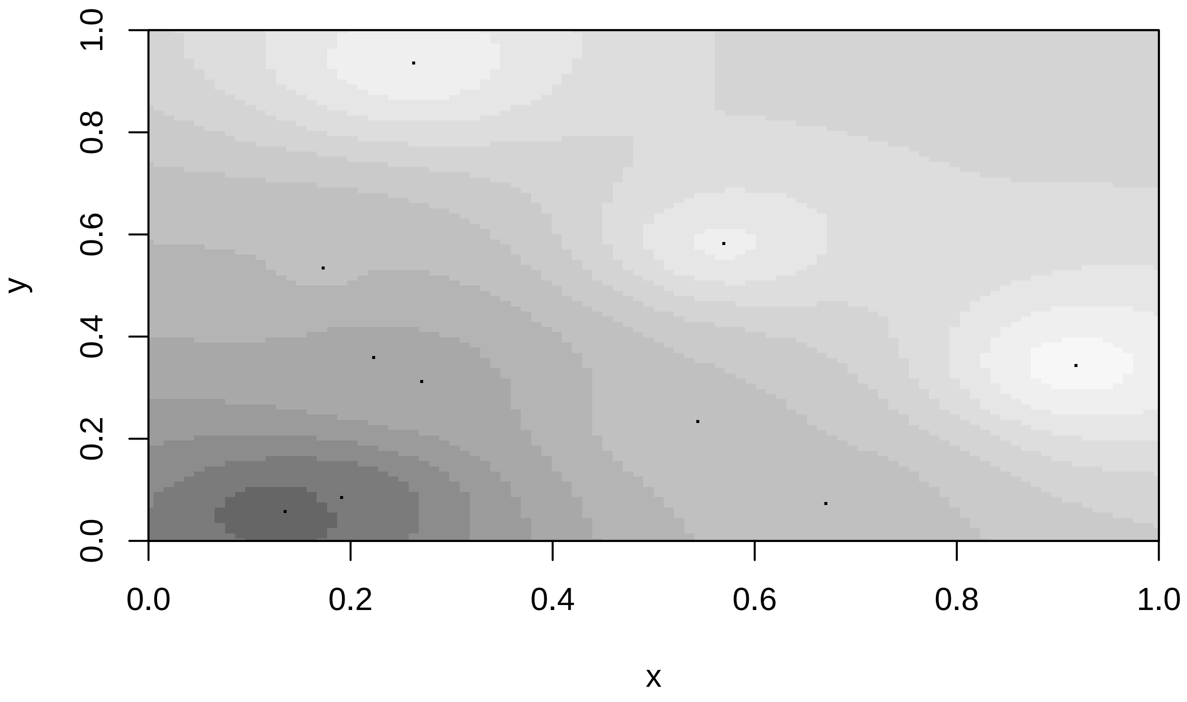
Using SimpleGridder: Gridding Environmental Data

Thomas Bryce Kelly

2024-10-15

Let's start with a working example to see how SimpleGridder works, what it does, and how the results look. If you don't have SimpleGridder installed already, it can be installed directly from Github using *devtools*: `devtools::install_github('tbrycekelly/SimpleGridder')`.

```r
library(SimpleGridder)

## Make test data (x, y, z)
n = 10
x = runif(n)
y = runif(n)
z = x + y

## Use SimpleGridder to make grid
grid = buildGrid(xlim = c(0,1), ylim = c(0,1), nx = 100, ny = 100)
grid = setGridder(grid, neighborhood = 20)
grid = appendData(grid, x, y, z, 'salinity')
grid = interpData(grid)
plotGrid(grid, 'salinity')
```



Great, now we have a template to refer to as we (1) see what each step does, (2) what options we have (or don't have), and (3) review the applications for this package.

To generate an interpolated data product (i.e. gridding) there are X distinct steps, each of which you control in SimpleGridder:

1. Construct an idealized grid with the locations you want to interpolate (and extrapolate) your data to.
2. Decide on an interpolation algorithm, for example nearest neighbor.
3. Prepare observations against which the interpolator will be applied.
4. Finally, generate your interpolated data product.

