Colour Schemes

Barry Rowlingson

April 13, 2012

Colours in R

Colour is an important factor in statistical graphics, especially now that cheap colour printers are available. Many of R's graphics functions use colour, and colours can be specified as RGB triples or via a large set of names.

R has several functions that produce or manipulate colours (see `colors`, `rgb`, `hcl` etc) and also functions that produce or manipulate vectors of colours (see `palette`, `rainbow`, `colorRamp`) in its default packages. Add-on packages provide further colour-related functionality (see `colorRamps`, `colorspace`, `dichromat`, `plotrix` packages). The `ggplot2` package also has several functions for colour handling within its graphics system.

What these functions lack is a way of easily controlling the mapping from your data to a colour value. If you are using `rainbow(10)` to plot data that spans the interval [-3,9], what colour is 5.45 going to be?

Colour Scheme Functions

This package aims to fill that gap by providing the means to create *colour scheme* functions. A colour scheme function is an R function that takes data values as argument and produces colours as an output.

Access the package functions in the usual way with the R `library` function:

```
> library(colourschemes)
```

The package provides constructors that return colour scheme functions. The general way of working is like this:

```
> sc = whicheverScheme(data,colours,etc)
> c1 = sc(data)     # vector of colours
> c2 = sc(newdata)  # another vector of colours
```

Firstly you construct a function (`sc` here) using a specific colour scheme function. The constructor may take parameters that are data values, colours, and other things. Secondly you apply the constructed function to some data - either the existing data or new data. At this point the function is fixed according

to the scheme and the parameters, and the returned colours are only dependent on the argument. This way you can be sure that the mapping of data values to colours is constant.

Colours For Factors

A very simple colour scheme is one that produces a colour for each level of a factor. The function `factorScheme` does that:

```
> f = factor(c("a","a","b","a","c"))
> fs = factorScheme(f,c("red","blue","green"))
> fs(f)

      a       a       b       a       c
  "red"   "red"  "blue"   "red" "green"
```
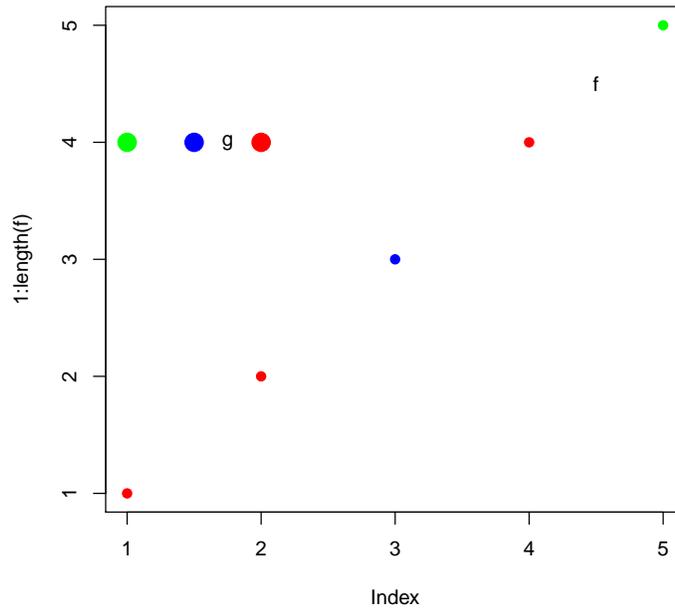
The function `fs` can be re-used with any data that has the same levels as the original source, to guarantee that the same colours are used. Here we use the `fs` function as an argument to the `col=` argument to colour plotted points correctly:

```
> g = factor(c("c","b","a"))
> fs(g)

      c       b       a
"green"  "blue"   "red"

> plot(1:length(f),col=fs(f),pch=19)
> ng = length(g)
> points(seq(1,2,len=ng),rep(4,ng),col=fs(g),pch=20,cex=3)
> text(4.5,4.5,"f")
> text(1.75,4,"g")
```
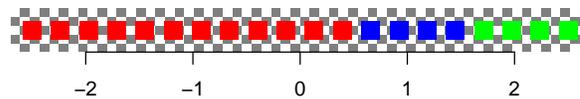
Univariate Continuous Colours

Nearest Colour

The *nearest* scheme takes a data frame of values and colours and produces a colour scheme function that maps data values to the colour corresponding to the nearest value in the data frame.
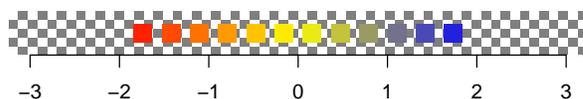
```
> ns = nearestScheme(data.frame(
+    values=c(0,1,2),col=c("red","blue","green")
+    ))
```

