Plotting with SpaDES

Eliot McIntire

Friday, September 12, 2014

Contents

1 Plotting in SpaDES

One of the major features of the SpaDES package is that can take advantage of the numerous visualization tools available natively or through user built packages (e.g., RgoogleVis, ggplot2, rgl). The main plotting function, Plot (i.e., with a capital P), is built using the grid package. We have specifically built a plotting system that allows for relatively fast plotting of rasters and points with the ability to make multi-frame plots without the module (or user) knowing which plots are already plotted. In other words, the main plotting function can handle SpaDES modules, each of which can add plots, without each knowing what the current state of the active plotting device is. This means that the plotting can be treated as modular. Importantly, conventional R plotting still works fine, so you can use the features provided in this package or you can use base plotting functions without having to relearn a completely new set of plotting commands.

To demonstrate plotting, we first load some maps. These maps are randomly generated maps that come with the SpaDES package. In the code snippet below, we create the list of files to load, which is every file in the "maps" subdirectory of the package. Then we load that list of files. Because we specified .stackName in the fileList, the loadFiles function will automatically put the individual layers into a RasterStack; the individual layers will, therefore, not be available as individual objects within the R environment. If .stackNames did not exist, then the individual files would be individual objects.

```
#  Make list of maps from package database to load, and what functions to use to load them
library(SpaDES)
fileList <-
    data.frame(files =
     dir(file.path(
                   find.package("SpaDES",
                                lib.loc=getOption("devtools.path"),
                                quiet=FALSE),
                  "maps"),
        full.names=TRUE, pattern= "tif"),
     functions="rasterToMemory",
     .stackName="landscape",
     packages="SpaDES",
     stringsAsFactors=FALSE)

# Load files to memory (using rasterToMemory) and stack them (because .stackName is provided above)
loadFiles(fileList=fileList)
```

```
## Warning: Global parameters .stackName are not used in any module.
```

```
# extract a single one of these rasters
DEM <- landscape$DEM
#'
```

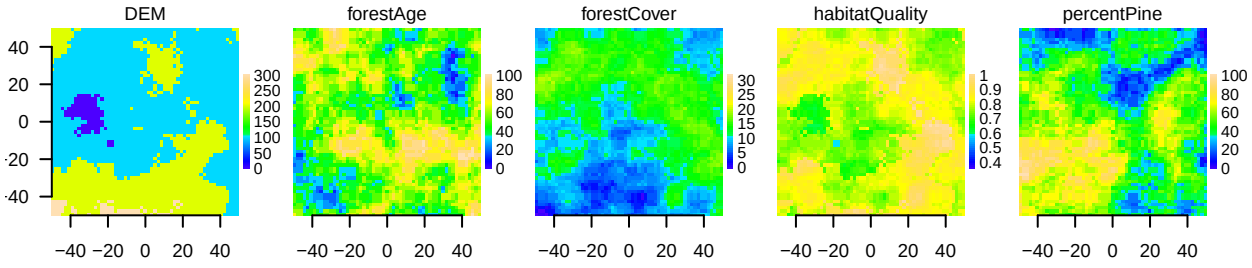
2 The `Plot` function

There are several features of `Plot` that are worth highlighting.

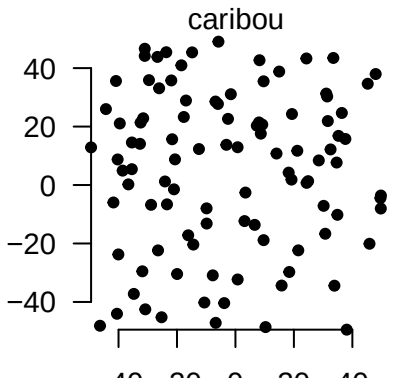
2.1 Layer types

Its main purpose is to plot spatial type objects. Specifically, it currently can plot RasterLayers, RasterStacks and SpatialPoints* objects.

```
Plot(landscape, add=FALSE)
```



