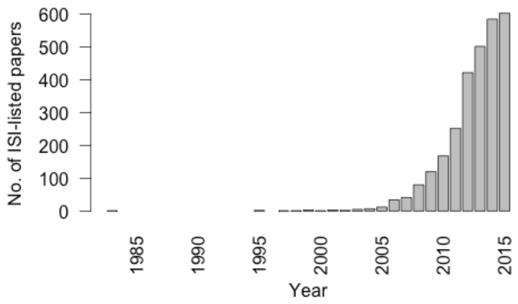
BIOL 3551: Species distributions in R

This workshop is intended, primarily, to introduce you to the technique of species distribution modelling. Over the last decade, Species Distribution Models (SDMs) have become probably the most widely-used techniques for forecasting the consequences of climate change for biodiversity. The rapid growth of this approach is shown by the number of papers that use it:



ISI-listed papers (i.e. listed on Web of Science) that include the phrase "Species distribution mode" (where the * indicates any text, so allows for model / models / modeling / modelling)*

SDMs are statistical descriptions of the geographical area in which a species (or population of a species) is found. They can incorporate as predictors (also referred to as "independent variables") a variety of spatial information, including climate, land use, terrain (e.g., slope or aspect), geology and, potentially, the occurrence of other species (although it is questionable whether the latter are independent variables). Occasionally, SDMs also use factors such as latitude and longitude as predictors. This is fine if the purpose is simply to describe where the organism occurs, but is rather pointless if the aim is to explain that occurrence, or to predict future changes in occurrence. More on SDMs can be found in the comprehensive review by Elith & Leathwick (2009).

SDMs can be derived by assessing the relationship between where a species has been recorded to occur and the characteristics of those sites relative to sites where it has not been recorded (Phillips et al. 2006). This carries the risk that sites where the species has not been recorded are simply those at which it has not been sought (i.e., they are **false absences**). As a result, most people prefer to work with presence/absence data (where absences are surveyed to the same extent as presences, and so are more likely to represent **true absences**) (Brotons et al. 2004), or even abundance data (Howard et al. 2014).

Here, we will look at an example of creating SDMs using presence/absence data and relating them exclusively to bioclimate variables (a process sometimes described as "climate envelope modelling"). To do this will require you to increase your familiarity with R, learning a variety of new techniques, including installing packages, writing functions, importing data, mapping data and, perhaps most importantly, building on your experience with the enormously helpful family of "apply" functions. By the end of this workshop, you should have done the following:

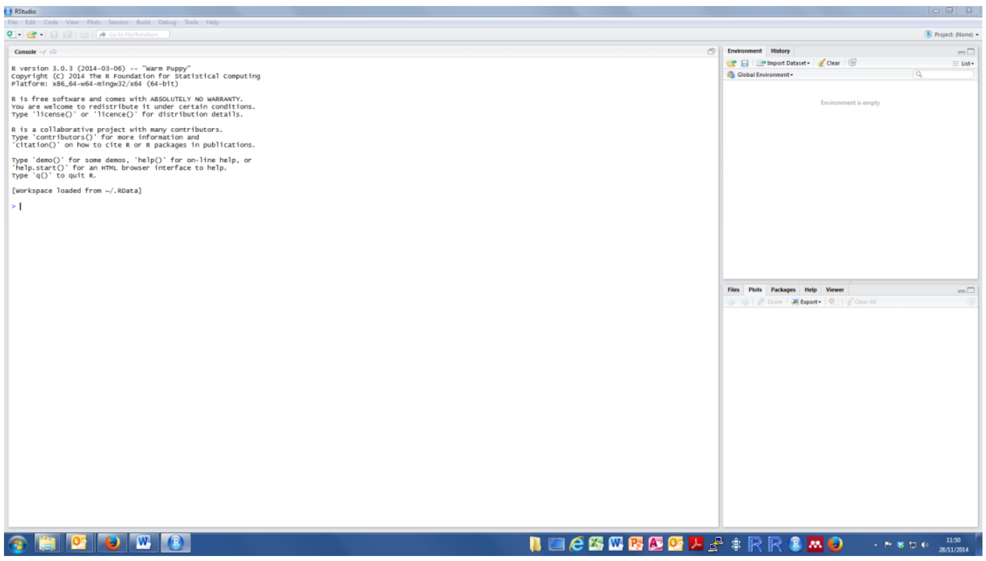
- Imported and mapped species distribution data
- Imported and mapped bioclimate data
- Run a simple general linear model (GLM) to relate a species' distribution to bioclimate
- Assessed the quality of the fitted model
- Made predictions about the future distribution of climatic suitability for the species
- Considered some of the advantages and limitations of SDMs

* The arguments don't need to have the names by which map.data will refer to them. When you send information to map.data, it will refer to the 1st, 2nd and 3rd pieces of information you send as 'dat', 'varnm' and 'period', respectively.

1. RStudio

For earlier workshops, you might have used the basic R environment for entering and running code. You should have used RStudio in the modelling workshop but might not have done so. You should definitely do so now. Like R, RStudio is free. Some of its advantages over the basic R environment include: its window setup (which makes it easy to view variable values, graphics, help files, scripts and the console simultaneously), auto-indenting of code (which helps to keep your code organised), auto-completion of brackets (which reduces lots of minor errors), and auto-formatting of code (which helps you to differentiate comments, code, reserved words, etc.).

