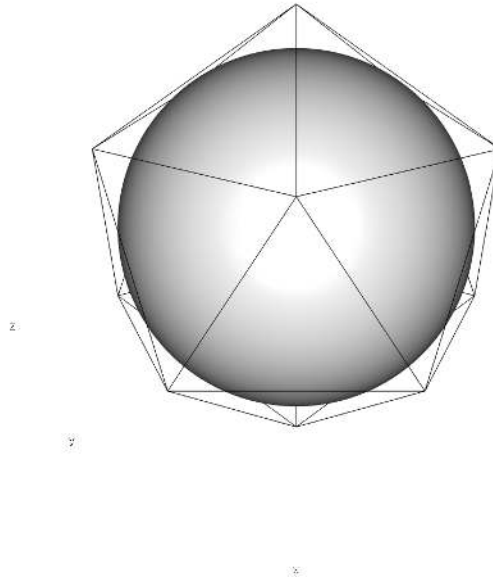
2.1. Grid creation

To create a triangular grid use the function `trigrid()`:

```r
# create a trigrid class object
tri <- trigrid()

# the show() method displays basic information of the package
tri
```

```
## A/An trigrid object with 12 vertices, 30 edges and 20 faces.
## The mean grid edge length is 7053.65 km or 63.43 degrees.
## Use plot3d() to see a 3d render.
```

```
# plot the object in 3d
plot3d(tri, guides=F)
```


Without any specified additional entry, the first line will create an icosahedron with the center of `c(0,0,0)` Cartesian coordinates and the 'R2' (authalic, as defined by IUGG (1)) radius of Earth between the object center and the vertices. These can be altered by setting the `radius` and `center` arguments if necessary. When dealing with properly georeferenced data, the model ellipsoid (or in this case, the sphere) is to be taken into account when the data and the grid interact. Therefore a slot called `proj4strig` is added to the grid object, which contains a CRS class string generated automatically from the input radius. With the default settings this is:
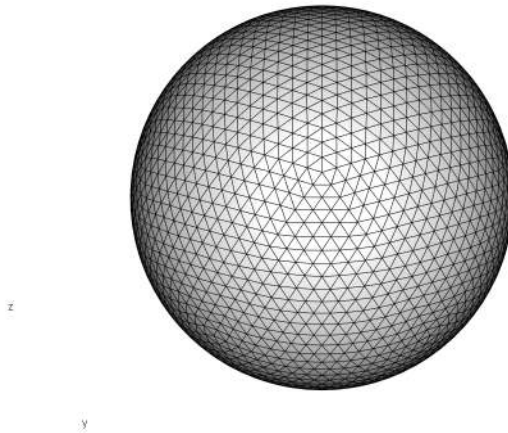
```
tri@proj4string
```

```
## CRS arguments: +proj=longlat +a=6371007 +b=6371007
```

Setting the first argument of the `trigrid()` function will create more complex objects that have tessellated faces:

```
# create a trigrid class object
gLow <- trigrid(tessellation=c(4,4))

# plot the object in 3d
plot3d(gLow, guides=F)
```

The result is another `trigrid` class object with the tessellation vector of `c(4,4)`. The tessellation vector is the primary argument influencing grid resolution. It consists of integer values which are larger than 1. These values will be passed in sequence to the tessellation function, using the result of the previous round as an input. In the example of the `c(4,4)` grid, the icosahedron will be tessellated with the value of 4 in the first round, meaning that every edge of the 20 faces are split to 4, which then results in 4 times 4 new triangular faces instead of the one original (4 times 4 times 20 new faces in total). The second round will be repeated for every newly formed face as well, so the total resolution of the grid will be 4 times 4 times 4 times 4 times 20 faces.

The obvious question is then: what is the difference between the `c(2,2,2,2))`, `c(4,4)`, `c(8,2)`, `c(2,8)` and `c(16)` grids? The answer depends on the applied tessellation method. The icosahedron itself is smaller in surface area and volume than the sphere. The points created between the faces need to be projected to the sphere, which can be done in a number of different ways.

The current version of the `icosa` package uses a single tessellation method, which requires the least amount of information to provide a consistent output: The `"meanGC"` method uses spherical functions to calculate new points directly on the great circles that connect points which are on a single edge without any sort of projection. The internal points are calculated by connecting the newly formed points on the edges. This results in some scatter for these internal points, as their position depends on the pair of edges that are connected. In this method, the points are defined as their centroids projected to the surface of the sphere, which results in a systematic increase in cell area as the center of the tessellated face is approached. Therefore, the answer to the question of the different tessellation vectors is: the number of faces will be equal as that is set by the total product of the tessellation vector, but as every tessellation round includes the above described procedure, the cell areas, cell shapes will be somewhat different with these. In the future, multiple tessellation methods are to be incorporated that produce grid cells with exactly the same areas just to mention one.

As grid complexity increases the time to create the structure increases as well (The highest resolution grid so far was the `c(10,10,4)` trigrid, which took about 2,500 seconds using a single thread of an Intel Xeon E5-1620 processor, it had 3,200,000 faces, the mean edge length of 0.17 degrees (20km) and its size was almost 2GB). Performance also becomes an issue with very large tessellation values, as they currently incorporate distance matrix calculations (will be updated later, if required).