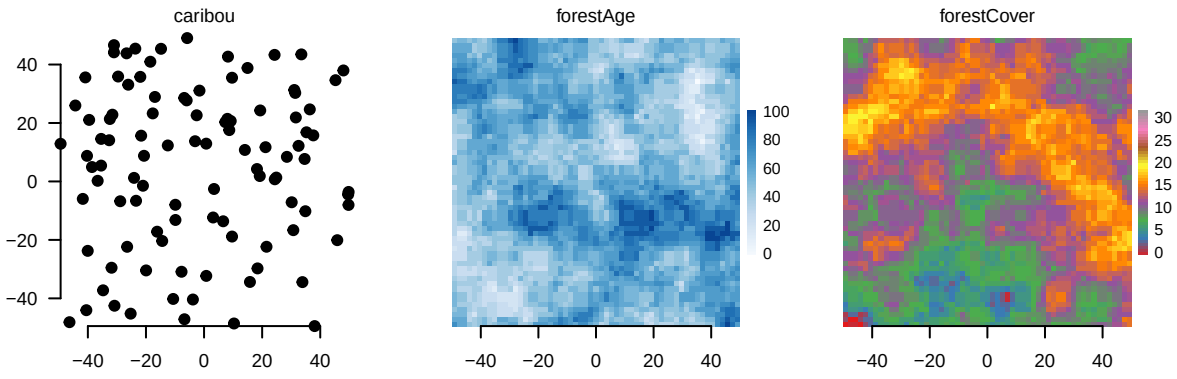
```
# make a SpatialPointsNamed object
caribou <- SpatialPointsNamed(coords=cbind(x=runif(1e2,-50,50),y=runif(1e2,-50,50)),
                              name="caribou")
Plot(caribou, add=FALSE)
```



2.2 Colors

We likely won't want the default colors for every map. Currently, the best way to change the color of a map is to give it a colortable using the `setColors` function in `SpaDES`. Every RasterLayer can have a colortable, which gives the mapping of raster values to colors. If not already set in the file (many .tif files and other formats already have their colortable set), we can use setColors(Raster*) with a named list of hex colours, if a RasterStack, or just a vector of hex colors if only a single RasterLayer. These can be easily built with the RColorBrewer package, with the function brewer.pal(). But there are many other ways in R, see `colorRampPalette`.

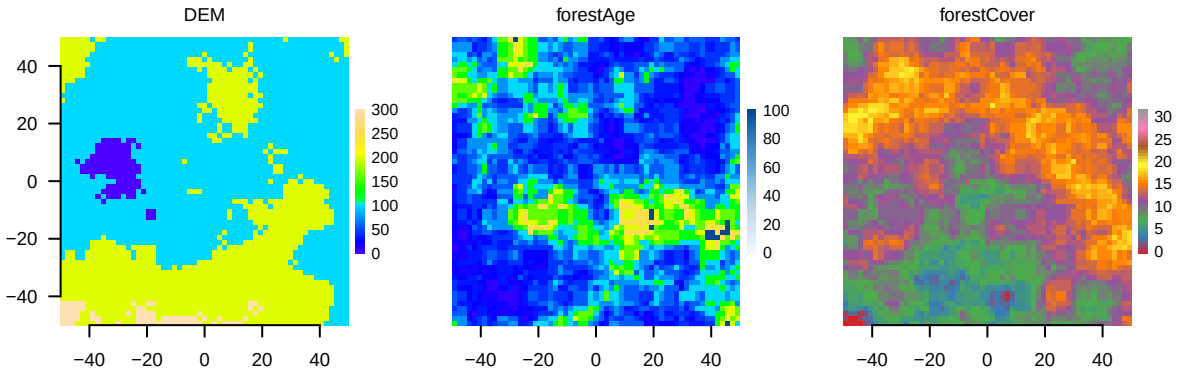
```
# can change color palette
library(RColorBrewer)
setColors(landscape, n=50) <-
              list(DEM=topo.colors(50),
                   forestCover=brewer.pal(9, "Set1"),
                   forestAge=brewer.pal("Blues", n=8),
                   habitatQuality=brewer.pal(9, "Spectral"),
                   percentPine=brewer.pal("GnBu", n=8))
Plot(landscape[[2:3]])
```