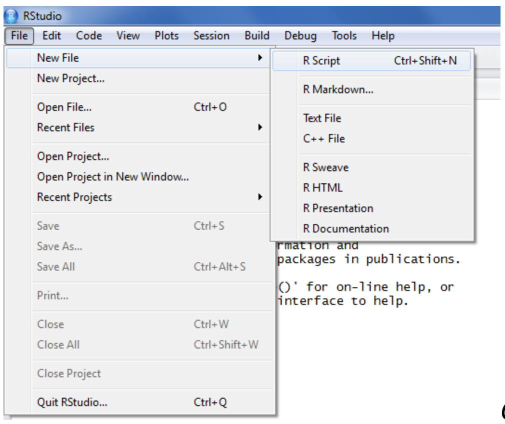
If you open up RStudio, it should look like this:



```
R version 3.0.3 (2014-03-06) -- "Warm Puppy"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> |
```

The RStudio environment with no script file open

As with R, you could simply type commands into the left hand pane (the console). However, as discussed previously, a script file is infinitely to be preferred, so that you have a record of what you did, and so that you can easily edit it and re-run it. To open a new script file, use the file menu, as shown:



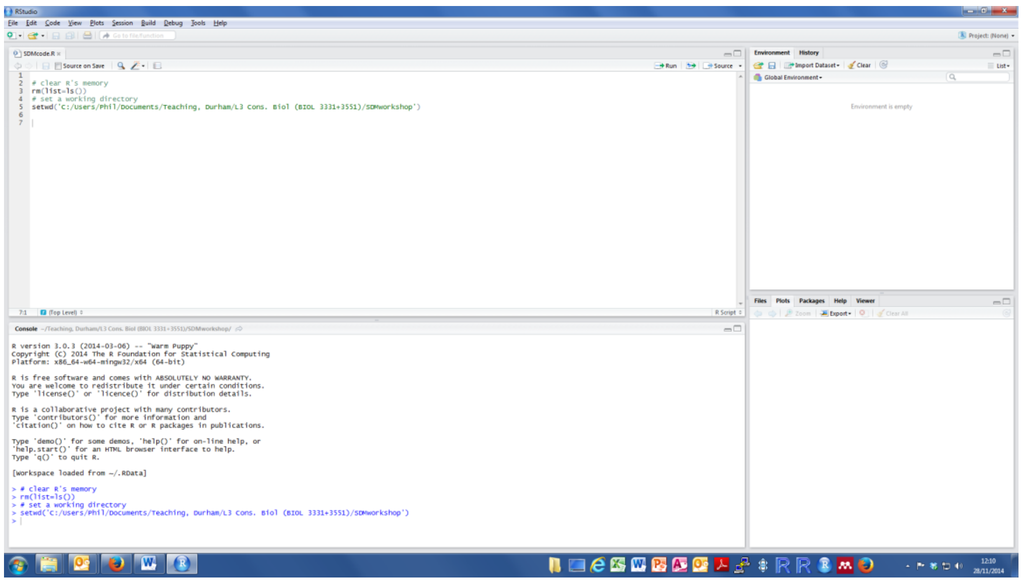
```
rmation and
packages in publications.

()' for on-line help, or
interface to help.
```

Creating a new script using the RStudio menu

As you will see, you could also press CTRL-SHIFT-N. You can start writing code in the top left panel. It is useful to start code files by doing 2 things. First, you might like to clear R's memory with the line

* The arguments don't need to have the names by which map.data will refer to them. When you send information to map.data, it will refer to the 1st, 2nd and 3rd pieces of information you send as 'dat', 'varnm' and 'period', respectively.

rm(list=ls()). Second, you should set a working directory to point to a single folder where you will store inputs and outputs relevant to this exercise. I will use the command **setwd('C:/Users /Phil/Documents/Teaching, Durham/L3 Cons. Biol (BIOL 3331+3551)/SDMworkshop')** but your code will need to point to a folder that you can access on the computer that you're working on. Remember that you can annotate your code by adding comments preceded by **#**. In RStudio, you can run your code by highlighting it and pressing either CTRL-ENTER or CTRL-R. You should also save the code file and then do so with some frequency thereafter, using CTRL-S. Your screen should now look something like this:



```

# clear R's memory
rm(list=ls())
# set a working directory
setwd('C:/Users/Phil/Documents/Teaching, Durham/L3 Cons. Biol (BIOL 3331+3551)/SDMworkshop')

```

```
R version 3.0.3 (2014-03-06) -- "Warm Puppy"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> # clear R's memory
> rm(list=ls())
> # set a working directory
> setwd('C:/Users/Phil/Documents/Teaching, Durham/L3 Cons. Biol (BIOL 3331+3551)/SDMworkshop')
> 
```

The RStudio environment with a script window at top left and the console at bottom left

2. Installing packages: an introduction to functions and lapply

One of the most useful things about R is that other people are constantly writing code that can do things that you need R to do. Often, they make this code available as a "package". A package is basically a library of functions that will take input data and return something you want. You can install a package and make the new functions available by typing

install.packages('foo')
library(foo)

where "foo" is the name of the package (note: there is no such package; for some reason, it's a convention among programmers to use "foo" as a meaningless name, a bit like you might use "Fred Bloggs" to refer to a generic person). Note that the first command (which installs the package) requires the package name in quotes, whereas the second (which makes the functions available) does not require quotes. Go figure, as they say on the other side of the Atlantic, all too frequently.

One problem with having these lines in your code is that every time you run your code, these lines will also run – reinstalling every package you want, slowing down your code and, potentially, confusing R. The way round this is to have a list of packages that you will need and check them, installing each only if it isn't already installed. To do this, we need to learn two new tools. I went

* The arguments don't need to have the names by which map.data will refer to them. When you send information to map.data, it will refer to the 1st, 2nd and 3rd pieces of information you send as 'dat', 'varnm' and 'period', respectively.

through these in the last workshop but, given their importance, and given – also – the extent to which students sometimes struggle with them, I think it does no harm to reiterate (after all, some time has elapsed). The first reminder is about the lapply function. This does something to every item in a list or vector; in this way, it's a bit like a loop – which you encountered in the introductory workshop. What it does to each item in the list is determined by a function, as I will describe.