So why so many steps?

While conceptually simple, gridding observations entails a lot of assumptions, decisions, and complexity that you were likely (and perhaps luckily) unaware of until now. Let's take a look at each of these functions (and steps) in turn.

Starting wit hthe `buildGrid()` function, let's see what the grid data structure (a list) looks like.

```
grid = buildGrid(xlim = c(0,1), ylim = c(0,1), nx = 100, ny = 100)
str(grid)
#> List of 6
#>  $ x     : num [1:100] 0 0.0101 0.0202 0.0303 0.0404 ...
#>  $ y     : num [1:100] 0 0.0101 0.0202 0.0303 0.0404 ...
#>  $ data  : list()
#>  $ interp: list()
#>  $ grid  :List of 2
#>   ..$ x: num [1:100, 1:100] 0 0.0101 0.0202 0.0303 0.0404 ...
#>   ..$ y: num [1:100, 1:100] 0 0 0 0 0 0 0 0 0 0 ...
#>  $ meta  :List of 8
#>   ..$ xlim    : num [1:2] 0 1
#>   ..$ ylim    : num [1:2] 0 1
#>   ..$ x.scale : num 0.0101
#>   ..$ y.scale : num 0.0101
#>   ..$ nx      : int 100
#>   ..$ ny      : int 100
#>   ..$ x.factor: num 1
#>   ..$ y.factor: num 1
```

Overall, most of this should both make conceptual sense and be unnecessary for you to ever interact with directly–but knowledge is power. A *grid* object is a list made up of 6 components: the **x** and **y** locations of the grid, **data** which will be used to store the observations that you want to grid, **interp** that will hold the interpolated data product you generate, **grid** that unsurprisingly are the matrix representations (i.e. 2D) of the (x,y) location pairs for the grid, and **meta** that holds all the metadata about the grid itself.

Again, hopefully this is all detail that you never interact directly with (but you can, especially if you want to customize anything!).

buildGrid

The arguments for the `buildGrid()` function are as follows:

```
buildGrid(xlim, ylim, nx, ny, x.scale, y.scale, x.factor, y.factor)
```

Both **xlim** and **ylim** control the spatial extent of the grid product, which for our examples were (0,1) for both the x and y axes. Then to set resolution you have the choice of either setting the number of grid cells (**nx** and **ny**) or by setting the grid size you want (**x.scale** and **y.sacle**). For either axis you can use either one, but if you try to use both methods the bad things might happen…

Finally there are the **x.factor** and **y.factor** arguments. These are a bit mysterious at first but are essentially setting the scale or units for the x and y axis, respectively. By default, both are set to 1 meaning that a change of 1 x unit is equivalent to a change of 1 y unit. The reason we need to set this is that most interpolation algorithms need to decide how far away an observation is from a given point (e.g. nearest neighbor). In most cases you don't want the end result to depend on whether you used feet or meter.

This also sets the relative length scales used. For example, if I want to estimate the temperature of the water at a given point based on observations either 10 feet from me horizontally or 10 feet below me, we would usually assume that the temperature besides me is a better guess than below me (cold water sinks). This allows us to set this relationship. Unfortunately this is one of the biggest sources of issues for qualitatively "bad" interpolations and there's not a great one-size-fits-all approach to set these values. This will likely be a topic of an entire conversation someother time.

setGridder

The `setGridder()` function sets the interpolation algorithm that you want to use with your data. This is set once and is applied to all data within a grid.

```
setGridder(grid,
           gridder = gridWeighted,
           weight.func = function(x) {1 / x^2},
           neighborhood = 20)
```

Arguments to this function include the *grid* itself, the interpolation algorithm **gridder** which defaults to `gridWeighter`, a weighting function (**weight.func**) that defaults to the inverse of the distance squared, and a local neighborhood that limits how many points are considered in the interpolation.

appendData

This simple function just stores a copy of the variables provided in the *data* list of the grid. This ensure that any preprocessing can be done if required (doesn't do anything like that currently) and offers the potential of re-interpolating data sets or cross-validating results later on.

```
grid = appendData(grid, x, y, z, label)
```