2.3 Names

It is critical in SpaDES plotting that every layer has a unique name. RasterLayers already have this functionality, contained within the element, `names`. RasterStacks do not, nor do SpatialPoints* objects. Names can be added to RasterLayers using `names` and to RasterStacks or SpatialPoints using the assignment functions `name` (in the form `names(Layer)<-"something"` or `name(Layer)<-"something"`). This would be necessary when a new Raster is created (say in a simulation) or if a new Raster is derived from another Raster, as the new one would inherit the original name. The new layer would then overplot the original layer, which is not the desired behavior.

```
#Make a new raster derived from a previous one; must give it a unique name
habitatQuality2 <- ((landscape$forestAge) / 100 + 1) ^ 6
#setColors(habitatQuality2) <- heat.colors(50)
Plot(landscape[[1:3]], add=FALSE)
Plot(habitatQuality2, add=TRUE)
```

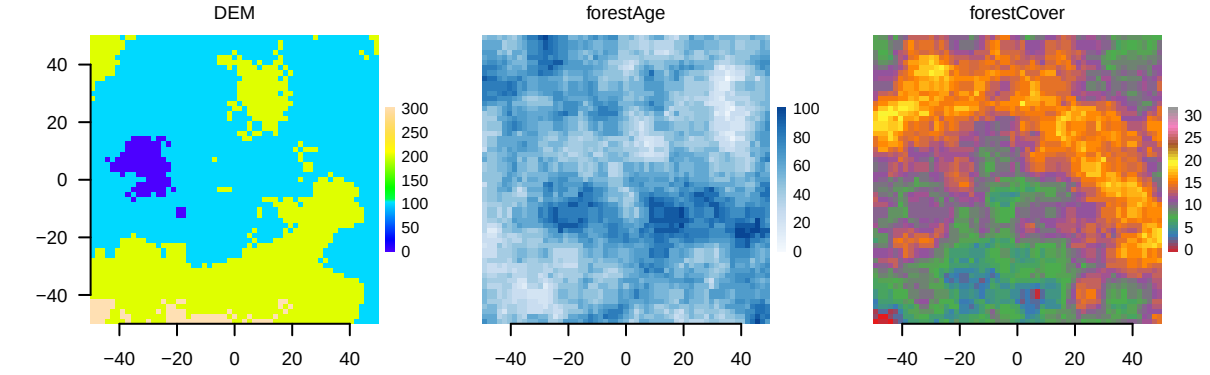


To get the correct behavior, give the new layer a unique name:

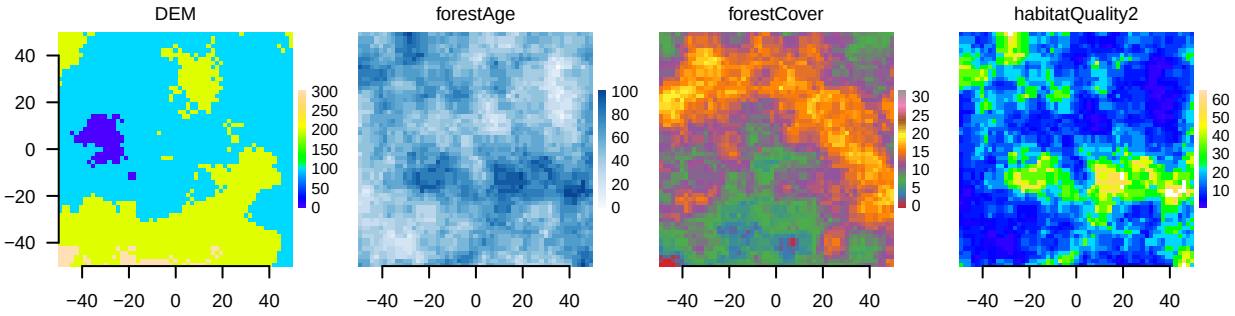
```
name(habitatQuality2) <- "habitatQuality2"
print(habitatQuality2)
```

```
## class       : RasterLayerNamed
## dimensions  : 100, 100, 10000, 1  (nrow, ncol, ncell, nlayers)
## resolution  : 1, 1  (x, y)
## extent      : -50, 50, -50, 50  (xmin, xmax, ymin, ymax)
## name        : habitatQuality2
## coord. ref. : NA
## names       : habitatQuality2
## min values  :               1
## max values  :              64
```

```
Plot(landscape[[1:3]], add=FALSE)
```