Functions are sections of code that take in 'arguments' and do something with them. Arguments can be any form of variable or data. Most of the commands you have used in R (e.g. print, for, setwd, etc.) are functions – but you haven't yet written any functions of your own. Functions that you write can be anonymous (i.e. they don't have a name, so are only useful in the section of code in which they appear), or can have a name (in which case you can call them from other parts of the code by using their name). We'll look at both types. First, an example of an anonymous function might be a function that you're unlikely to use again, designed to produce a small set of graphs with different properties. Consider this code:

```
foo <- lapply(c(10,20,50),function(x){
 fnm <- paste('Histogram, random sample, SD =',x,'.jpg',sep='')
 dat <- rnorm(200,200,x)
 jpeg(fnm,width=4,height=3,units='in',res=200)
 hist(dat,main=paste('Mean = 200, SD =',x,', n = 200',sep=''),xlab='Value')
 dev.off()
})
```

Recall what I said about the use of "foo". Here, foo is the name given to what the lapply function returns. However, as the function actually sends output files to your computer, you won't need to ask to see what foo holds (hence the dismissive use of foo).

The lapply function has two arguments, both found within the brackets that are coloured red in the online version of this document. The first argument it takes is the vector c(10,20,50). This is just a list of three numbers. What lapply will do with each of those numbers in turn is determined by its second argument – in this case, an **anonymous function (AF)**. The AF is the one starting "function(x)" and the details of the AF are everything in curly brackets after that (coloured blue in the online version of this document). Like in a loop command, we have assigned an arbitrary name to the values that the AF will use. That name is **x** but it could have been anything. **x** will take the values in the first argument of the lapply function, one at a time. – i.e. the AF will be executed 3 times, first with x = 10, then with x = 20 and finally with x = 50.

What does the AF do? You have seen most of the commands in the AF before. The first line is just a paste command that makes a text string which we can subsequently use as a file name. The string is identical each time the AF runs, except that it will include the value of x (which changes each time). The second line of the AF generates a set of 200 normally distributed random numbers, drawn from a distribution with mean = 200 and SD = x, and calls that set of numbers "dat". Thus, the spread of numbers should get larger as x gets larger. The third line of the AF says that subsequent code (up until the command "dev.off()" is encountered) should be used to produce a jpeg document of 4 inches by 3 inches, with a resolution of 200 ppi. The fourth line creates a histogram of the numbers in "dat" and specifies a title and x-axis label. The fifth line halts the production of the jpeg file.

* The arguments don't need to have the names by which map.data will refer to them. When you send information to map.data, it will refer to the 1st, 2nd and 3rd pieces of information you send as 'dat', 'varnm' and 'period', respectively.

Hopefully, you can see that the code should produce 3 jpegs with different file names, each depicting a frequency distribution of random numbers with the standard deviation specified by the first argument of the lapply function. Check your working directory to see if this happened. If you have any doubt about what should have happened, don't forget that it's easy to check what different functions do by typing, for example, **?paste** into the RStudio console. The value of this AF example is questionable but it does show you how you can use lapply to apply a function to each item in a list or vector. Why not use a loop for this? I don't have a glib answer to that question – but it is widely recognised that, in R, apply functions (including lapply) are more rapid and more efficient with memory than loops are (unless you reserve memory for your function's output in advance – but I don't want to go into that, or people will start to mutter the word "geek" in a way that I suspect is meant to be audible). They are also much more efficient when your first argument is something more complex than a simple vector of numbers.

So, that was an anonymous function – but what about named functions that might actually be useful? Try entering the following:

```
# Function to install packages
check.install.packages <- function(list.packages){
  foo <- lapply(list.packages, function(req.lib){
    is.installed <- is.element(req.lib, installed.packages()[,1])
    if(is.installed == FALSE){install.packages(req.lib)}
    require(req.lib, character.only = TRUE)
  })
}
```

This creates a function called "check.install.packages". Because the function has a name, we can call it from anywhere in the code. Much as you would say **print()** with whatever arguments you want printed in the brackets, once you have run the code above you can say **check.install.packages()** with whatever list of package names you like. Briefly, the function will apply an internal AF (using lapply) to every item in the object **list.packages**. If list.packages is a list of packages you want, the AF will check each of them to see if it's installed and, if it isn't, install it. The AF will then make available the contents of the package [using require(), which is similar to library()].

Needless to say, I don't yet know what packages we might need in the course of this tutorial (I'm only just writing it), so for now, we must content ourselves with assigning a single package name to the list.packages object:

```
list.packages <- c('PresenceAbsence')
check.install.packages(list.packages)
```

This package (PresenceAbsence) provides a set of functions useful when evaluating the results of presence-absence models. Later on, we will use some of those functions.

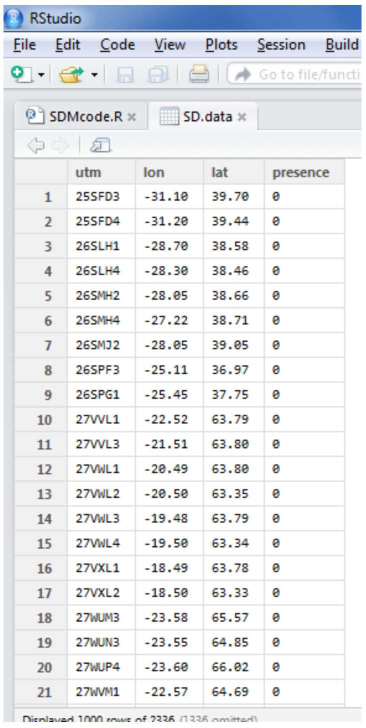
* The arguments don't need to have the names by which map.data will refer to them. When you send information to map.data, it will refer to the 1st, 2nd and 3rd pieces of information you send as 'dat', 'varnm' and 'period', respectively.