Everything in this function call should be self explanatory except for the **label** argument. Use **label** to set the name of the data set that you are appending. This should be a valid R name (e.g. starting with a number is invalid), so just don't be too wild with it.

interpData

This is the final step to generate the interpolated data product and is hopefully the lest eventful. This function accepts no arguments except for the grid object created earlier. During this function call all the calculations are performed and the resulting interpolated data set is actually created. The resulting data set is placed into *interp* and is given the same label as provided to the original data above.

```
grid = interpData(grid)
```

plotGrid

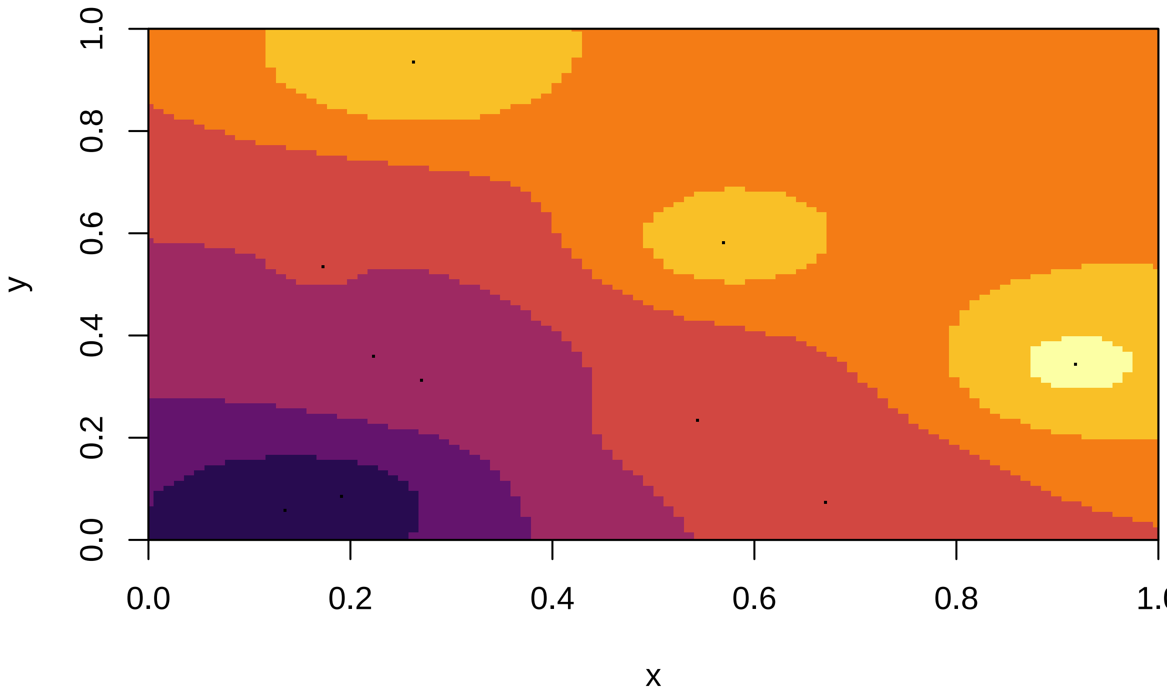
While I said the last step was the final step, it would not be very fun to leave it there. The visualization tools are currently limited to a simple plot of the interpolated data. Eventually there will be additional functions for special applications and diagnostics.

```
plotGrid(section, label, xlim, ylim, xlab, ylab, zlim, pal, ...)
```