A rectangular grid has an additional problem that is not solved by triangular replacement, which is the definition of neighbouring cells. With both the rectangular and the triangular grid, two types of possible

connections exist: cells can share either one or two vertices (an edge), which leads to problems with cell to cell relationship calculations. The inversion of the triangular grid solves this problem: if every center of the face becomes a new vertex a hexagonal pattern emerges, which creates a neighbourhood pattern where the neighbouring faces can share exactly two vertices only. Every resolution triangular grid can be turned to a penta-hexagonal one, which is directly created by the `hexagrid()` function.

```
# create a hexagrid object
hLow <- hexagrid()

# plot it in 3d
plot3d(hLow, guides=F)
```
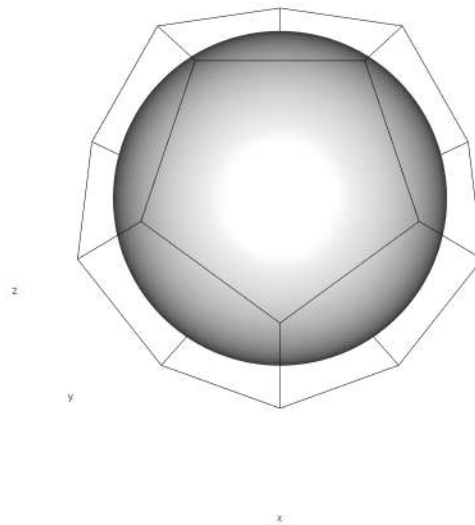

By default (`tessellation=1`), the `hexagrid()` function inverts the regular icosahedron, creating a regular pentagonal-dodecahedron. This object paradoxically has no hexagonal faces. Increasing the tessellation vector, however, will add these, while keeping the 12 pentagonal faces at the positions which were originally containing the icosahedron's vertices.

```
# create a hexagrid object
hLow <- hexagrid(c(4,4))

# plot it in 3d
plot3d(hLow)
```

4
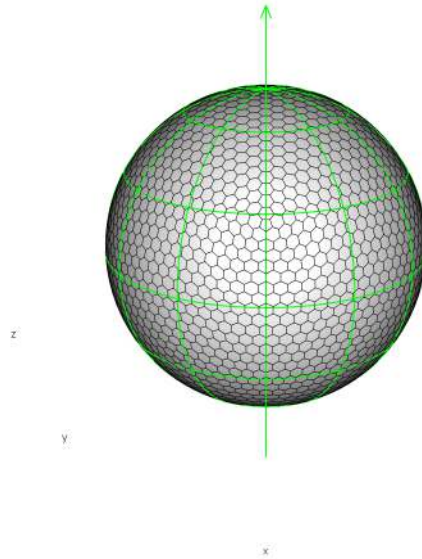
The function of the tessellation vector is exactly the same as for the `trigrid()` function, which is invoked by the `hexagrid()` function before the inversion is implemented. This naturally leads to an equality between the vertex numbers of the hexagrid and face numbers of the trigrid, and the face numbers of the trigrid and the vertex numbers of the hexagrid objects.

All methods that are implemented for the trigrid are implemented for the hexagrid as well. The examples that follow use the two types of grids at random, and work interchangably.

2.2. Grid structure

The grids implented by this package represent compound objects that have different 'dimensions'. For example, grids represent both a regular 3d object structure and an object of interconnected cells. The primary 3d structure of the grid is similar to a generic 3d .obj file structure. There are two main tables: one contains the grid vertex coordinates and the other contains which coordinates form which faces. This information is stored by the vertices and faces slots, respectively:

```
# the beginning of the vertices matrix
head(gLow@vertices)
```

```
##               x         y        z
## P1     0.0000    0.0000 6371.007
## P2 -418.9419 -136.1225 6355.760
## P3    0.0000 -440.5015 6355.760
## P4  418.9419 -136.1225 6355.760
## P5  258.9203  356.3732 6355.760
## P6 -258.9203  356.3732 6355.760
```

```
# the beginning of the faces matrix
head(gLow@faces)
```

```
##     [,1] [,2] [,3]
## F1 "P1" "P2" "P3"
## F2 "P1" "P3" "P4"
## F3 "P1" "P5" "P4"
```

```
## F4 "P1" "P5" "P6"
## F5 "P1" "P2" "P6"
## F6 "P2" "P6" "P7"
```

The information content is stored and all the calculations are executed in XYZ Cartesian space instead of a polar coordinate system. This facilitates the definition of additional projection methods, potential grid-grid interaction, 3d plotting and calculations, and it also permits higher overall flexibility. The Cartesian coordinates are based on the value of the grid radius and center.

```
# grid radius
gLow@r
```

```
## [1] 6371.007
```

```
# grid center
gLow@center
```

```
## [1] 0 0 0
```

The centers of the faces can also be directly accessed in a format that is similar to the grid vertices format:

```
head(gLow@faceCenters)
```

```
##              x          y        z
## F1 -139.7730 -192.3810 6366.568
## F2  139.7730 -192.3810 6366.568
## F3  226.1574   73.4830 6366.568
## F4    0.0000  237.7960 6366.568
## F5 -226.1574   73.4830 6366.568
## F6 -453.0188  147.1947 6353.176
```

Both the `vertices` and the `faceCenters()` slots are accessible using the shorthand functions `vertices()` and `centers()`, which also do coordinate transformations, if requested.

The vertices forming the edges (these are not ordered in the current version) can be extracted from the `edges` slot:

```
head(gLow@edges)
```

```
##    [,1]  [,2]
## E1 "P1"  "P3"
## E2 "P1"  "P4"
## E3 "P3"  "P4"
## E4 "P39" "P22"
## E5 "P39" "P40"
## E6 "P22" "P40"
```