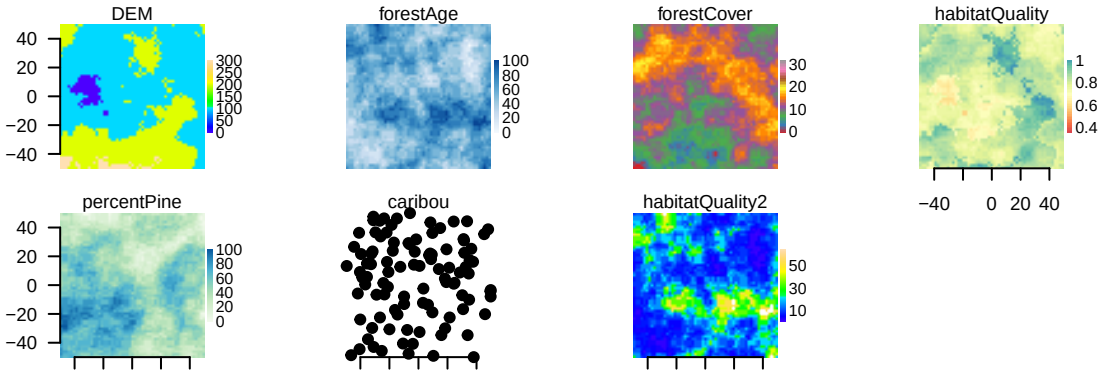

```
Plot(habitatQuality2, add=TRUE)
```



2.4 Mixing Layer Types

Any combination of RasterStacks, RasterLayers, and SpatialPoints* objects can be plotted.

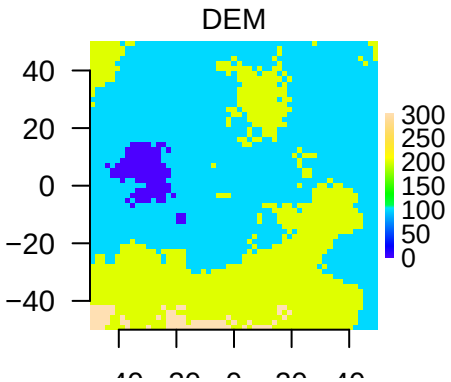
```
Plot(landscape, caribou, habitatQuality2, add=FALSE)
```



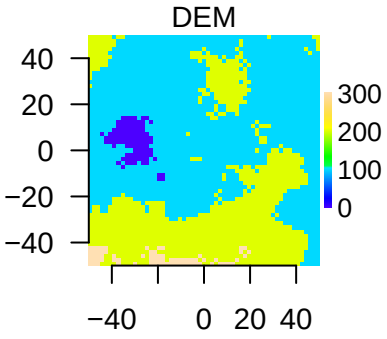
2.5 visualSqueeze

Under most circumstances, the plotting regions will be automatically scaled to maximize the area taken by the map layers, minimizing white space, but allowing axes, legends and titles to be visible when they are plotted. In some devices, this automatic scaling is imperfect, so axes or legends may be squished. The `visualSqueeze` argument is an easy way to shrink or grow the plots on the device. The default value is 0.75 representing ~75% of the area. If the plots need to be slightly smaller, this could be set to 0.6; if they can be larger, visualSqueeze could be set to 0.8.

```
# x axis gets cut off in pdf and html
Plot(DEM, add=FALSE)
```




```
Plot(DEM, visualSqueeze=0.6, add=FALSE)
```



A key reason why the legends or axes are cut off sometimes is because there is a minimum threshold for font size for readability. So, either visualSqueeze can be set or making a larger device will usually also solve these problems.