Ramp Interpolation

The *ramp interpolation* scheme produces colours that map a range of numeric values onto a colour ramp. The ramp is produced using the base `colorRamp` function, and extra "..." arguments are passed through to that. This gives a way of controlling the specifics of the interpolation such as whether to interpolate in RGB or CIE Lab colour space.
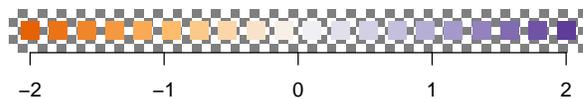
```
> rs = rampInterpolate(limits=c(-2, 2), ramp = c("red", "yellow", "blue"))
```


Note that colours outside the set limits are returned as `NA`, and so result in invisible points.

Since you can give any vector of colour values to `colorRamp`, you can pass through a palette for interpolation, such as one of the `RColorBrewer` palettes:

```
> require(RColorBrewer)
> rs2 = rampInterpolate(c(-2,2),brewer.pal(5,"PuOr"))
```
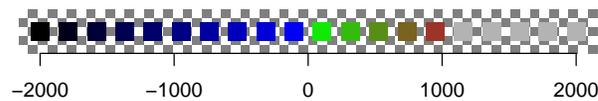

Multiple Ramp Interpolation

In some fields, complex colour ramps are used. For example in topographical mapping there may be a range of blues for underwater values (below zero), then a ramp of green to brown as altitude increases, then a sudden and flat white for the snowline. The *multi ramp* scheme handles these cases.

Its arguments are a data frame of min-max values for each ramp, and a list of ramp specifications as used in the `rampInterpolate` function, but without the facility to pass extra "..." parameters to `colorRamp`. Any gaps between the min-max values will return an `NA` instead of a colour.
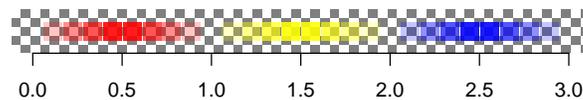
4

Here is a sample topographical multiple ramp scheme. The ocean sinks away to black, and the snowline is at 1000 units up. I've made the ice gray so it shows on a white background:

```
> tramp = multiRamp(rbind(c(-2000,0),c(0,1000),c(1000,9000)),
+    list(c("black","blue"),c("green","brown"),c("gray70","gray70"))
+    )
```


The package also supports transparency. The following colour scheme consists of three bands of red, yellow and blue with transparency that fades in and out.

```
> threeband = multiRamp(rbind(c(0,1),c(1,2),c(2,3)),
+    list(c("#ff000000","#ff0000ff","#ff000000"),
+        c("#ffff0000","#ffff00ff","#ffff0000"),
+        c("#0000ff00","#0000ffff","#0000ff00")))
```


Multivariate Continuous Colours

The rgb Scheme

This scheme simply maps sets of three values in your data to red, green, and blue values which it feeds to the `rgb` function. By default it scales red to the full range of the first data value, green to the full range of the second, and so on. The `equalScale` parameter can be used so that each of the colours is scaled depending on the global scale of all three input data values.

```
> m3=matrix(sample(1:12),4,3)
> print(m3)
```

```
      [,1] [,2] [,3]
[1,]    8    2    5
[2,]    6   12   11
[3,]    4   10    7
[4,]    3    1    9
```

```
> r=rgbScheme(m3,equalScale=TRUE)
> r(m3)
```

```
[1] "#A2175D" "#74FFE8" "#46D18B" "#2E00B9"
```

```
> r=rgbScheme(m3,equalScale=FALSE)
> r(m3)
```

```
[1] "#FF1700" "#99FFFF" "#33D155" "#0000AA"
```

The hsv Scheme

This scheme is similar to the rgb scheme but instead passes the values to the `hsv` function, so the three columns of the data refer to hue, saturation, and colour value.

```
> h=hsvScheme(m3,equalScale=FALSE)
> h(m3)
```

```
[1] "#000000" "#0066FF" "#47550F" "#AAAAAA"
```

The "f3" Scheme

The rgb and hsv schemes are specialised cases of the "f3" scheme. This scheme takes a function as parameter in its constructor that takes three equal-length vector arguments that range from 0 to 1 and returns colour values. The existing `rgb` and `hsv` functions in R fit this pattern, but you can write your own. The constructor also takes the `equalScale` argument and needs a label string to identify your scheme.