Each grid has an orientation which is stored in the `orientation` slot. The values are in radians, and denote the xyz rotation relative to the default. The faces and vertices table are organized so that both vertices and faces spiral down from the zenith point to the nadir. This can be visualized in 3d using the `gridlabs3d()` function.

```
plot3d(gLow)
gridlabs3d(gLow, type="v", col="blue", cex=0.6)
```
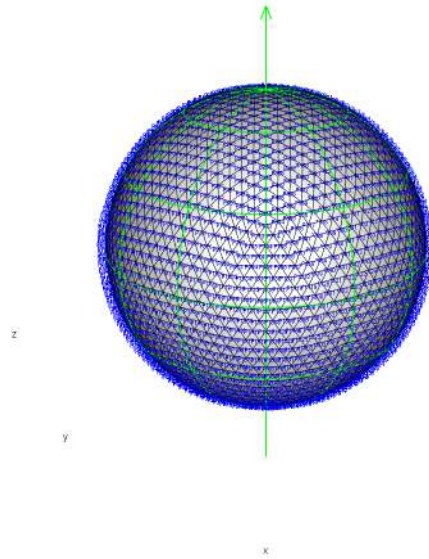
The grid orientation can be changed using the `rotate()` function. To see the effect of this on the 3d plots, compare the orientations of the grids using the `guides3d()` function that displays the polar gridding oriented to match the cartesian coordinate system.

```
gLow2 <- rotate(gLow) # random rotation
plot3d(gLow2)
guides3d(col="green")
```


7
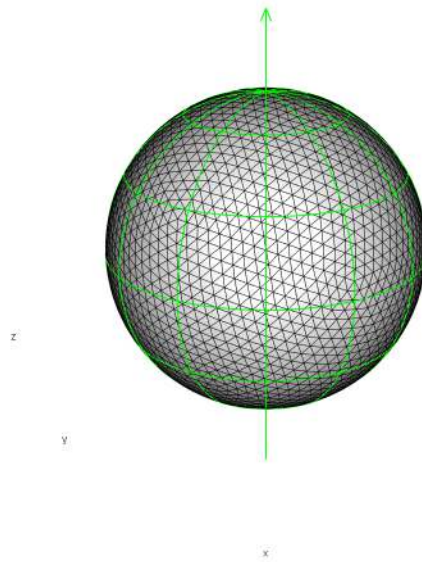
2.3. Plotting
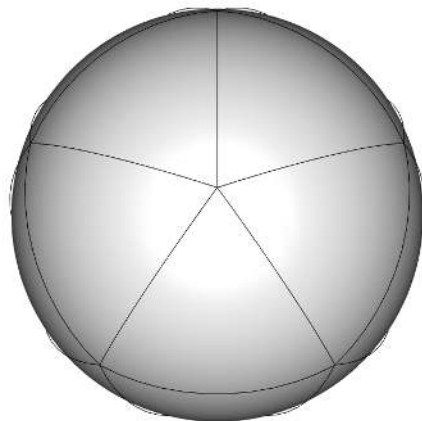
2.3.1. Three-dimensional plots

Both 3d and 2d plotting are incorporated in the package. As the grid structure exists in 3d space, 3d is the default plotting scheme which is implemented with the package `rgl`'. All 3d plotting functions pass arguments to either the `points3d()`, `segments3d()`, `triangles3d()` and `text3d()` functions.

The `plot3d()` method of the grids call for either the border plotting function `lines3d()` or the face plotting function `faces3d()`. In a workflow involving 3d plotting, these functions are used usually to create a compound plot representing different types of information. Experiment with these to optimize the 3d plotting experience.

The inner sphere is plotted by default, but can be turned off by setting the `sphere` argument of the `plot3d()` function to `FALSE`. The radius of the sphere can also be set using this argument. In case it is not set by the user, it defaults to the distance of the planar face center from the center of the grid.

The 3d plots so far showed only linear edges, but the plotting of arcs can be forced by setting the `arcs` argument to `TRUE`.

```
plot3d(tri, guides=F, arcs=T, sphere=6300)
```



### 2.3.2. Two-dimensional plots

The nature of the triangular/hexagonal grids is that they are intuitive in 3 dimensions, but behave cumbersome in 2d projections. Still, in any sort of printed or software publications, maps are the primary way to publish geographic data, which renders the projections very important. This part of the package is linked to the `sp` packages, which deal with the projection of data.

Each grid can be converted to either a `SpatialLines` or a `SpatialPolygons` object defined by the `sp` package. Two dimensional plotting can only happen if the 2d representation is calculated, which is (to save computation time) not automatic, but can be called for on demand.

```
hLow <- hexagrid(c(4,4), sp=TRUE)
# After this procedure finishes, a regular 2d plotting function can be invoked:
plot(hLow)
```
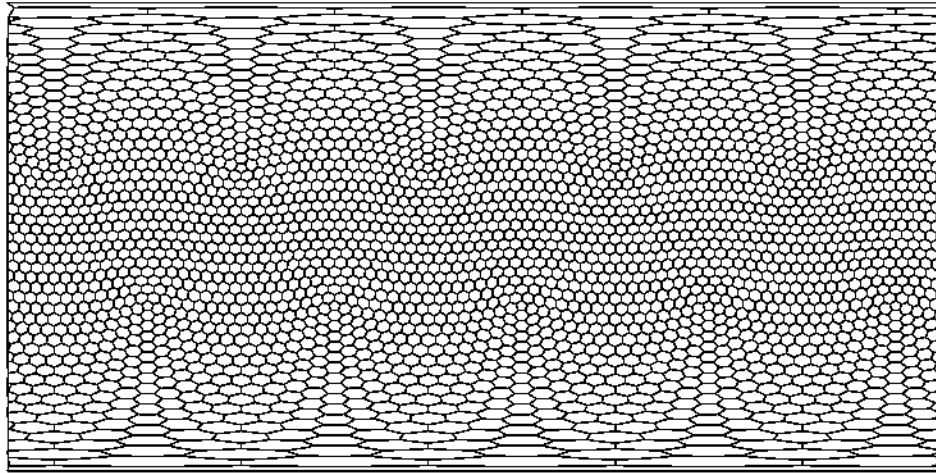

Here are some additional examples of projections using the World Borders Dataset (2) that can be accessed in the `SpatialPolygonsDataframe` format using the following chunk of code:

Here are some additional examples of projections using the 'z1' resolution of landy polygons from the OSM archive (2) that can be accessed in the `SpatialPolygons` format using the following chunk of code.

```
# use the rgdal package
library(rgdal)

# file path
file <- system.file("extdata", "land_polygons_z1.shx", package = "icosa")

# read in the shape file
wo <- readOGR(file, "land_polygons_z1")
```

A grid can be plotted easilly with this map, after their projection methods are adjusted:

```
# transform the land data to long-lat coordinates
wo <- spTransform(wo, gLow@proj4string)

#triangular grid
gLow<-newsp(gLow)

# # load in a map
```
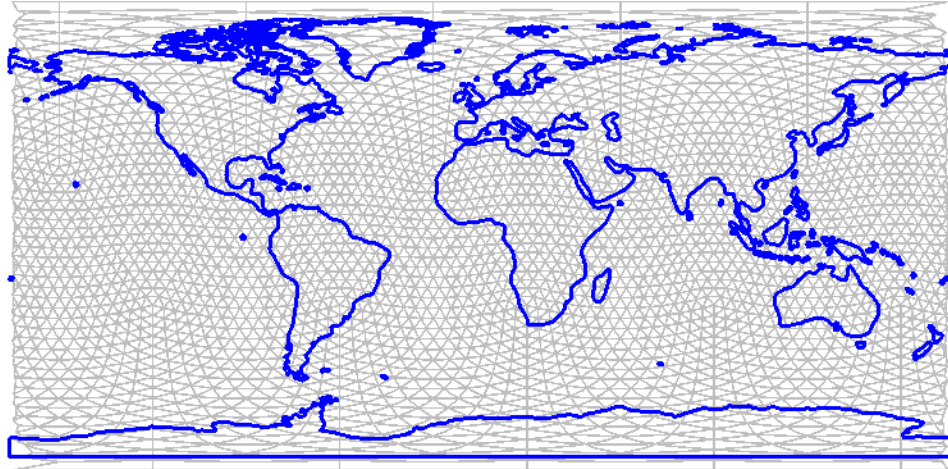
```
# plot the grid (default longitude/latitude)
plot(gLow, border="gray", lty=1)

# the reconstruction
lines(wo, lwd=2, col="blue")
```


The `gridlabs()` function can also be of use here to locate the vertices and faces of the plotted grid. The `type` argument is used to choose which part of the grid is to be shown. The rest of the argumnets are passed to the `text()` function.

```
# a very low resolution hexagrid
hVeryLow<-hexagrid(c(4))
# add 2d component
hVeryLow<-newsp(hVeryLow)
# the Robinson projection
robin <- CRS("+proj=robin")
# plot with labels
plot(hVeryLow, projargs=robin)
gridlabs(hVeryLow, type="f", cex=0.6,projargs=robin)
```
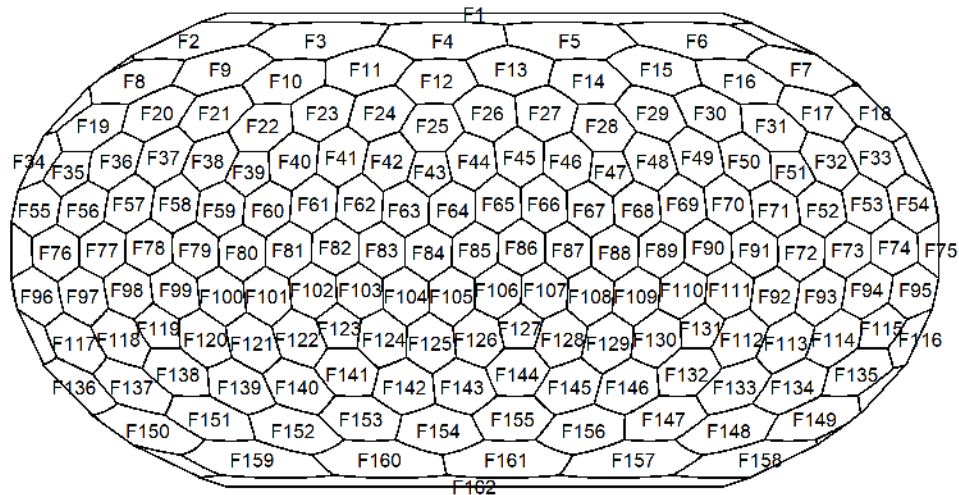
Similarly useful can be the `pos()` function, that retrieves the position of a named element in the grid, e.g. vertices and face centers:

```
pos(hLow, c("P2", "F12", NA))
```

```
##       long      lat
## P2     -54 87.86095
## F12    -18 82.07063
## <NA>    NA       NA
```

2.4. Layers

The grid itself operates as a scaffold for all kinds operations we can do based on data which can be organized in layers. At this moment, the layers are built on vectors, but in the next major update of the package they will incorporate both memory and harddrive-stored data similar to the `RasterLayer` class objects defined in the `raster` package.

Currently only the `facelayer` class is defined, which link individual values to the faces of a `trigrid` or `hexagrid` class object.

```
fl1<-facelayer(gLow) # the argument is the grid object to which the layer is linked
fl1
```

```
## class        : facelayer
## linked grid  : 'gLow' (name), trigrid (class), 4,4 (tessellation)
## dimensions   : 5120 (values) @ mean edge length: 481.07 km, 4.33 degrees
## values       : logical
## max value    : NA
```

```
## min value    :   NA
## missing      :   5120
```

```
str(fl1)
```

```
## Formal class 'facelayer' [package "icosa"] with 6 slots
##   ..@ grid       : chr "gLow"
##   ..@ tessellation: num [1:2] 4 4
##   ..@ gridclass  : chr "trigrid"
##   .. ..- attr(*, "package")= chr "icosa"
##   ..@ names      : chr [1:5120] "F1" "F2" "F3" "F4" ...
##   ..@ values     : logi [1:5120] NA NA NA NA NA NA ...
##   ..@ length     : int 5120
```

The `facelayer` has the same number of values as the the number of faces in the linked grid, accessed by the `length()` function

```
length(fl1)
```

```
## [1] 5120
```

The stored values can be assigned or shown by the values function:

```
values(fl1) <-1:length(fl1)
values(fl1)[1:10]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Besides storage and data manipulation, layers can be especially useful for plotting data. For logical data the `faces3d()` function will indicate which faces are occupied.