3 Modularity

One of the main purposes of the `Plot` function is modularity. The goal is to enable any `SpaDES` module to be able to add a plot to the plotting device, without being aware of what is already in the plotting device. To do this, there is a hidden global variable (a `.spadesArrN` [where N is the device number] object of S4 class, "arrangement") created when a first Plot function is called. This object keeps the layer names, their extents, and whether they were in a RasterStack (and a few other things). So, when a new `Plot` is called, and `add` is used, then it will simply add the new layer. There may not be space on the plot device for this, in which

case, everything will be replotted in a new arrangement, but taking the original R objects. This is different than the grid package engine for replotting. That engine was not designed for large numbers of plots to be added to a region; it slows down immensely as the number of plots increases.

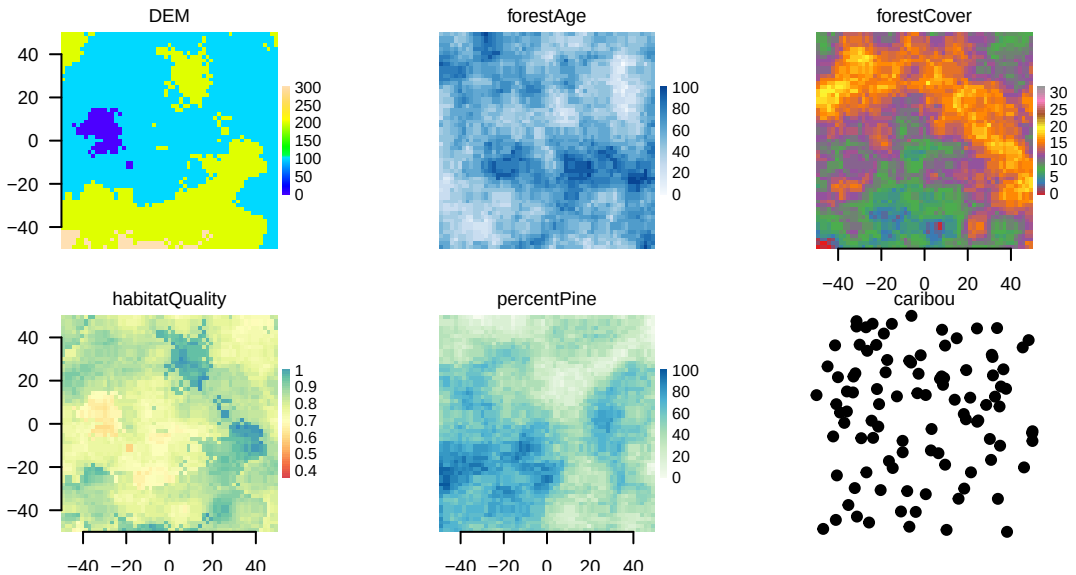
3.1 The add argument

There are essentially 3 types of adding that are addressed by this argument, 1) adding a new plot with enough empty space to accommodate the new plot, 2) without this empty space, and 3) where the device already has a pre-existing plot of the same name.

3.1.1 a new name to a device with enough space

The `Plot` function simply adds the new plot in the available empty space.