3. Importing species distribution data

Species distribution data usually come in the form of grids. The grids are composed of rows and columns of cells, identified by the latitude and longitude of their centres. Usually, each grid cell is associated with a value for presence/absence of the focal species: a 1 for present and a 0 for absent. We will use the example of the Dartford warbler (*Sylvia undata*), a handsome little chap most usually associated with more Mediterranean climes. The distribution data for the Dartford warbler are available in a file called "Sylvia_undata_dist.csv", which I will put on DUO. You should download the file and place it in the folder that you have designated as your working directory. Your job then is to read the data into RStudio. You will do this using the read.csv function. The only argument that you need to supply that function with is the name of the file that you want to read in. Hence, you can write:

```
SD.data <- read.csv('Sylvia_undata_dist.csv')
```

You should check that the data imported correctly. There are several ways to do this. One is to click on the name of the resultant object (SD.data) in the top right panel of RStudio. If the name doesn't appear, the data haven't imported correctly. If it does, clicking on it should open a new tab next to your script window, in which you can view the imported data as a table. It should look as shown.



	utm	lon	lat	presence
1	25SFD3	-31.10	39.70	0
2	25SFD4	-31.20	39.44	0
3	26SLH1	-28.70	38.58	0
4	26SLH4	-28.30	38.46	0
5	26SMH2	-28.05	38.66	0
6	26SMH4	-27.22	38.71	0
7	26SMJ2	-28.05	39.05	0
8	26SPF3	-25.11	36.97	0
9	26SPG1	-25.45	37.75	0
10	27VVL1	-22.52	63.79	0
11	27VVL3	-21.51	63.80	0
12	27VWL1	-20.49	63.80	0
13	27VWL2	-20.50	63.35	0
14	27VWL3	-19.48	63.79	0
15	27VWL4	-19.50	63.34	0
16	27VXL1	-18.49	63.78	0
17	27VXL2	-18.50	63.33	0
18	27WUM3	-23.58	65.57	0
19	27WUN3	-23.55	64.85	0
20	27WUP4	-23.60	66.02	0
21	27WVM1	-22.57	64.69	0

Displayed 1000 rows of 2336 (1336 omitted)

You should also be able to see that all rows and columns have imported correctly. Specifically, in the Environment window (top right of RStudio), after the name of the object (SD.data), it should say "2336 obs. of 4 variables". If it doesn't, you will need to check what's happened.

Notice that the data include a row for each grid cell, each labelled with a utm, or Universal Transverse Mercator code. Each cell also has a longitude (lon) and latitude (lat), as well as a column indicating whether the species is present in that cell (presence).

The next challenge is to plot the distribution data to see where the species occurs at present.

* The arguments don't need to have the names by which map.data will refer to them. When you send information to map.data, it will refer to the 1st, 2nd and 3rd pieces of information you send as 'dat', 'varnm' and 'period', respectively.

4. Plotting species distribution data

For a simple plot of the species' distribution, I recommend a two-step process. First, plot all the data points (regardless of presence or absence), then plot only the presences in a different colour. The first stage will provide the backdrop to the geographical area that we're considering, whilst the second stage will overlay that with the data points to show the species' current range. To do this, enter the following code:

```
jpeg('Dartford_Warbler_distribution.jpg',height=6,width=4.5,units='in',res=200)
par(las=1)
plot(SD.data$lon,SD.data$lat,pch=19,cex=0.7,col='grey90',xlab='Longitude',ylab='Latitude')
pres <- subset(SD.data, presence == 1)
points(pres$lon,pres$lat,pch=19,cex=0.7,col='darkgreen')
dev.off()
```

You have seen the first and last lines before. These simply ensure that the intervening lines are used to create a graphical file of the dimensions, resolution and name that we specified. The second line simply sets a graphical parameter (which, in this instance, ensures that y-axis numbering will be written vertically; don't ask why – I must have had a nasty run-in with a reclining number when I was a child). The third line creates the plot and plots a point for every pair of longitudes and latitudes in SD.data. The points are circular (pch=19) and smaller than the default (cex=0.7); they are also light grey (col='grey90').

The fourth line of the above code is used to create a subset of the original data frame. The function used to do this is **subset**. subset takes 2 arguments: the name of the data frame (or table) to be subsetted (in this case, SDdata) and a logical statement describing the criteria by which to subset (in this case specifying that we are only interested in rows of data in which presence == 1). The double = is not a typo. When you use a single =, that assigns one thing to another. For logical queries (i.e., the query of whether one thing equals another) we must use ==.

The fifth line of the above code adds data points to the plot but only for longitudes and latitudes of data in the subset of data associated with presences (i.e. the new data frame, pres). The points are as before but are of a different colour to make them stand out from the background.

Examine the plot you produced by running the code above. You will see that the grid is not entirely regular. For example, the points become further apart as you go further north.

Q1: Why do you think the points become further apart as latitude increases?
Q2: Why do you think it's important that grid cells are of approximately equal area?
Q3: Where do you think these data come from and do you think that they are equally reliable from all parts of the area shown?

5. Importing and mapping bioclimate data

Next, we will import the bioclimatic data, which includes several variables that, for each cell, have been averaged over the period 1961-1990 (a period during which the species' distribution was

* The arguments don't need to have the names by which map.data will refer to them. When you send information to map.data, it will refer to the 1st, 2nd and 3rd pieces of information you send as 'dat', 'varnm' and 'period', respectively.

mapped; furthermore, a period for most of which the climate was relatively stable). The file we want is called “Bioclimate_1961-1990.csv”. Import it and examine it. For what it’s worth, I called the imported object “Biocli”.

As before, the data have a row for each grid cell. The first three columns match those from before, whilst the next three contain the bioclimate data. They include, specifically:

- gdd5, the sum of residual temperatures over 5°C across all days when the temperature exceeded 5°C (a measure of how much opportunity there was for vegetation to grow)
- mtco, the mean temperature of the coldest month of the year
- apet, the ratio of actual to potential evapotranspiration (a measure of water stress)

Q4: Why are these referred to as bioclimatic data?

We want to plot spatial variation in all three of these variables. Clearly, it would make sense to write a function to do this (rather than writing code to plot each separately). What’s more, we might later want to plot other bioclimatic data in the same way, so we should probably make this a named function to allow us to make use of it then as well.

