The apply functions

Chris

25/01/2019

- The apply() family
- Some data
- tapply()
- lapply()
- Using lapply() for simulations
- mclapply()
- sapply()
- vapply()
- mapply()
- rapply()
- apply()
- Competition!

The apply() family

The apply suite of functions are base (i.e. including in R from the get go, not morally questionable) and, as the name suggests, apply specified functions to data. They are incridibly useful when used right, but can sometimes be a tad confusing. This walkthrough should get you up and running, and will also introduce some basic parallelization protocols for massively speeding up your analyses.

There are a whole bunch of apply functoins, some of which I had never heard of until I wrote this intruduction:

```
rm(list=ls())
##the apply functions
apply()
lapply()
sapply()
vapply()
mapply()
rapply()
tapply()
```

They are all fairly obviously named, e.g. lapply is list-apply, mapply is multivariate-apply, etc.

I find tapply() and lapply() the most useful, but then I don't work much with matrices or arrays.

So, here we go!

Some data

First off, lets load in some data. We will use the iris data set from last session.

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
##we are going to two other columns to the data set.
##the first is going to be the location that these measurements were made in
iris$location<-sample(1:3, size=nrow(iris), replace=T)

##and the second the sex of the plants measured:
iris$sex<-sample(c("m", "f"), size=nrow(iris), replace=T)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species location sex
## 1          5.1         3.5          1.4         0.2  setosa        1   m
## 2          4.9         3.0          1.4         0.2  setosa        3   f
## 3          4.7         3.2          1.3         0.2  setosa        1   f
## 4          4.6         3.1          1.5         0.2  setosa        3   f
## 5          5.0         3.6          1.4         0.2  setosa        2   f
## 6          5.4         3.9          1.7         0.4  setosa        1   m
```

tapply()

tapply is perhaphs the most basic, and useful, of the apply functions. It basically says, apply a function to a column in my data set, where 1 or more other columns specify how that data should be subset.

….Probably easier to just give you an example:

```
##the frist argument is the column you want to apply the function to
##the second is a list of the columns you want to subset the data by
##the third is the function you are applying
tapply(iris$Sepal.Length, list(iris$Species), mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

So you can see that when we specify we want the mean for each species (i.e. subset the data by the column “Species” using “list(iris$Species)”) tapply will return a vector of mean values, with the column names as the factors from the “Species” column.

Of course you dont have to just use the mean function…

Other functions you can pass

Of course we dont have to just calculate the mean, a whole bunch of functions can be used in tapply:

```
tapply(iris$Sepal.Length, list(iris$Species), max)
```

```
##     setosa versicolor  virginica
##        5.8        7.0        7.9
```

```
tapply(iris$Sepal.Length, list(iris$Species), min)
```

```
##     setosa versicolor  virginica
##        4.3        4.9        4.9
```

```
tapply(iris$Sepal.Length, list(iris$Species), sd)
```

```
##     setosa versicolor  virginica
##  0.3524897  0.5161711  0.6358796
```

```
tapply(iris$Sepal.Length, list(iris$Species), median)
```

```
##     setosa versicolor  virginica
##        5.0        5.9        6.5
```

We can also really easily write our own functions and add them in. Here to calculate standard error:

```
se<-function(x)sd(x)/sqrt(length(x))

tapply(iris$Sepal.Length, list(iris$Species), se)
```

```
##     setosa versicolor  virginica
## 0.04984957 0.07299762 0.08992695
```

One thing to note here is that when we pass a funciton to tapply it is going to put a vector into that function (in the above example, the values in the Sepal.Length column). Because of this we cant have a function that takes multiple arguments from the data - i.e. it has to be able to run on just the vector of values being passed to it. In the “se” example above you can see that the function just takes a vector (which we have named x) and uses only those values to calculate the standard error, rather than needing information from a second column in the data.frame.

