# Colour Schemes

Barry Rowlingson

June 8, 2009

## 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 simple colour scheme function 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)
```

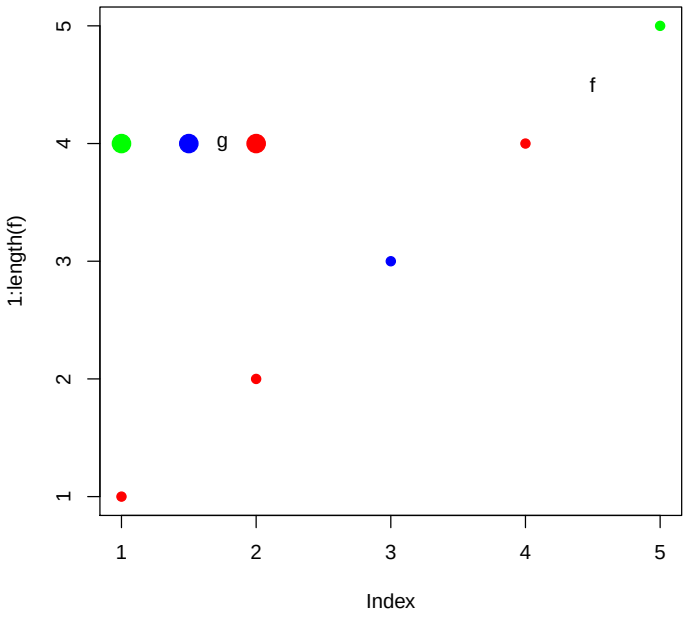
```
    a     a     b     a     c
"red" "red" "blue" "red" "green"
```

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")
```

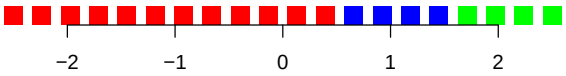



# Univariate Continuous Colours

## 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.

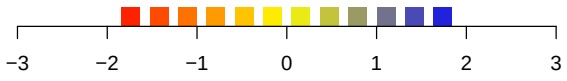
```
> ns = nearestScheme(data.frame(
+    values=c(0,1,2),col=c("red","blue","green")
+    ))
```



## 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.

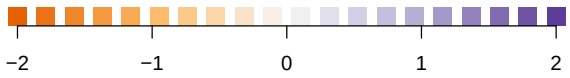
```
> rs = rampInterpolate(limits=c(-2, 2), ramp = c("red", "yellow", "blue"))
```




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"))
```



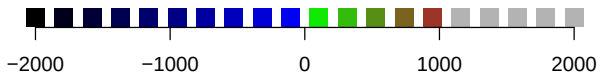

## Multiple Ramp Interpolation

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.

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"))
+    )
```




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

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

```
[1] "#D10046" "#5DA2FF" "#B92E74" "#E8178B"
```

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

```
[1] "#D50000" "#00FFFF" "#AA4940" "#FF2460"
```

## 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" "#FF0000" "#2E2E40" "#605252"
```

## The "f3" Scheme

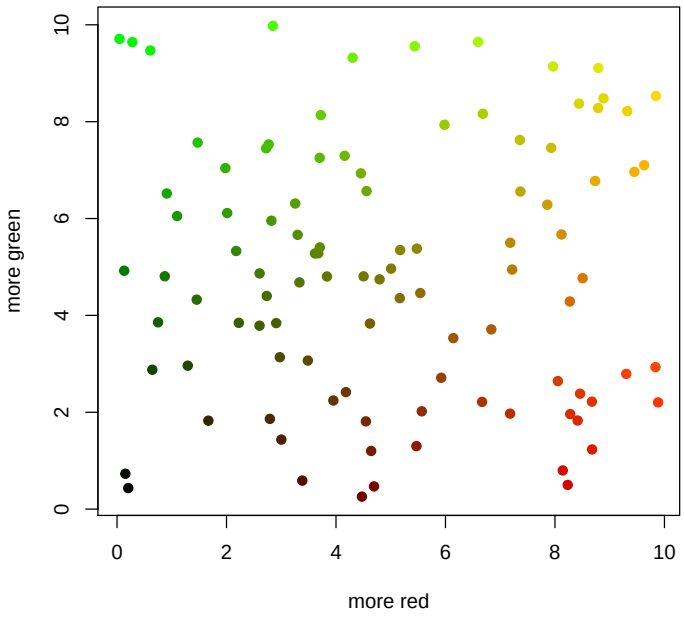
The rgb and hsv schemes are specialised cases of the "f3" scheme. This scheme takes a function as parameter in its constructor that takes three equal-length vector arguments that range from 0 to 1 and returns colour values. The existing `rgb` and `hsv` functions in R fit this pattern, but you can write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



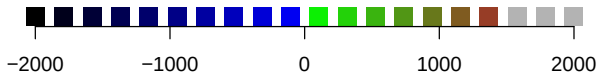

## Scheme Information

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

```
myScheme = function(options){
  force(options)
  f = function(data){
    colours=someFunction(data,options)
    return(colours)
  }
  class(f)= c("myScheme","colourScheme","function")
  attr(f,"type")="my scheme"
  return(f)
  }

schemeData.myScheme = function(object){
  return(list(options=get("options",env=environment(object))))
  }
```

## Examples

```
> 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
> 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[
+ }
```