For example, suppose you have bivariate data and want to represent the first variable as a variation in the redness, and the second variable as variation in greenness, and with no blue. Write an `rg` function:

```
> rg <- function(r,g,b){return(rgb(r,g,0))}
```
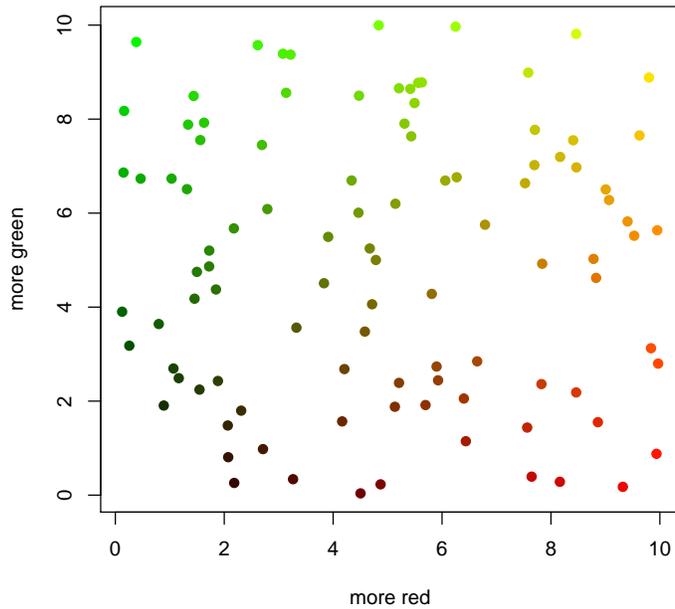
That function maps values of `r` and `g` from 0 to 1. So now we wrap it in an `f3Scheme` function, to get a colour scheme that is bound to our data ranges:

```
> m=matrix(10*runif(300),ncol=3)
> rg <- function(r,g,b){return(rgb(r,g,0))}
> rgs=f3Scheme(m,rg,"red-green scheme")
> plot(m[,1],m[,2],col=rgs(m),pch=19,xlab="more red",ylab="more green")
```
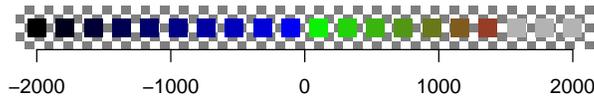
6

Scheme Information

Each colour scheme function object has a `schemeData` method that returns a list giving the specification of the scheme. This list can be fed back into the scheme constructor function with `do.call`. If unmodified, you will get back an identical colour scheme function, but this is useful if you wish to slightly modify an existing colour scheme function. Here we raise the snowline from the multiple ramp example to 1800:

```
> sd = schemeData(tramp)
> print(sd$ranges)

      [,1] [,2]
[1,] -2000    0
[2,]     0 1000
[3,]  1000 9000

> # lets raise the snowline
> sd$ranges[2,2]=1500
> sd$ranges[3,1]=1500
> tramp2 = do.call(multiRamp,sd)
```

Writing Your Own Colour Schemes

If you want to write your own colour scheme functions that conform to the existing ones, here is how you do it.

First, define a new scheme function:

```
myScheme = function(a,b,c){
```

This function takes whatever data and options are needed. Then you need to make sure these options are evaluated in order to fix their values. Use the `force` function for this:

```
force(a)
force(b)
force(c)
```

Next we create a function within this function that only takes the new data as argument:

```
f = function(data){
  colours=someFunction(data,a,b,c)
  return(colours)
}
```

Note that this function only takes the data as an argument, but it does have the values of `a`, `b`, and `c` available within it. These values will be in the environment of the function when it is called, since they are there as the function is created.

Then we need to make sure we return the function with a new class, and a 'type' string attribute which is a short description used by the print method.

```
class(f)= c("myScheme","colourScheme","function")
attr(f,"type")="my scheme"
return(f)
}
```

Finally, we write a `schemeData` method for our new scheme. It will be passed the function object with the arguments wrapped in its environment:

```
schemeData.myScheme = function(object){
  eo=environment(object)
  return(list(a=get("a",env=eo),
              b=get("b",env=eo),
              c=get("c",env=eo)))
}
```

See the R code for the included functions for more.