Multiple arguments

Its also reall easy to add multiple column arguments in, say if we want the mean of each species at each location. To do this we just add to the list() argument:

```
tapply(iris$Sepal.Length, list(iris$Species, iris$location), mean)
```

```
##                   1        2        3
## setosa     5.080000 5.009524 4.921429
## versicolor 6.077778 5.844444 5.871429
## virginica  6.704545 6.553333 6.430769
```

Its worth noting at this point that the order in which you put your arguments into the list() function determins the shape of the data that come out:

```
tapply(iris$Sepal.Length, list(iris$Species, iris$location), mean)
```

```
##                   1        2        3
## setosa     5.080000 5.009524 4.921429
## versicolor 6.077778 5.844444 5.871429
## virginica  6.704545 6.553333 6.430769
```

```
tapply(iris$Sepal.Length, list(iris$location, iris$Species), mean)
```

```
##     setosa versicolor virginica
## 1 5.080000   6.077778  6.704545
## 2 5.009524   5.844444  6.553333
## 3 4.921429   5.871429  6.430769
```

tapply() can handle as many of these arguments as you have in your data, but things start to get a bit messy. Lets include sex as well:

```
tapply(iris$Sepal.Length, list(iris$Species,
                               iris$location,
                               iris$sex), mean)
```

```
## , , f
##
##                   1        2        3
## setosa     5.012500 4.955556 4.900000
## versicolor 6.133333 5.742857 5.842857
## virginica  6.792308 6.755556 6.750000
##
## , , m
##
##                   1        2        3
## setosa     5.157143 5.050000 4.950000
## versicolor 5.966667 5.909091 5.900000
## virginica  6.577778 6.250000 6.288889
```

Ok, you can see now that tapply has given us back the answers in the form of an array (basically a stacked list, but more on those below). Not particularly easy to read. However tapply() works fantastically with the functions in the reshape2 package, particularly the melt() function. Lets take a look:

```
require(reshape2)
```

```
## Loading required package: reshape2
```

```
melt(tapply(iris$Sepal.Length, list(iris$Species,
                               iris$location,
                               iris$sex), mean))
```

```
##          Var1 Var2 Var3    value
## 1      setosa    1    f 5.012500
## 2  versicolor    1    f 6.133333
## 3   virginica    1    f 6.792308
## 4      setosa    2    f 4.955556
## 5  versicolor    2    f 5.742857
## 6   virginica    2    f 6.755556
## 7      setosa    3    f 4.900000
## 8  versicolor    3    f 5.842857
## 9   virginica    3    f 6.750000
## 10     setosa    1    m 5.157143
## 11 versicolor    1    m 5.966667
## 12  virginica    1    m 6.577778
## 13     setosa    2    m 5.050000
## 14 versicolor    2    m 5.909091
## 15  virginica    2    m 6.250000
## 16     setosa    3    m 4.950000
## 17 versicolor    3    m 5.900000
## 18  virginica    3    m 6.288889
```

Much better! now we have a data frame where the results are in long format, which is easy for us (and R) to understand. Getting this sort of output means its then really easy to plot the results using ggplot(). (Consider also using tapply() to calculate the standard error, and then adding it to the above data.frame, and then plotting).

Missing data

If you have NA's in your data, tapply will throw back an NA. E.g.:

```
##introduce an NA
iris.NA<-iris
iris.NA$Sepal.Length[10]<-NA

tapply(iris.NA$Sepal.Length, list(iris.NA$Species), mean)
```

```
##     setosa versicolor  virginica
##         NA      5.936      6.588
```

This is easily solved, as you can add in the additional arguments taken by the function (in this case mean) after specifying the function:

```
##remove the NA's when calculating the mean
tapply(iris.NA$Sepal.Length, list(iris.NA$Species), mean, na.rm=T)
```

```
##     setosa versicolor  virginica
##   5.008163   5.936000   6.588000
```

lapply()

lapply() is one of the functions I use most in R, in everything from running simulations, to producing plots, to analysing data. As the name suggests lapply() applies a function to a list, but is much more felxible about the functions you can specify. lapply takes the following arguments:

```
lapply(list, function)
```

So, lets put this into practice using the iris data set from aboe. First we need it to be in list form. We will do an analysis by species (as we did above), so we need to break the data down into a list containing a data.frame for each species. The easiest way to do this is using the split() function. Split is the perfect partner for lapply, it splits (duh!) a data.frame into a list based on the information contained in 1 or more columns. So effectively we say we want this data.frame to be seperated into a bunch of smaller data.frames, in our case one data.frame for each species:

```
##this is saying split the data.frame into a list, with each object in the list a smaller data.fra
me indexed by the informaiton in the species column
split.iris<-split(iris, list(iris$Species))
class(split.iris)
```

```
## [1] "list"
```

You can also do this by multiple arguments, say split it by the Species and location (run this and see what you get):

```
## in the same way tapply takes multiple arguments to split the data, so does split:
split(iris, list(iris$Species, iris$location))
```

But we will use the data just split by species.