```
a <-facelayer(gLow)
values(a) <- sample(c(T,F), length(a), replace=T)
# plot the grid first
plot3d(gLow, guides=F)

# invoke lower level plotting for the facelayer
# (draws on previously plotted rgl environemnts)
faces3d(a, col="green")
```
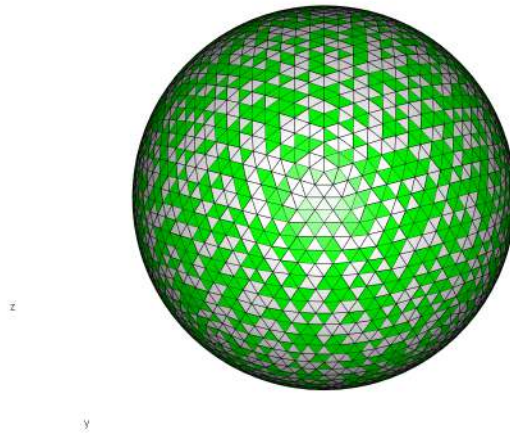
This is the lower level graphic function, that is called when the `plot3d()` method of the `facelayer` is called. For numeric data, heatmaps are built automatically based on the range of the data. Let's examine the basic case, where its number in sequence is assigned to every face.

The colors of the heatmaps can be changed by adding standard color names to the `col` argument:

```
# new layer
# grid frame
plot3d(gLow)
# the heatmap
faces3d(a, col=c("green", "brown"))
```
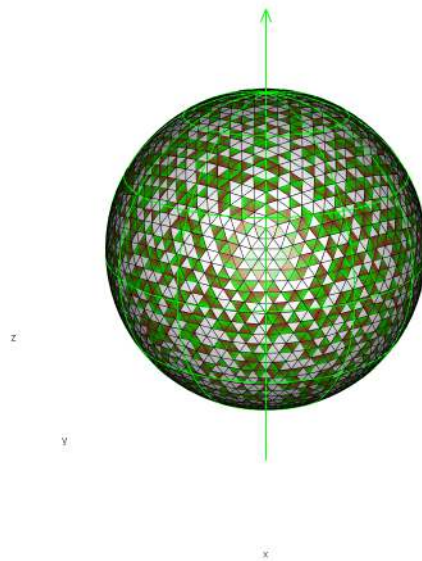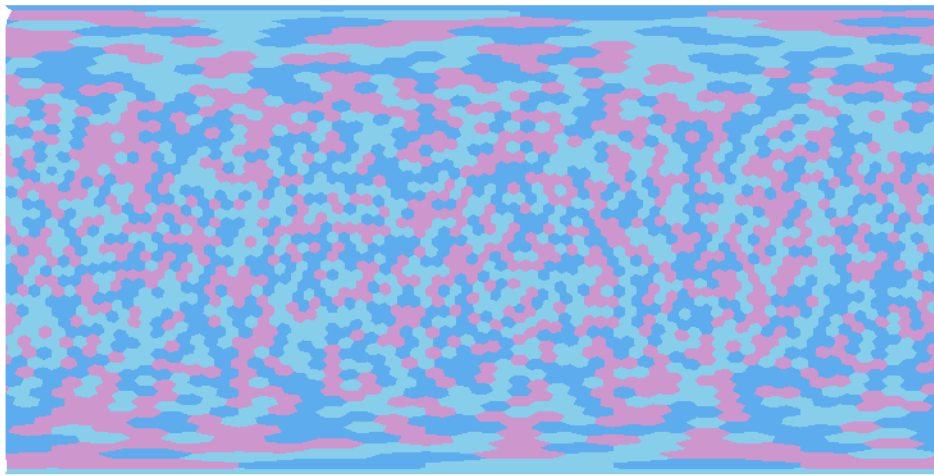
Categorical values can also be stored and plotted with the facelayer. By default, these values will be plotted with random colours, without a legend.

```
# new layer
catLayer<-facelayer(hLow)

# assign random information
catLayer@values<-sample(c("one","two","three"),length(catLayer), replace=T)

plot(catLayer)
```


3. Application

3.1. Lookup

Until this point only those features of the package were demonstrated that have no practicality on their own. All real world application of a gridding scheme relies on the capacity to look up coordinates and assign them to grid cells. The overall performance of the package boils down to the speed of this procedure. `icosa` uses a very efficient point-in-tetrhedron check to get the assigned cells to each set of coordinates. In the case of the `trigrid`, every face on the surface of the grid outlines a tetrahedron with the center of the object. At high resolutions this in itself can be very slow, especially if the number of queries is large, hence the necessity of the skeleton slot and the multiple levels of tessellations. With the `meanGC` tessellation method, the vertices of the input do not change, which means that every level of resolution can be retained when multiple rounds of tessellation happen. This allows the implementation of a hierarchical lookup algorithm, which searches the position of a point given by progressively refining the resolution, so an exhaustive lookup is not required.

3.1.1. The 'locate()' function - point query

The most straightforward implementation is the `locate()` function which is used to find the position of a set of points on the grid:

```
# generate 5000 random coordinates on a sphere of default radius
pointdat <- rpsphere(5000)

# and locate them on the grid 'gLow'
cells<-locate(gLow, pointdat)

# the return of this function is vector of cell names
head(cells)
```

```
## [1] "F1203" "F321"  "F4098" "F4711" "F129"  "F4158"
```

The function accepts matrices in longitude-latitude, and XYZ format as well. An object of the `SpatialPoints` class defined in the package `sp` can alse be provided as input. In the case of the polar coordinate entry, the coordinates will be transformed to the xyz Cartesian coordinate system using the default radius. This function returns the names of the faces that the points fell on. In the case of points that fall on vertices or edges (which is extremely unlikely with real world data), the returned values are by default NAs. The `locate()` function is especially powerful if it combined with the `table()` and `tapply()` functions or similar types of iterators:

```
tCell <- table(cells)
fl <- facelayer(gLow,0)
# [] invokes a method that save the values to places that
# correspond to the names attribute of tCell
fl[] <-tCell #
# heat map of the point densities
plot3d(fl)
```
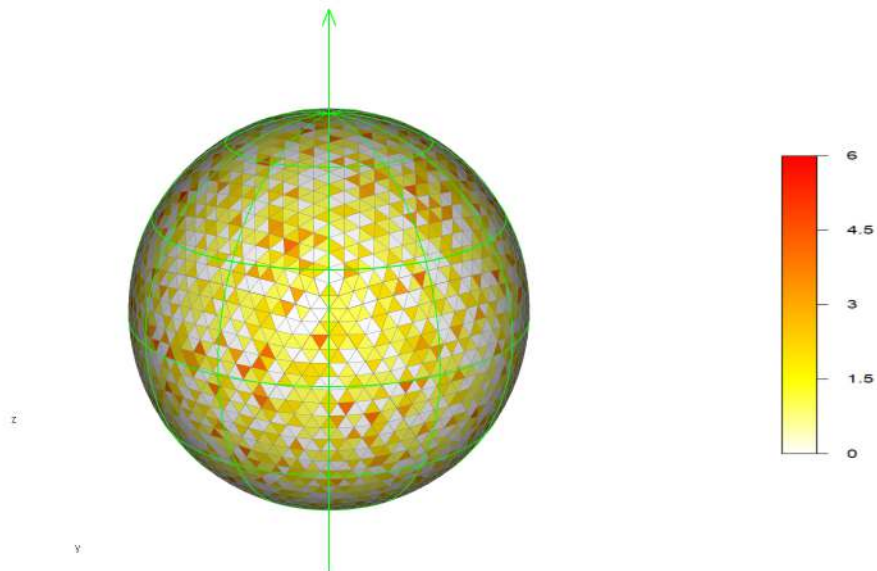

This function operates just as fine with the `hexagrid` object, and uses subfaces to locate the points. Every hexagonal face consists of 6 subfaces and every pentagonal face contains 5 subfaces.

The performance of the `locate()` function is linearly related to the number of queries. It is also positively related to the grid resolution, although larger tessellation values will increase computation time more than using multiple levels of tessellations.
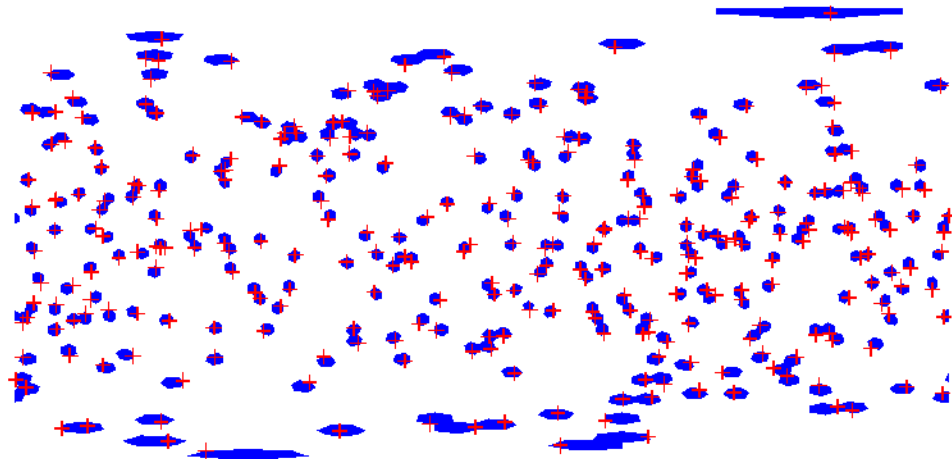
3.1.2. The occupied() function

For presence-absence values the function `occupied()` can be used. It returns a `facelayer` class objecte with logical values (`TRUE` when the face is occupied an `FALSE` when the face is not).

The example below shows how the occupied cells can be shown with the points:

```
# run function only on the first 300
fl<-occupied(hLow, pointdat[1:300,])

# the plot function can also be applied to the facelayer object
plot(fl, col="blue")

# show the points as well
points(CarToPol(pointdat[1:300,]), col="red", pch=3, cex=0.7)
```
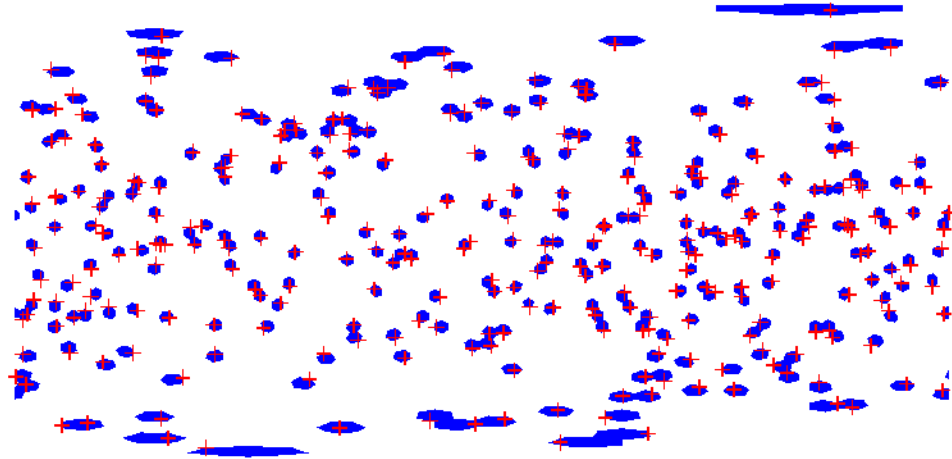

Naturally the grid can be shown as well, for instance with `lines()`:

```
# the plot function can also be applied to the facelayer object
plot(fl, col="blue")

points(CarToPol(pointdat[1:300,]), col="red", pch=3, cex=0.7)
lines(hLow, col="gray")
```

The `occupied()` function also applies to various other object types and behaves as a wrapper function around methods that return which faces are occupied by the input objects. Most notable among these is the `SpatialPolygons`, `SpatialLines`, and `SpatialPoints` classes defined by the package `sp`. The method changes the coordinate reference system (CRS) of the input object is used to transform it to the spherical model first, and then the function transforms the coordinates to XYZ Cartesian space.

3.1.3. Handling raster-type data

Most global data compilations use raster formats to store information. These data can be fitted to the icosahedral grids using the `resample()` function. The arguments of this function depend on the nature and interpretation of the data points. As resampling requires some form of interpolation, it needs assumptions on the representativity of the measurements. Each original data point can be thought of either as an entity that represent the entire cell or only the center of the cell. In the first case the original raster object needs to be upscaled with the nearest neighbour method, and in the latter, another form of interpolation is necessary (e.g. the bilinear or bicubic resampling). The 'method' argument of this function is passed to the `resample()` function in the `raster` package, and is used to generate higher resolution data from the original raster.The `resample()` function can also be used to upscale, or downscale a `facelayer` linked to `trigrid` or `hexagrid` object as well.

3.2. Surface-graph representation

The grid structure is a compound object, can also be understood as a graph of connected faces. This representation is efficiently implemented using the `igraph` package. On default, `igraph` represenation of the grid is added to the `graph` slot of the grid object. In this graph, each face is connected to its direct neighbours, which allows etheir efficient lookup, the implementation of shortest path algorithms and more.

3.2.1. Neighbours

The most direct application of this representation is the `vicinity()` function that allows the user to look up cells that are closest to a focal cell, without calculating distance matrices. This particular example gets all the neighbouring cells of the `F125` cell.

```r
# calculate a very coarse resolution grid
gVeryLow<-trigrid(8, sp=T)
# names of faces that are neighbours to face F125
facenames<-vicinity(gVeryLow, "F125")
# plot a portion of the grid
plot(gVeryLow, xlim=c(0,180), ylim=c(0,90))
# plot the original and the neighbouring faces
plot(gVeryLow@sp[facenames], col="red", add=T)
# the names of all the cells
gridlabs(gVeryLow, type="f", cex=0.5)
```
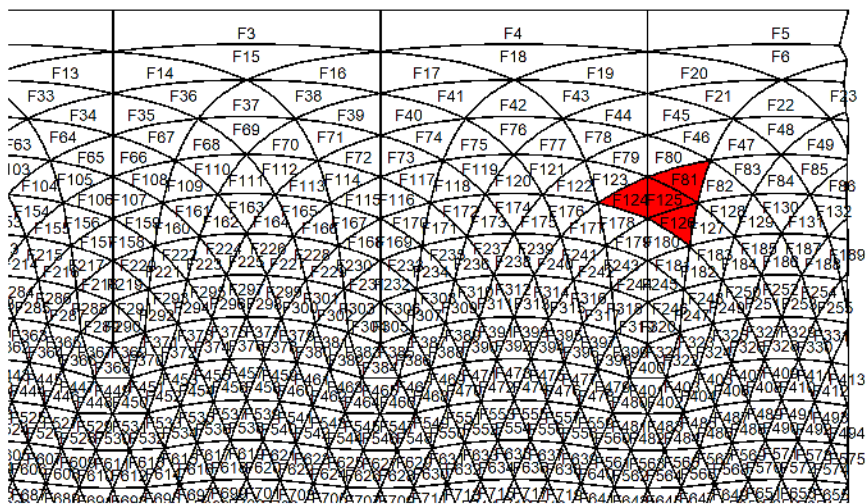

3.2.2. Using 'igraph' in geographic calculations

Using a separate `igraph` class object can be especially useful when subsets of the grids are to be used for an analysis or simulation.

```r
# attach igraph
library(igraph)
```

```
##
## Attaching package: 'igraph'

## The following objects are masked from 'package:icosa':
```

18

```
##
##     edges, vertices
## The following object is masked from 'package:raster':
##
##     union
## The following objects are masked from 'package:stats':
##
##     decompose, spectrum
## The following object is masked from 'package:base':
##
##     union
```

Please note that `igraph` masks out some of the auxilliary functions written in this package as well. Naturally, you can use the `induced_subgraph()` function of the `igraph` package directly on the grid representation of the grid.