Consider the following code:

```
map.data <- function(dat, varnm, period){
 # create a file name for output
 filenm <- paste(period,'_',varnm,'.jpg',sep='')
 # set up output and plotting parameters
 jpeg(filenm,height=6,width=9,units='in',res=200)
 par(mfrow=c(1,2),las=1)
 # identify variable roles
 x <- as.numeric(dat[,1]); y <- as.numeric(dat[,2]); z <- as.numeric(dat[,3])
 # find range of z variable to map z values to colours
 lo <- min(z); hi <- max(z); range <- hi-lo
 # map each z value to a number between 1 and 255
 cols <- 1 + 254 * (z - lo)/range
 # create the plot
 plot(x,y,
    pch=19,cex=0.6,
    col=colorRampPalette(c("blue", "yellow", "red"))(255)[cols],
    xlab='Longitude',ylab='Latitude')
 # create a key
 plot(c(0,1),c(0,1),type='n',bty='n',axes=F,xlab='',ylab='') # a dummy plot to set up the boundaries
within which we must produce the key
 for (i in 1:255) lines(c(0,0.3),rep(1-i/255,2),
       col=colorRampPalette(c("blue", "yellow", "red"))(255)[i],lwd=2)
 mtext(varnm,at=0.15)
 for (i in c(1,255)){
   if (i == 1) val <- lo else val <- hi
   text(0.33,1-i/255,format(round(val,3),nsmall=3),pos=4)
```

* The arguments don’t need to have the names by which map.data will refer to them. When you send information to map.data, it will refer to the 1st, 2nd and 3rd pieces of information you send as ‘dat’, ‘varnm’ and ‘period’, respectively.

```
 }
 dev.off()
}
```

This looks long and fearsome. However, worry not: (a) it is long because I have inserted comments throughout; (b) much of it is code you have come across before; and (c) I'm going to explain parts of it that might be less familiar.

First, consider **map.data <- function(dat, varnm, period=NULL)**. This tells you that what we are making is a named function (called "map.data") which requires 3 arguments*:

- an object called dat (a data frame with 3 columns, corresponding to x, y and z coordinates of your plot, where z is the variable for which we are interested in plotting values spatially – i.e. either GDD5, MTCO or APET, at this stage)
- a string called varnm, which is the name of the z variable you are sending the function
- a string called period, which is shows the period over which the bioclimate data are averaged (which might be important if we later use the same function to plot bioclimate data from a different period).

Next, look at the line **x <- as.numeric(dat[,1]); y <- as.numeric(dat[,2]); z <- as.numeric(dat[,3])**. This demonstrates two things. First, you can have multiple commands on a single line in R, as long as they are separated by semicolons. Second, this shows how to turn a column from a data frame into a simple vector of numbers (using the function **as.numeric**).

Following that line are some lines of code that relate the range of z values (whatever that range might be) to a scale from 1 to 255. This is necessary because, when we plot the data, we want to colour each point according to its z value. The colours come from a customised palate that we create using **colorRampPalette**, which we have requested to go from blue (for low values), via yellow to red (for high values). The resultant colours are on a scale from 1 to 255, so that is why we need to calibrate our z values to that scale. Whatever the lowest z value is, we want to specify that the point(s) with that z value should be plotted in colour 1 (blue); whatever the highest value of z is, we want to specify that the point(s) with that z value should be plotted in colour 255 (red). All the other points need to be coloured on a smooth scale between these extremes, depending on the z value at those points and how far it is between the minimum and maximum values of z.

The lines following the comment **# create a key**, are all intended to draw a colour key to relate the colours to the z values. They do this in a second panel (note the mfrow command at the start of the whole function, which sets up a plotting area with 1 row and 2 columns – i.e., two plotting regions next to each other). The main map of bioclimate data appears in the left hand panel and the key is drawn in the right hand panel. All the numbers within the code that draws the key are chosen to try to get the key to appear in a suitable location, so that the scale is represented smoothly from the minimum to the maximum value of z. Notice that I have used **for** commands in a couple of places in this code. R purists would suck their teeth and roll their eyes, but R purists aren't writing this in the small hours of Sunday morning.

Now we can make use of the named function we've created. To do that, I've made use of lapply with an anonymous function:

* The arguments don't need to have the names by which map.data will refer to them. When you send information to map.data, it will refer to the 1st, 2nd and 3rd pieces of information you send as 'dat', 'varnm' and 'period', respectively.

```
period <- '1961-1990'
foo <- lapply(4:6, function(C){
  focal.data <- Biocli[,c(2,3,C)]
  focal.name <- toupper(names(Biocli[C]))
  map.data(focal.data,focal.name,period)
})
```

The bioclimate variables are located in columns 4 to 6 of the Biocli data frame. Consequently, the lapply function steps through those values, with **C** taking the values 4, 5 and 6 in turn. In each case, it selects columns 2 (longitude), 3 (latitude) and the focal column to send to our map.data function. It picks out the name of column C (and converts it to upper case) to send as well. It then calls the map.data function that we have just created. Check that this works for you and produces 3 suitably named graphics files in your working directory. Have a look at the climate data and see if you can see what the Dartford warbler likes.

Q5: Where do you think the bioclimate data come from? Do you think all data points are equally reliable?

6. Creating an SDM

To create an SDM, it is quite useful if the presence/absence data and the bioclimatic data are in the same data frame. This is easily accomplished using the **merge** function. Try:

```
all.dat <- merge(Biocli,SD.data)
```

This merges two data frames on the basis of columns with identical names. Thus, it simply adds the presence/absence data column from SD.data to the Biocli data frame.

Q6: Why not simply bind the 'presence' column to the Biocli data frame (which is easily done using 'cbind' to bind together columns)?