So now we can use the lapply function. lapply() is quite different in its arguments to tapply. The easiest way to get to grip with this is to briefly understand how lapply() works. In effect is does the same as a for() loop: it will iterate through each object in the list provided (in our case there are 3, a data frame for each species, created with split()) and will apply a specified function.

So, the first argument is the list you want to iterate over, the second is then the function you want to impliment. In the below example, we are going to specify a function, and that function takes the single argument k. k is the object in the list, in our case a data.frame, but it could be a model output or anything.

So we then want to write a function which we want to impiment to every object (data.frame, k) in our list. Let's do something simple:

```
lapply(split.iris, function(k){
  ##we use the return() function to tell lapply what we want to get back, in this case it is the o
nly thing we are calculating, but that isnt true when things get more complicated (see below)
  return(mean(k$Sepal.Width))
})
```

```
## $setosa
## [1] 3.428
##
## $versicolor
## [1] 2.77
##
## $virginica
## [1] 2.974
```

Ok so to read this, for every object in split.iris (which we call "k", and which lapply will look for under the name "k"), calculate the mean of the column called Sepal.Width. The name of the object (k) isnt important, as long as you keep it consistent. It could for example be:

```
lapply(split.iris, function(supercalifragilisticexpialidocious){
  return(mean(supercalifragilisticexpialidocious$Sepal.Width))
})
```

Now this is very basic, and, as you should realise, could more easily be done with tapply().

However, we can make really complex functions to do a huge amount of analyses instead of something simple.

Using lapply() for simulations

So one of the things I use lapply() for the most is running simulations. We are going to do this with a simple simulation of stock prices taken from https://www.r-bloggers.com/stochastic-processes-and-stocks-simulation/ (https://www.r-bloggers.com/stochastic-processes-and-stocks-simulation/).

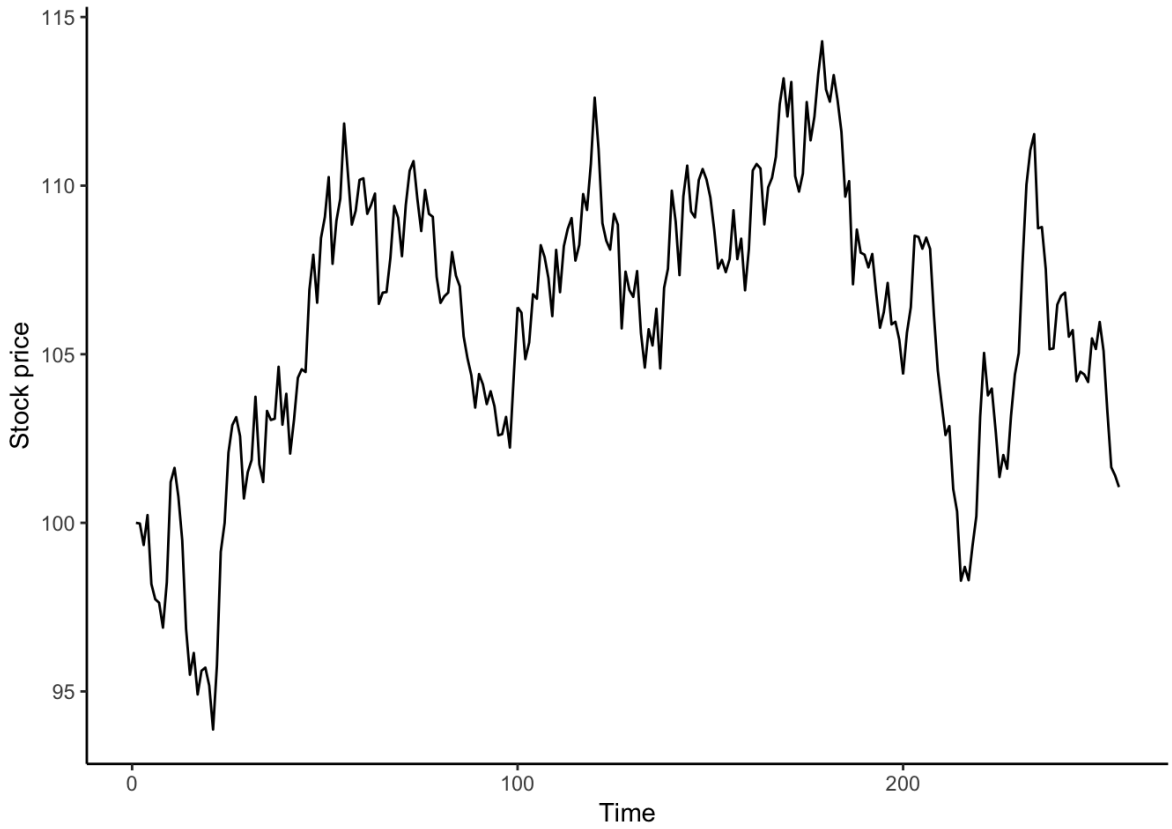
```
##load ggplot to make some plots:
require(ggplot2)
```

```
## Loading required package: ggplot2
```

```
##parameters for the model:
Z <- rnorm(255,0,1)   # Random normally distributed values, mean = 0, stdv = 1
u <- 0.3              # Expected annual return (30%)
sd <- 0.2             # Expected annual standard deviation (20%)
s <- 100              # Starting price
price <- c(s)         # Price vector
a <- 2                # See below
t <- 1:256            # Time. Days to put on the x axis

##the stochastic simulation:
for(i in Z)
{
  S = s + s*(u/255 + sd/sqrt(255)*i)
  price[a] <- S
  s = S
  a = a + 1
}
##bind the data together
dd<-data.frame(t, price)

ggplot(dd, aes(x=t, y=price))+geom_line()+theme_classic()+xlab("Time")+ylab("Stock price")
```