Using colourschemes with `image`

Most base graphics functions such as `plot`, `lines` and friends have a `col=` argument to directly specify colours of plotted entities. However, no such option is available for the `image` function. This function can only plot against a colour palette.
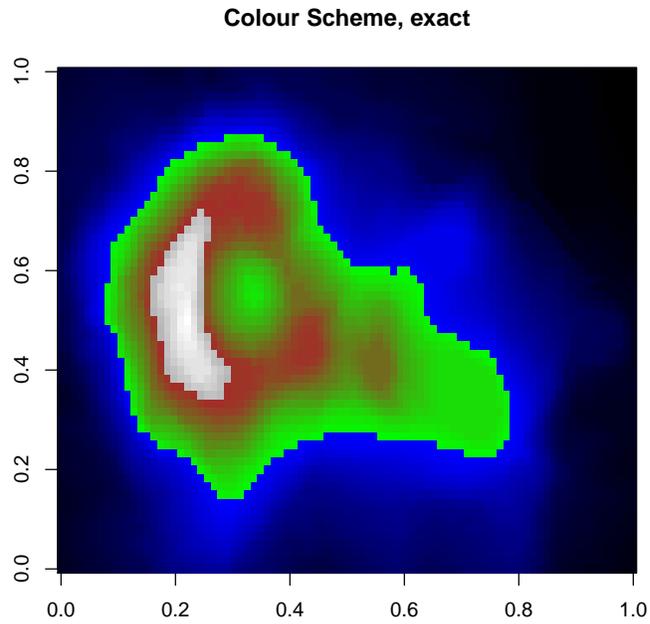
One way round this is to create a palette of the same length as the number of cells in the image, and create a matrix of the same shape as the data but containing the values from 1 to the number of cells. Then we use `image` on that matrix, and the colour is then looked up in our long palette.

In this example we create a modified version of the volcano dataset scaled to give us some negative values. We then use a colour scheme for topological data with a snowline at 1500 units:
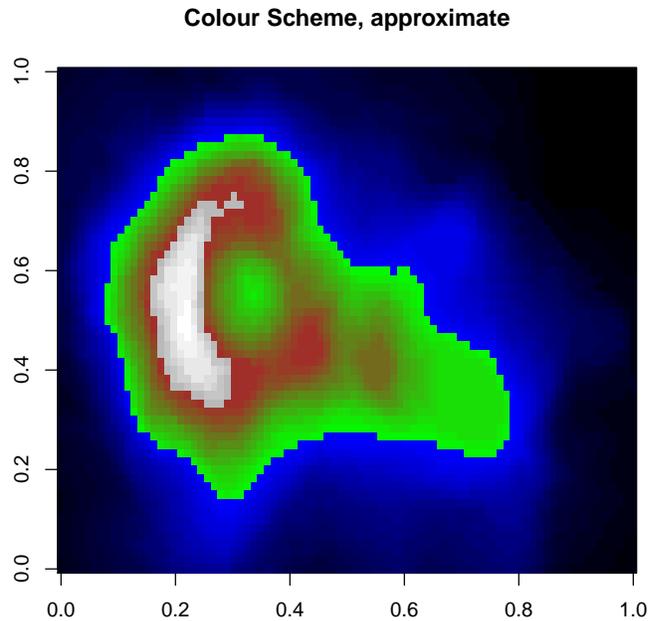
```
> v=((volcano-94)*40)-2000
> vramp = multiRamp(rbind(c(-2000,0),c(0,1500),c(1500,max(v))),
+   list(c("black","blue"),c("green","brown"),c("gray70","white"))
+   )
> image(matrix(1:length(v),nrow=nrow(v)),col=vramp(v))
> title("Colour Scheme, exact")
```

Colour Scheme, exact

This method is expensive in terms of memory since it involves the creation of objects of the same size as your data, but it does produce the exact correct colours. Another option is to create a fixed palette from your colour scheme based on the range of your data:
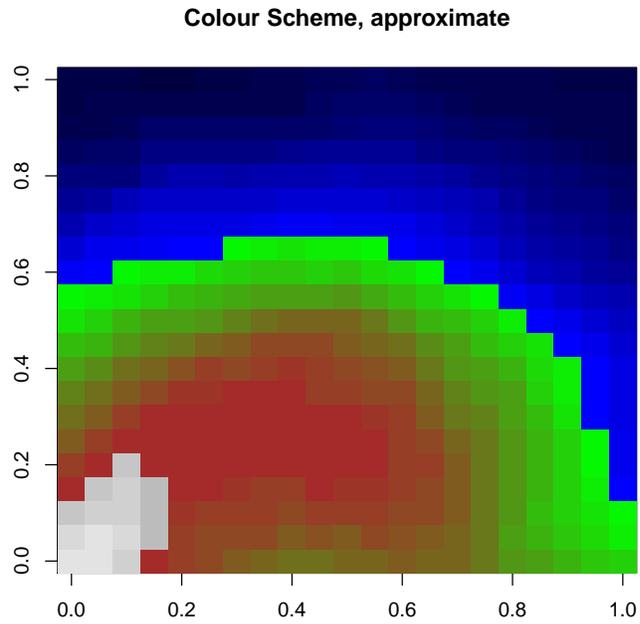
```
> p=vramp(seq(min(v),max(v),len=50))
> image(v,col=p)
> title("Colour Scheme, approximate")
```

Colour Scheme, approximate

This gives an approximation to the exact colours, something most easily noticed in the difference in the icecap. A better approximation can be made by using a longer palette.