We now have a single object (all.dat) that contains both our predictors (gdd5, mtco, apet) and our response variable (presence). We want to create a **general linear model** or **GLM** to relate the response to the predictors. A GLM with continuous predictor variables is also known as a multiple regression. You will be familiar with linear regression, in which $y = a + bx$. Here, y is the response variable and x is the predictor. Both a and b are constants, and it is the job of the statistical package that you are using to find out the best values of a and b. A multiple regression is similar, but takes the form $y = a + b_1x_1 + b_2x_2 \dots + b_nx_n$, where there are n different predictors. Here, the job of the statistical package is to determine the best values for a and all b_is.

If we assume that all predictors are broadly linearly related to the response, then we would simply have 3 predictors. However, experience tells us that the response often shows complex relationships with the predictors that are far from linear. It is not uncommon to allow those relationships to be 3rd order polynomials (where a second order polynomial is a quadratic relationship and a third order polynomial is a cubic relationship). A 3rd order polynomial with a single predictor would be written as $y = a + b_1x + b_2x^2 + b_3x^3$. Fortunately, R doesn't require us to write things out at that length. We can just write y ~ poly(x,3). The tilde (~) is read as "is a function of" and the poly(x,3) specifies that we want a third order polynomial. Given that we have 3 predictors, we will need to write y ~ poly(x_1,3) + poly(x_2,3) + poly(x_3,3). One other point is that the response variable is not continuous. Instead, it's what's known as a binomial outcome (i.e., all values are one of two numbers: 0 or 1). Now we know all this, we can fit the model. The only code we need to do this is the line:

SDM <- glm(presence ~ poly(gdd5,3) + poly(mtco,3) + poly(apet,3), data=all.dat, family='binomial')

Hopefully, given the explanation above, you will be able to see why we write the model like this. We can look at a summary of the model's fit by writing **summary(SDM)**. A key part of the output that this produces is the table of coefficients. Using the notation above, the first two columns of the table are the estimates and standard errors for a (the intercept) and all the b_is. The last column of the table gives the probability that the presence/absence data would have been seen if the variable in each row played no role at all. You will see that those probabilities are all very low – i.e., all variables in the model are highly significant.

Now, what if we want to look at the suitabilities that this model predicts (where suitability is, for each cell, the probability that the species occurs in that cell)? This also takes very little effort. In fact, we can add these suitabilities to the all.dat data frame using the line:

all.dat$Suitability <- predict(SDM, type='response')

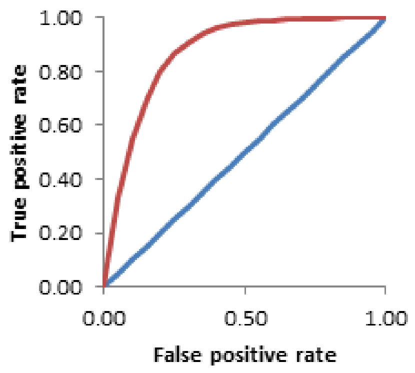
This simply adds a column (called "suitability") to the data frame, with values predicted by the SDM. We might want to look at how these values vary spatially. Now, we can take a brief moment to wallow in smugness: we already have a function that plots spatial variation in a continuous variable! Thus, plotting these suitabilities is no more complicated than sending the relevant information to our function:

map.data(all.dat[,c(2,3,8)],'Suitability','Present day')

Have a look at the predictions of the SDM and compare them to the original distribution.

7. Assessing model fit

Does the model look like a good fit? Whether it does or not, we usually have to provide some metric of model fit. One in reasonably common use is known as the **AUC**, or **Area Under the Curve of the receiver operating characteristic**. This area relates to a plot of the true positive classification rate against the false positive classification rate. For example, consider the blue line in this graph. This shows the expectation if a model has no discriminatory power at all. If we increase the threshold at which we expect occupancy (i.e., if we increase the level of suitability required for us to predict occupancy), then with each increase in threshold we include just as many unoccupied cells as occupied ones. Most models do better



than this, with a relationship more like that shown by the red line. A perfect model would show a point at which we correctly predicted all occupied cells (i.e., true positive rate = 1.0) and didn't predict occupancy in any unoccupied cell (i.e. false positive rate = 0.0). In that case, the red line would pass through the point (0,1) on the graph. So, reality lies between a disastrous model that looks like the blue line and a model that perfectly describes the whole upper left triangle of the graph (everything above and to the left of the blue line). A sensible measure to describe this is to ask what portion of the upper left triangle lies under the red curve. This is bounded by 0 and 1, with 0 indicating a hopeless model and 1 indicating a perfect model. In general, models with an AUC > 0.7 are usually thought to be pretty good.

To calculate AUC could be quite a long-winded process but fortunately for us, others have been through that process before. Earlier, we installed a package called PresenceAbsence and this includes a function called auc, which computes AUC for us. The function requires a data frame with a column of cell IDs, a column of observed data and a column of predicted suitabilities. We can make that with the line: **eval.data <- all.dat[,c(1,7,8)]**. We can then compute the AUC using:

(AUC <- round(auc(eval.data,st.dev=FALSE),3))

This calls the auc function and sends it the data frame we just made. It also specifies that we aren't interested in the standard deviation of the AUC value. The result returned by the auc function is rounded to 3 decimal places and assigned to the variable 'AUC'. Remember that, because the whole line is in brackets, output will appear immediately. This should tell you that the AUC of our model is 0.968.

To be fair, the model does seem to have done rather well. There are several caveats to this, which I don't intend to rectify here (this is, after all, a mere 3 hour workshop). Nevertheless, I can't completely overlook them, as much as I'd like to. The first caveat is that we simply specified a single model and assumed that it was an appropriate way to describe the relationship between the predictors and the response. The model included cubic relationships between the response variable and each of three predictors. However, it might be that simpler models would have done almost as good a job (without the same risk of *overfitting* the model to the data). To clarify what I mean by overfitting, consider a situation in which we measure the heights of 5 Durham students, 3 of whom are called Rupert and 2 of whom are called Tarquin. I might choose to measure their heights and take an overall mean height, or I might wonder if height is associated with name. If, by chance, the 3 Ruperts were taller than the two Tarquins, it might appear that knowing a man's name provides useful information about height and, thus, I might suggest that the average of the 3 Ruperts' heights is a better prediction of the height of people called Rupert and vice versa for Tarquin. Arguably, however, my model would be overfitted as a consequence of the specific sample I measured. A different sample might have yielded a different inference. A model that was less overfitted to the data would have used the mean of all 5 heights as an estimate of the mean height of male students at Durham. That model wouldn't do too badly as a description of the sample and, in the long run, would probably do better as a prediction of the heights of all male students (unless, of course, there is a significant difference in the heights of Ruperts and Tarquins: discuss).