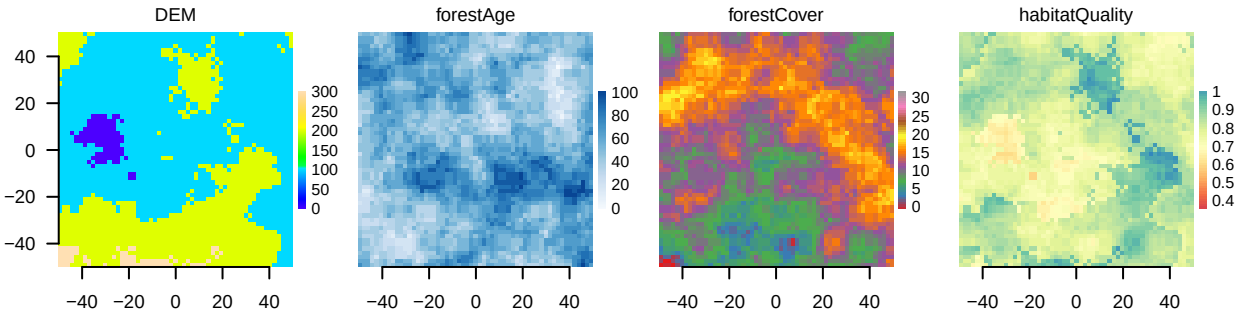
```
Plot(landscape, add=FALSE)
# can add a new plot to the plotting window
Plot(caribou, add=TRUE, axes=FALSE)
```



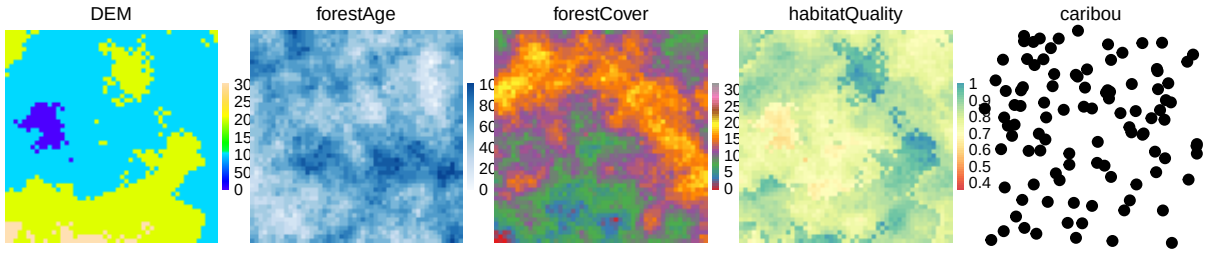
3.1.2 a new name to a device without enough space

The `Plot` function creates a new arrangement, keeping the pre-exising order of plots, and adding the new plots afterwards. The plots will all be a little bit smaller (assuming the device has not changed size), and they will be in different locations on the device.

```
Plot(landscape[[1:4]], add=FALSE)
```



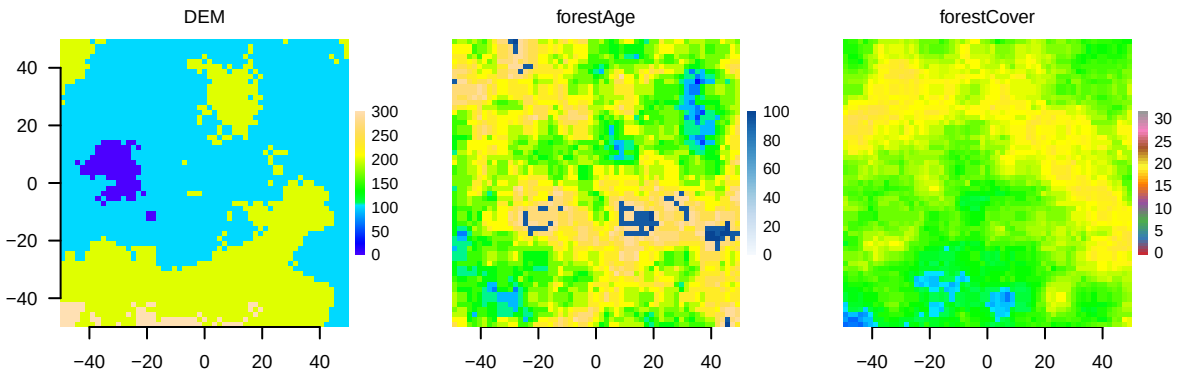

```
# can add a new plot to the plotting window
Plot(caribou, add=TRUE, axes=FALSE)
```