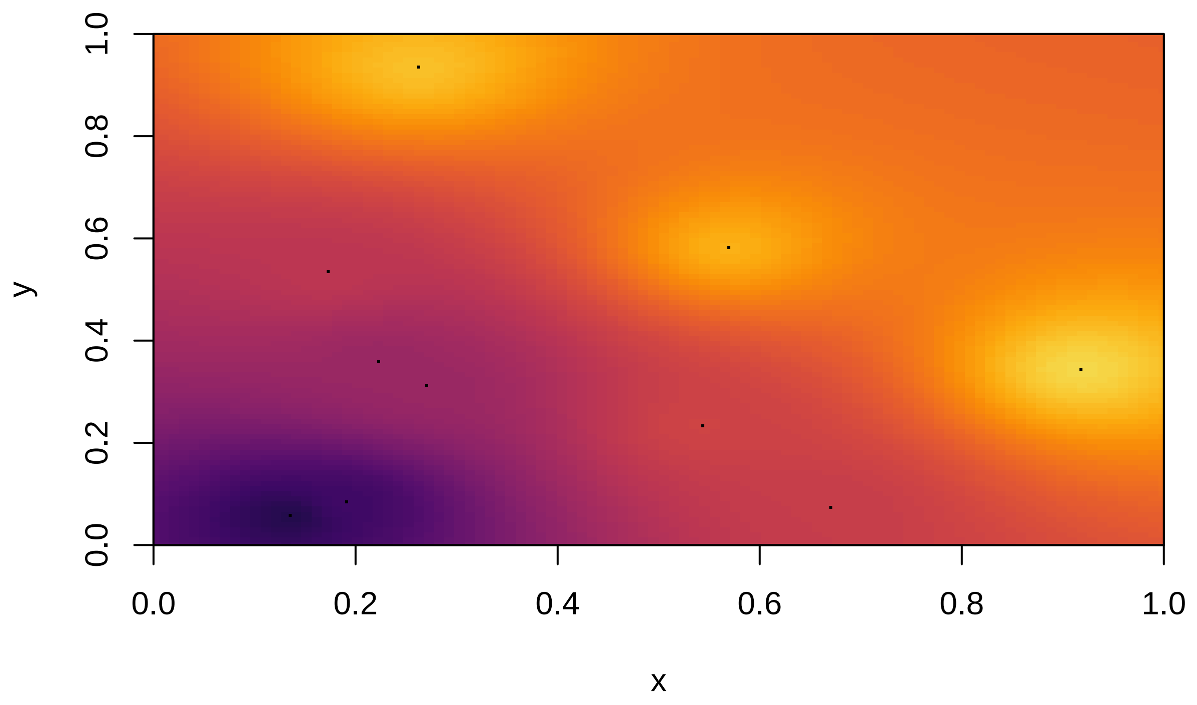
The `plotGrid()` function accepts a grid product as input along with the data set label (as above) to indicate which interpolated values to plot. The usual formatting options are available with **xlab, ylab, xlim,** and **ylim**. Additionally the colorscale range can be set with **zlim** and the color palette provided to **pal** as a set of colors. The default palette is `grayscale(16)`, or a set of 16 grayscale values from black to white. I recommend using the many excellent palatte functions provided in *pals*: e.g. `pals::inferno(8)` or `pals::ocean.haline(32)`. Note that the number provided here is arbitrary and just sets the number of distinct colors available.

Here are a couple of example plots using the same grid as the first example:

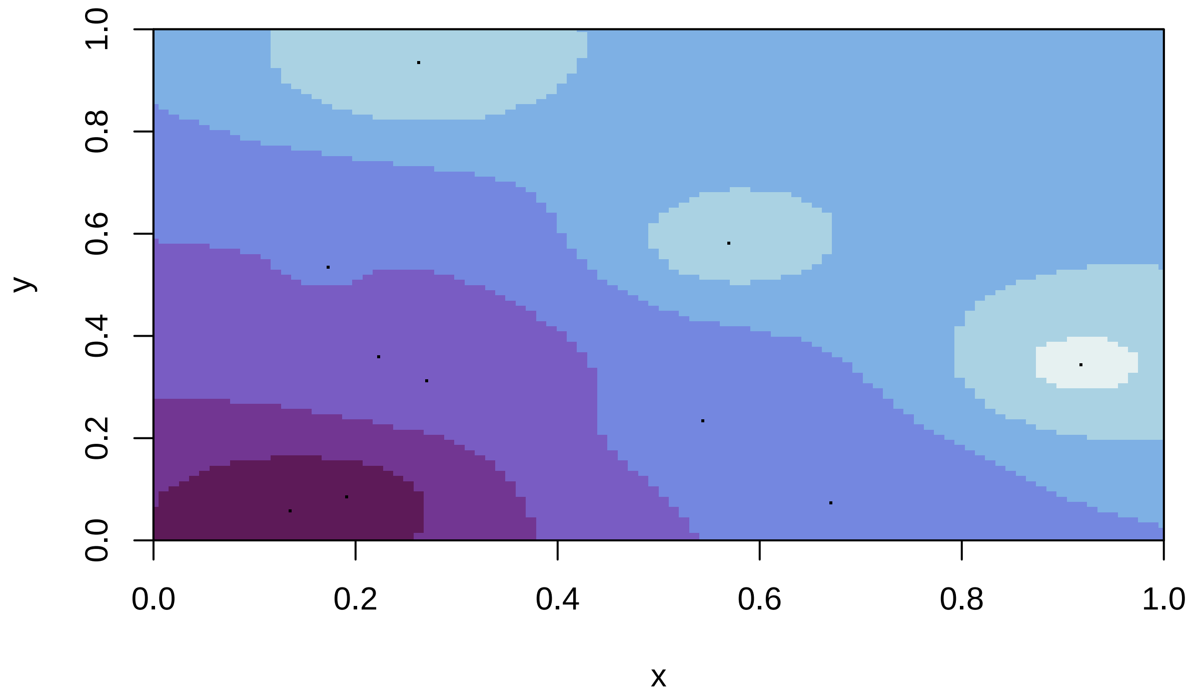
```
grid = buildGrid(xlim = c(0,1), ylim = c(0,1), nx = 100, ny = 100)
grid = setGridder(grid, neighborhood = 20)
grid = appendData(grid, x, y, z, 'salinity')
grid = interpData(grid)

#par(mfrow=c(2,2))
plotGrid(grid, 'salinity', pal = pals::inferno(8))
```



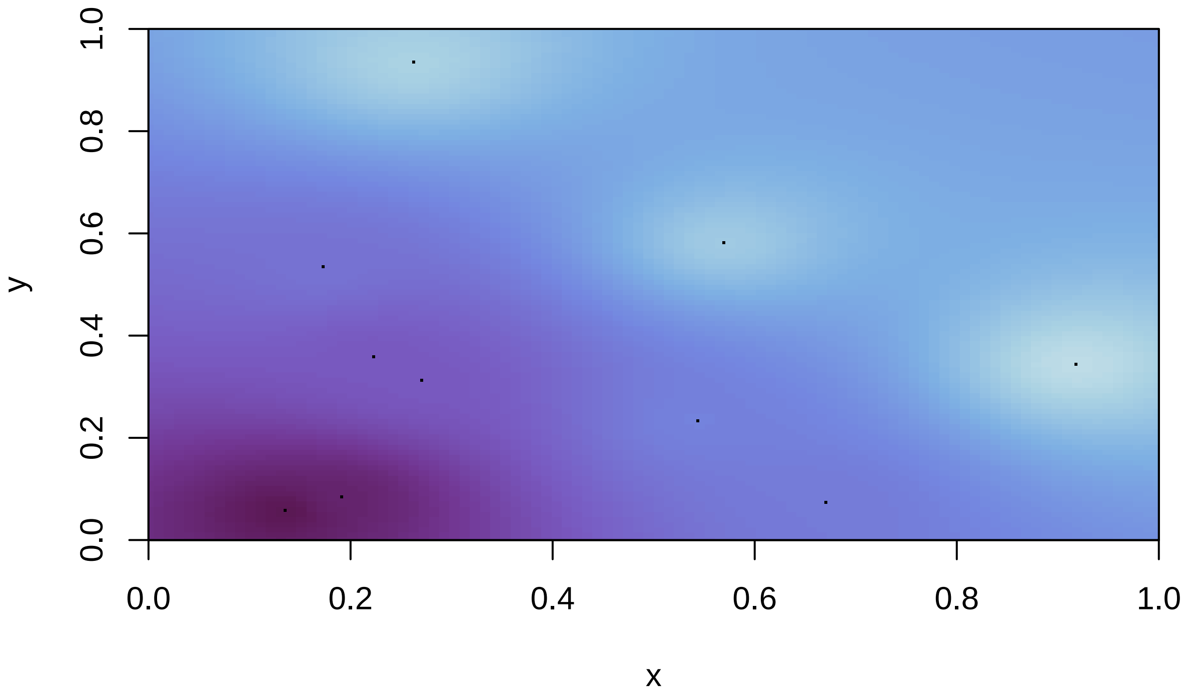
```
plotGrid(grid, 'salinity', pal = pals::inferno(128))
```



```r
plotGrid(grid, 'salinity', pal = pals::ocean.dense(8))
```