```
faces<-paste("F", 1:10, sep="")
subGraph <- induced_subgraph(gVeryLow@graph,faces)
plot(subGraph)
```
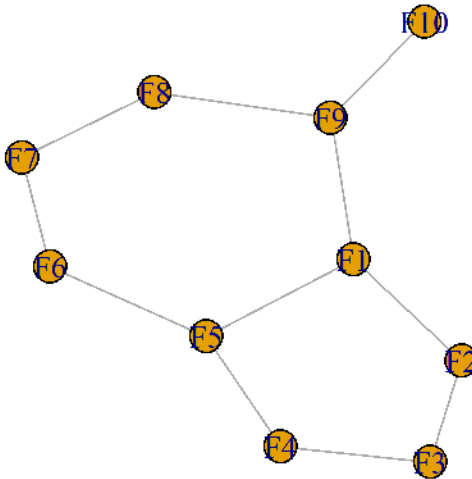

The subsetting of the grid will also subset the `igraph` class representation:

```
lowGraph<-gLow[1:12]@graph
```

or you can create it from a logical `facelayer`, for example from the occupied cells of the land data we imported earlier:

```
# look up the polygons
landFaces<-occupied(hLow, wo)

# create a new grid from a facelayer
landGraph<-gridgraph(landFaces)
plot(landFaces, col="brown")
```


This particular graph is a rough estimate for the presence of terrestrial settings, and can be useful for path calculations.

```
# shortest path in igraph
path <- shortest_paths(landGraph, from="F432", to="F1073", output="vpath")
# the names of the cells in order
cells<-path$vpath[[1]]$name
# plot the map
plot(landFaces, col="brown", xlim=c(0,90), ylim=c(0,90))
# make a subset of the grid - which corresponds to the path
routeGrid<-hLow[cells]
# plot the path
plot(routeGrid, col="red", add=T)
```
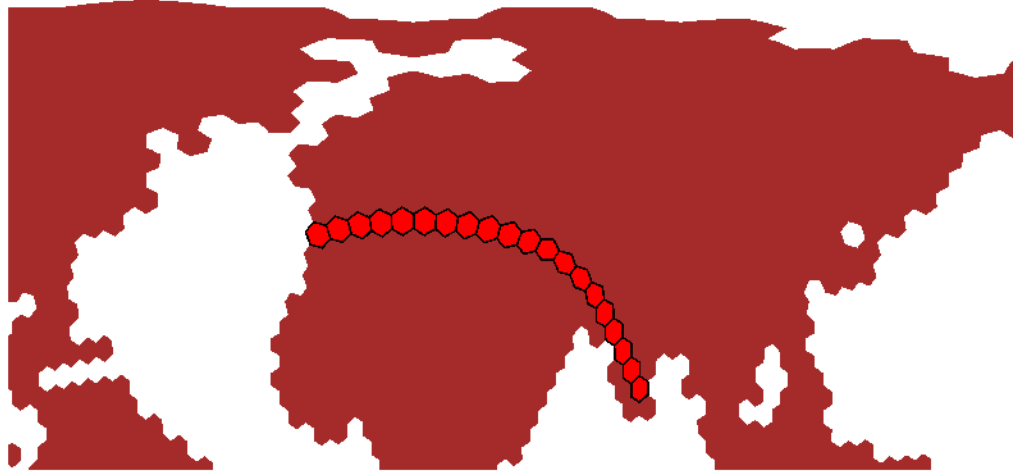
The shortest path using grid cells is a suboptimal estimate of the actual shortest route between two points, as the graph structure limits the angles the path can turn to. A future update will include a function that allows more accurate estimates of the actual shortest paths.

Random walk simulations can also be built using the graph represetation. In this example a random walker will walk 100 steps on the grid, starting from face `F432`.

```r
# plot the map
plot(landFaces, col="brown", xlim=c(0,90), ylim=c(0,90))
# create a random walk from source cell with a given no. of steps
randomWalk <- random_walk(landGraph, steps=100, start="F432")
# the names of the cells visited by the random walker
cells<-randomWalk$name
# the source cell
plot(hLow["F432"], col="green",add=T)
# the centers of these faces
centers<-CarToPol(hLow@faceCenters[cells,], norad=T)
# draw the lines of the random walk
for(i in 2:nrow(centers)){
    segments(x0=centers[i-1,1], y0=centers[i-1,2], x1=centers[i,1], y1=centers[i,2], lwd=2)
}
```
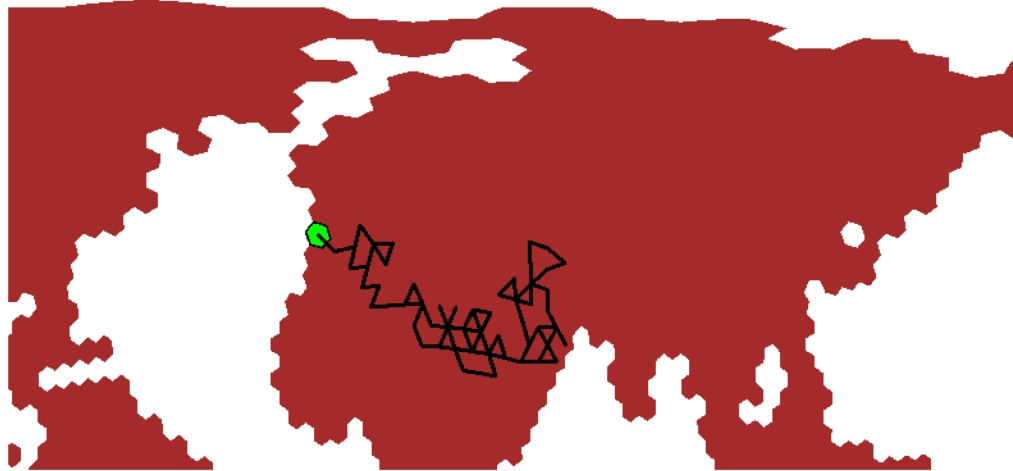
Acknowledgements

References

(1) Moritz, H. 2000. Geodetic Reference System 1980. Journal of Geodesy, 74, 128-162.

(2) http://openstreetmapdata.com/

(3) http://www.worldclim.org/