3.1.3 a pre-existing name to a device

The `Plot` function will overplot the new layer in the location as the layer with the same name. If colors in the layer are not transparent, then this will effectively block the previous plot. *This will automatically set legend, title and axes to FALSE.*

```
Plot(landscape[[1:3]], add=FALSE)
landscape$forestAge = (landscape$forestAge +10 %% 100)
landscape$forestCover = (landscape$forestCover +10 %% 30)
# can add a new plot to the plotting window
Plot(landscape[[2:3]], add=TRUE)
```



4 Plotting Speed

A second main purpose of the `Plot` function is to plot as fast as possible so that visual updates, which may be frequent, take as little time as possible. To do this, several automatic calculations are made upon a call to `Plot`. First, the number of plots is compared to the physical size of the device window. If the layers are RasterLayers, then they are subsampled before plotting, automatically scaled to the number of pixels that would be discernible by the human eye. If the layer is a SpatialPoints* object, then a maximum of 10,000 points will be plotted. These defaults can be adjusted by using the `speedup` argument. Broadly, `speedup` is a number >0, where the default is 1. Numbers >1 will plot faster; numbers between 0 and 1 will plot slower. See below for using the `speedup` argument.

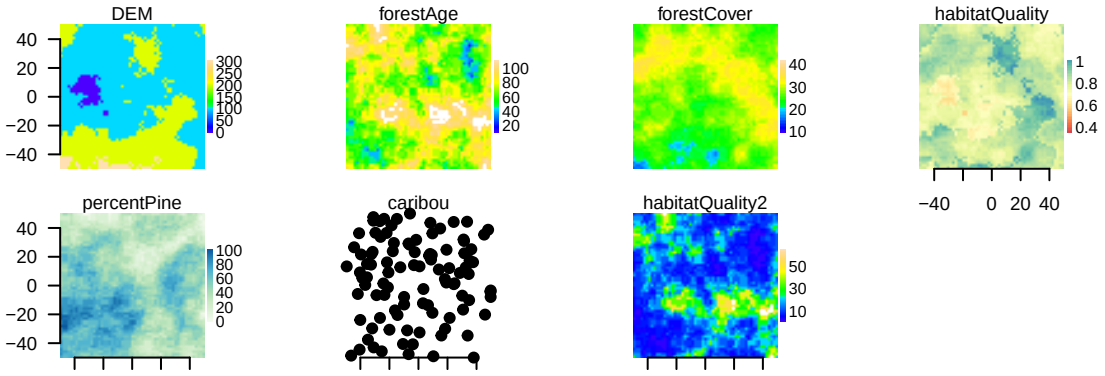
4.1 `speedup`

The `speedup` argument is a *relative* speed increase at the cost of resolution if it is >1. If it is between 0 and 1, it will be a relative speed decrease at the gain of resolution. This may be used successfully when the layer texture is particularly coarse, i.e., there are clusters of identical pixels, so subsampling will have little effect. In the examples below, the speedup gains are modest because the Rasters are relatively small (10,000 pixels). This speed gain will be much greater for larger rasters.

For SpatialPoints, the default is to only plot 10,000 points; if there are more than this in the object, then a random sample will be drawn. Speedup is used as the denominator to determine how many to plot `10000/speedup`.