When reusing the colour scheme on a subset or other data set using this method, just construct a palette from the range of the subset. Here we zoom in on part of the volcano and use the same colour scheme:

```
> vs=v[20:40,40:60]
> ps=vramp(seq(min(vs),max(vs),len=50))
> image(vs,col=ps)
> title("Colour Scheme, approximate")
```
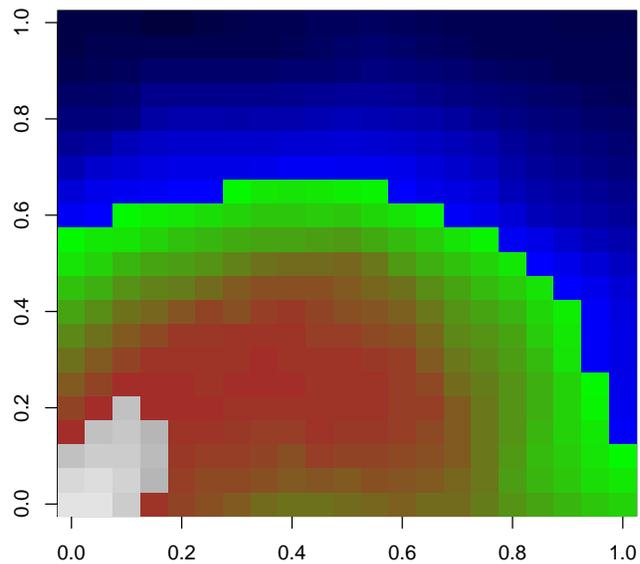
Colour Scheme, approximate

Note that the icecap now looks closer to the original region than before because we are using a better approximation for our palette - the 50 colours are now interpolating between the smaller range of values in our subset matrix.

The exact method for a subset looks like this:

```
> image(matrix(1:length(vs),nrow=nrow(vs)),col=vramp(vs))
> title("Colour Scheme, exact")
```

Colour Scheme, exact

More Examples

The following code produces some more example colour schemes.

```
> set.seed(123)
> require(RColorBrewer)
> x=seq(-2.5, 2.5, len=50)
> srange = max(abs(range(x)))*c(-1,1)
> schemes = list()
> schemes[[length(schemes)+1]] =
+   nearestScheme(data.frame(values=c(-1.5,-0.5,0,0.5,1.5),
+                             col=brewer.pal(5,"Set3")))
> schemes[[length(schemes)+1]] =
+   rampInterpolate(c(-2,2), c("red","yellow","blue"))
> schemes[[length(schemes)+1]] =
+   rampInterpolate(c(-2,2), c("red","yellow","blue"),
+                    interpolate="spline",bias=0.3)
> schemes[[length(schemes)+1]] =
+   rampInterpolate(c(-2,2), brewer.pal(5,"PuOr"))
> schemes[[length(schemes)+1]] =
+   rampInterpolate(srange, brewer.pal(5,"PuOr"))
```

```
> schemes[[length(schemes)+1]] =
+   multiRamp(rbind(c(-2,0), c(0,.6), c(.6,2)),
+             list(c("black","blue"),
+                   c("yellow","brown"),
+                   c("green","white")))
> nt = length(schemes)
> plot(srange, c(1,nt+1), type='n', xlab="", ylab="",
+      axes=FALSE, main="colour schemes")
> axis(1)
> box()
> for(i in 1:nt){
+
+   points(x,rep(i,length(x)),col=schemes[[i]](x),pch=19,cex=1.5)
+   for(k in 1:1){
+     xr = runif(30,min(x),max(x))
+     points(xr,rep(i,length(xr))+.2*k,col=schemes[[i]](xr),pch=19,cex=1)
+   }
+   text(min(x),i+.4,class(schemes[[i]])[1],adj=c(0,.5))
+   legend(par()$usr[2],i+0.4,xjust=1,yjust=0.5,
+          as.character(pretty(x)),
+          fill=schemes[[i]](pretty(x)),
+          horiz=TRUE,bty='n',cex=1,x.intersp=.1,text.width=0.05)
+ }
>
```

14

colour schemes