To avoid overfitting, we try to fit the most *parsimonious* model possible – i.e. the simplest model that, nonetheless, provides an adequate description of our data. The question of what constitutes 'adequate' is another issue. In the past, people have generally used something called stepwise regression, which involves starting with a simple model and adding parameters sequentially until further parameters are no longer significant (or the opposite process, until all non-significant parameters have been removed). This is now recognised to be bad practice (Whittingham et al. 2006). One alternative is to use likelihood ratio tests, which compare the fits of models that differ by one parameter. More flexible, however, are information criteria, such as Akaike's Information Criterion (AIC). These are useful because they allow us to compare many different types of model – including lots of (arguably superior) alternatives to GLMs. Sadly, time doesn't permit me to go into that here but, if anyone's interested, it could be a fascinating conversation for another time …

Another caveat to our model fitting is the problem known as *Spatial Autocorrelation*. This problem arises because most species have a constrained distribution. Many processes might constrain the distribution but, because climates are more similar between nearby cells than between distant cells, a constrained distribution is likely to create the appearance of climatic preferences, whether or not preferences exist. This is a thorny problem to overcome.

Q7: What types of processes might constrain species distributions in ways that are unrelated to contemporary climate?
Q8: How might we overcome the problem of spatial autocorrelation?

8. Import and map future bioclimate data

Predictions of future climates are made using very different models (including models based on either global or regional circulation), different time frames and different scenarios for future economics and carbon emissions. We will use one scenario (known as the A1 scenario) for 2080, produced some years ago by a model known as HadGEM (the Hadley Centre Global Environment Model). The details are unimportant in the context of this short introductory workshop but you should be aware of the variety of models and scenarios available.

Future climate data are in a file called "HadGem A1 2080.csv". Import the file and examine its contents. This file has utm codes but not longitudes and latitudes. It also spans a larger area than we are interested in. To remedy both problems, we can merge the data with coordinates by using **merge(all.dat[,1:3],Future)** (where 'Future' is what I called the 2080 bioclimate data when I imported them). This loses about 50 cells for which the future climate data were suspect and which were not imported.

Map the future climate data. Notice that, frustratingly, the scales for each bioclimate plot have shifted between the two time periods (1961-1990 and 2080). This makes it difficult to make a direct comparison. We can do something useful here to remedy that. Specifically, we can alter the map.data function, changing the declaration to:
map.data <- function(dat, varnm, period, lo=NA, hi=NA)

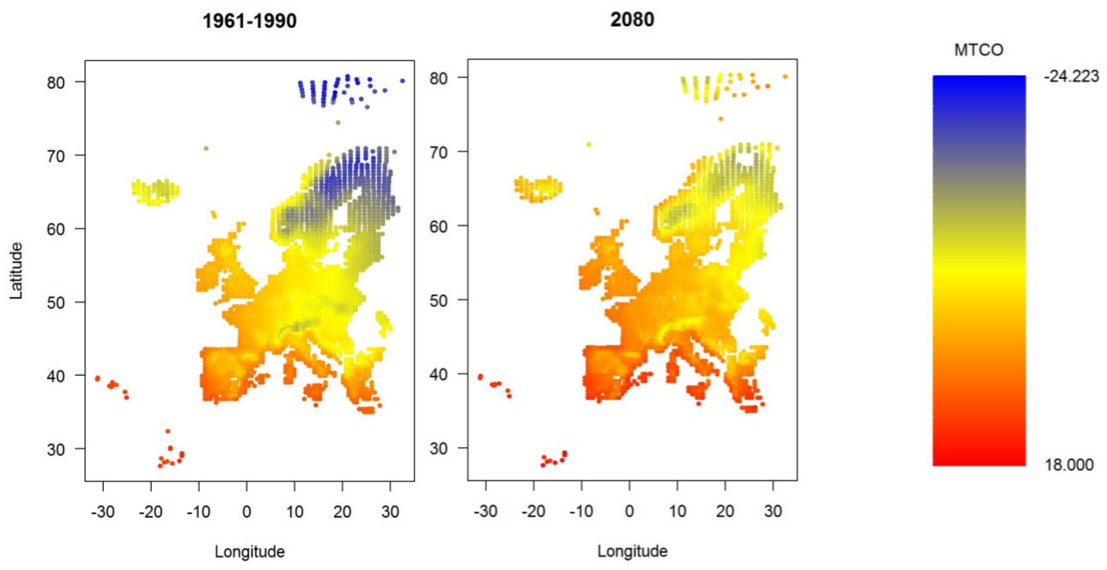
This means that the default values for two variables that are used in the map.data function, lo and hi, are both NA. These variables are used to scale the values that we are plotting to the range of colours available (1 to 255; see section 5). It is now possible to run the function without specifying the values of lo and hi (as we have done up to now), or to send it values of lo and hi to override these defaults. Here, we can make use of the latter option. One other thing we need to do is to modify the map.data function, changing the code under the comment **# find range of z variable to map z values to colours** to:

```
 if (is.na(lo) | is.na(hi)){
   lo <- min(z); hi <- max(z)}
 range <- hi-lo
```