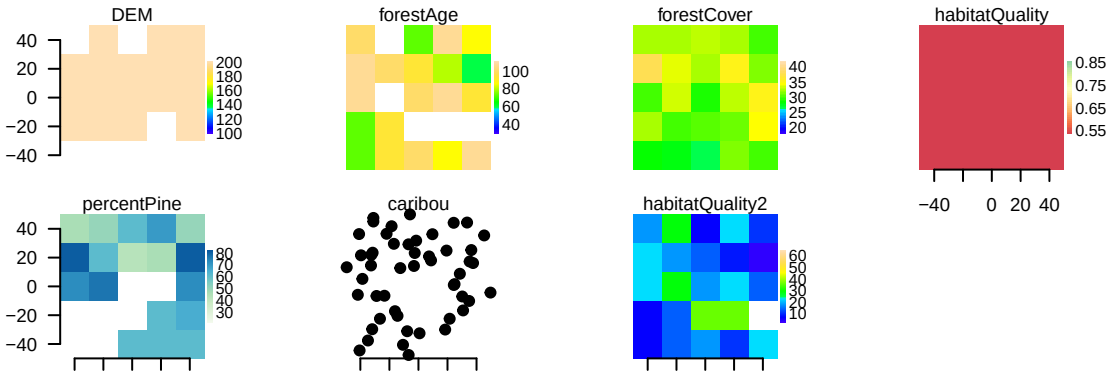
In the example here, the speedup is "extreme" so that an actual speedup can be observed with such small Rasters. This means that the resulting images are very pixelated. This would not be so extreme when the original Raster had 1e8 pixels, but it will plot faster.

```
system.time(Plot(landscape, caribou, habitatQuality2, add=FALSE))
```




```
##    user  system elapsed
##    0.35    0.00    0.34
```

```
system.time(Plot(landscape, caribou, habitatQuality2, speedup=200, add=FALSE))
```



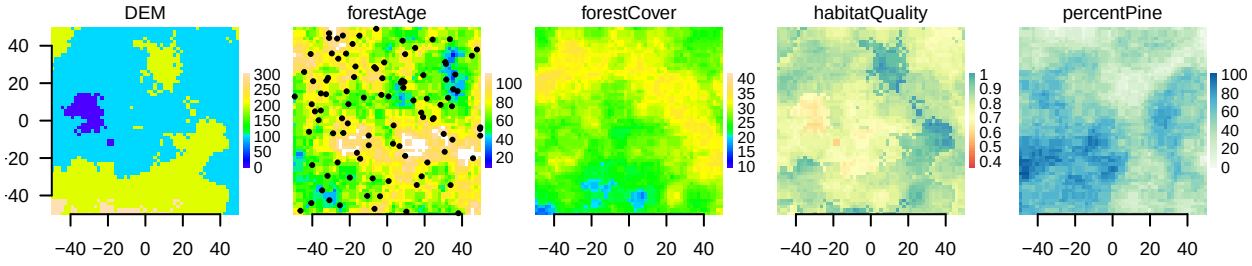

```
##    user  system elapsed
##    0.31    0.00    0.31
```

```
# can add a new plot to the plotting window
```

5 Overplotting: addTo

There are times when it is useful to add a plot to a different plot with a different name. In these cases, the `add` argument will not work. The argument `addTo` will allow plotting of a RasterLayer or SpatialPoints* object on top of a RasterLayer, *that does not share the same name*. This can be useful to see where agents are on a RasterLayer, or if there is transparency on a second RasterLayer, it could be plotted on top of a first RasterLayer.

```
Plot(landscape, add=FALSE)
Plot(caribou, addTo="forestAge", size=2, axes=F)
```



There are several situations that do not plot. A call to Plot where there are two RasterLayers with the same name will return an error. This is true even if one of the layers is in a RasterStack, so is not explicitly named in the call to `Plot`. The following code will create an error.

```
Plot(landscape, caribou, DEM, add=FALSE)
```

6 Using RStudio Plots window

The built in RStudio Plot window is particularly slow. It is recommended to always create a new plotting device whenever real simulations are being done and they will be substantially faster. This may change in

a future version of RStudio. Until then, we have created a function, `dev` which will add devices up to the number in the parenthesis, or switch to that device if it is already open. If an RStudio plot has not been called, dev(2) will create a new device outside RStudio. If a plot has already occurred in RStudio's embedded plot window, then dev(4) will create a new device outside RStudio. dev(4) on its own will either create 3 new devices (device numbers 2, 3 and 4 because device number 1 is never used in R), or 1 new device.

```
# simple:
dev(4)

# better:
#Plot all maps on a new plot windows - Do not use RStudio window
if(is.null(dev.list())) {
   dev(2)
} else {
 if(any(names(dev.list())=="RStudioGD")) {
   dev(which(names(dev.list())=="RStudioGD")+3)
 } else {
   dev(max(dev.list()))
 }
}
```