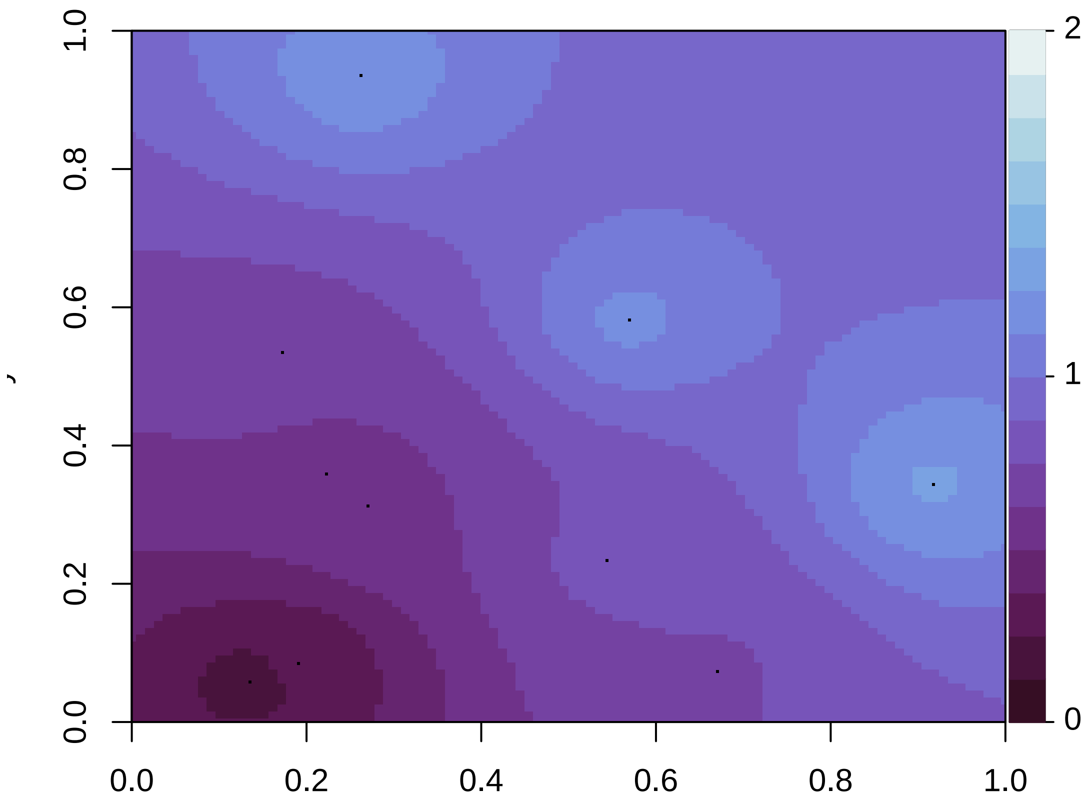


```r
plotGrid(grid, 'salinity', pal = pals::ocean.dense(128))
```



Finally, *SimpleGridder* contains a basic colorbar function:

```
par(plt = c(0.1, 0.8, 0.1, 0.9)) # Changing plot aspect ratio to make room for colorbar
plotGrid(grid, 'salinity', pal = pals::ocean.dense(16), zlim = c(0,2))
colorbar(pal = pals::ocean.dense(16), zlim = c(0, 2))
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2
1
0