This ensures that we only use lo and hi values taken from the data we are mapping if no lo and hi values were supplied when the function was called (note that | is used in logical functions in place of the word "or"). Having changed the map.data function, you need to re-run the code from the start of the function to the end of it, in order to change what R does when you call that function. Now, we can call the function using code like this:

```
period <- c('1961-1990','2080')
foo <- lapply(period, function(P){
 foo <- lapply(4:6, function(C){
   if (P == '2080') focal.data <- Future.clim[,c(2,3,C)] else focal.data <- Biocli[,c(2,3,C)]
   focal.name <- toupper(names(focal.data[3]))
   lo <- min(Biocli[,C],Future.clim[,C])
   hi <- max(Biocli[,C],Future.clim[,C])
   map.data(focal.data,focal.name,P,lo,hi)
 })
})
```

The code steps through both time periods (the first lapply function) and the three focal variables (the second lapply function). In each case, it draws a graph by calling map.data. Importantly, however, minimum and maximum values for scaling the graph are sent to map.data, having been calculated based on the full range of values for the climate variable in the earlier and the future time period. Your maps should look something like this example for MTCO (I further modified the map.data function to add a title to the main plot, using **main=period**):



MTCO variation in Europe, 1961-1990 and 2080

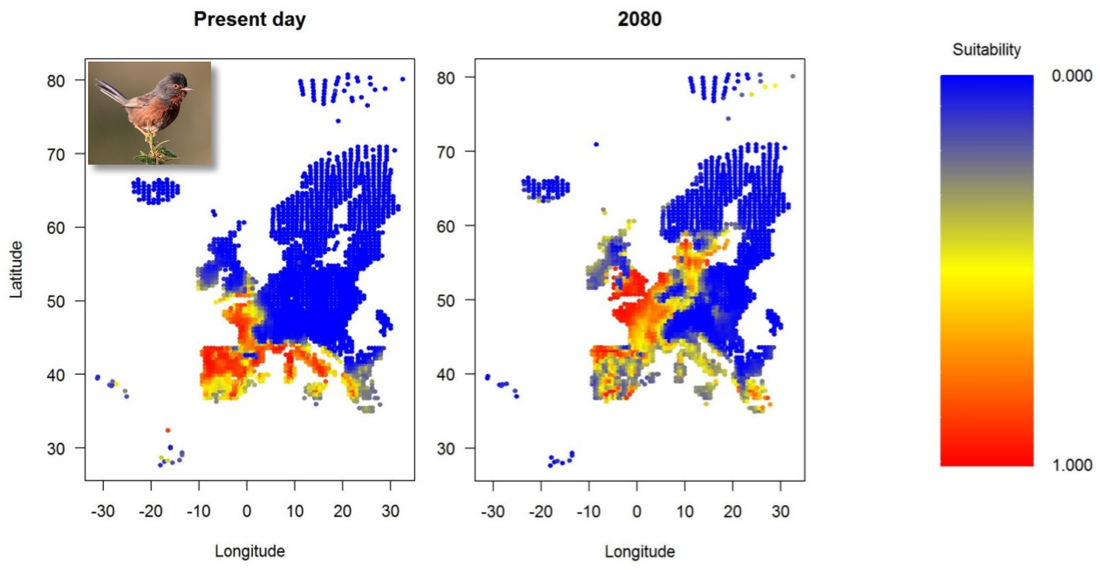
9. Predict future suitabilities

Now that we have a model for suitability (SDM) and we have future bioclimate data (I called this Future.clim when I merged the imported future climate data with the coordinates of cells of interest), we can make predictions about suitability in the future. To do this, we only need the line:
Future.clim$suitability <- predict(SDM, newdata=Future.clim, type='response')

This is very similar to what we did in Section 6 to predict suitabilities based on 1961-1990 bioclimate. Here, however, we are making sure that the predict function uses new data to make its predictions. The new data are the relevant variables in the object Future.clim. As before it is easy to plot these out and we might take advantage of our modified map.data function to make sure that predictions of suitabilities are on the same scale:

```
period <- c('Present day','2080')
foo <- lapply(period, function(P){
  if (P == '2080') focal.data <- Future.clim[,c(2,3,7)] else focal.data <- all.dat[,c(2,3,8)]
  map.data(focal.data,'Suitability',P,lo=0,hi=1)
})
```

This should yield maps along the following lines:



Present (modelled) and future (predicted) suitabilities for the Dartford warbler

These predictions are consistent with a lot of predictions for birds in the northern hemisphere: suitability is expected to decline in the south of the range and increase in the north of the range. The distribution is expected to shift northward and, potentially, eastward.

Q9: Thinking back to the previous R-based workshop, what's a major limitation regarding making predictions of where species will be in the future, based solely on their relationship with climate?
Q10: Why might the predicted suitabilities bear a poor resemblance to the distribution of the species in 2080?

Q11: Recall that we had to redraw the climate graphs for the 2 periods, in order to put them on the same scale? Why was this? Why might it matter that climate variables in the future will lie beyond the range of variables to which we fitted our model?
Q12: Can you identify 4 different sources of uncertainty that will affect our predictions of future suitability? How might we represent uncertainty when we present results of predictive exercises like these?
Q13: How might we use predictions like these for conservation?

Hopefully, subsequent reference to SDMs in lectures will make more sense now that you've actually had a go at constructing an SDM yourselves.

References (provided only in case you want to follow up any of the specific points raised; I would, however, urge you to look at Elith and Leathwick 2009 if you are interested in SDMs)

Brotons et al. (2004) Presence-absence versus presence-only modelling methods for predicting bird habitat suitability. *Ecography* 27: 437-448

Elith & Leathwick (2009) Species Distribution Models: ecological explanation and prediction across space and time. *Ann. Rev. Ecol., Evol. & Syst.* 40: 677-697

Howard et al. (2014) Improving species distribution models: the value of data on abundance. *Methods Ecol. Evol.* 5: 506-513.

Phillips et al. (2006) Maximum entropy modeling of species geographic distributions. *Ecol. Model.* 190: 231-259

Whittingham et al. (2006) Why do we still use stepwise modelling in ecology and behaviour? *J. Anim. Ecol.* 75, 1182-1189