Great, so here we have a simple stochastic simulation. But lets say we want to run this for a range of different input values of u (the expected annual return):

```
# Expected annual return (30%)
u <- seq(0.2, 0.4, length.out=5)
u
```

```
## [1] 0.20 0.25 0.30 0.35 0.40
```

So we have generated 5 values for u, and now we want to run simulations using these using lapply(). The easy way to do this is to first turn u into a list:

```
u.list<-as.list(u)
u.list
```

```
## [[1]]
## [1] 0.2
##
## [[2]]
## [1] 0.25
##
## [[3]]
## [1] 0.3
##
## [[4]]
## [1] 0.35
##
## [[5]]
## [1] 0.4
```

now you can see that each object in the list corresponds to a value of u we want to try in the simulation. Now using lapply, we can iterate over these:

Remember, that the function() bit relates to the objects being iterated through in the list, in this case this is going to be the “u” values.

we are going to save these out as a list call stoc.sims:

```
stoc.sims<-lapply(u.list, function(u){

  ##fixed parameters for the model:
    Z <- rnorm(255,0,1)   # Random normally distributed values, mean = 0, stdv = 1
    sd <- 0.2             # Expected annual standard deviation (20%)
    s <- 100              # Starting price
    price <- c(s)         # Price vector
    a <- 2                # See below
    t <- 1:256            # Time. Days to put on the x axis

  ##we havent specified the u values because these are coming from the list above

  ##the stochastic simulation:
    for(i in Z)
    {
      S = s + s*(u/255 + sd/sqrt(255)*i)
      price[a] <- S
      s = S
      a = a + 1
    }
  ##bind the data together
  dd<-data.frame(t, price)

  ##but we are now doing this for a range of values of u, but we want to know wich simulation rela
tes to which value, so add the u values to the data.frame you are exporting from lapply()
  dd$u<-u

  return(dd)
})
##what does lapply return?
class(stoc.sims)
```

```
## [1] "list"
```

```
##and how many objest are in it? (i.e. how many u values were there...)
length(stoc.sims)
```

```
## [1] 5
```

```
##have a look at the first object in the list:
head(stoc.sims[[1]])
```

```
##   t    price   u
## 1 1 100.0000 0.2
## 2 2 100.1194 0.2
## 3 3 100.0526 0.2
## 4 4 100.9790 0.2
## 5 5 102.8860 0.2
## 6 6 102.9179 0.2
```

Great! that seems to have worked. You can see we have 5 objects in the list (one for each value of u). We have the u values recorded in the data frame, as well as the price and time.

Now we want to plot them. To do this with ggplot we need them in the form of a data.frame, but currently they are a list. The data.table package has a fantastic function for this which binds lists together into a data.table (equivelant to a data.frame, but deals better with big data sets):

```
require(data.table)
```

```
## Loading required package: data.table
```

```
##
## Attaching package: 'data.table'
```

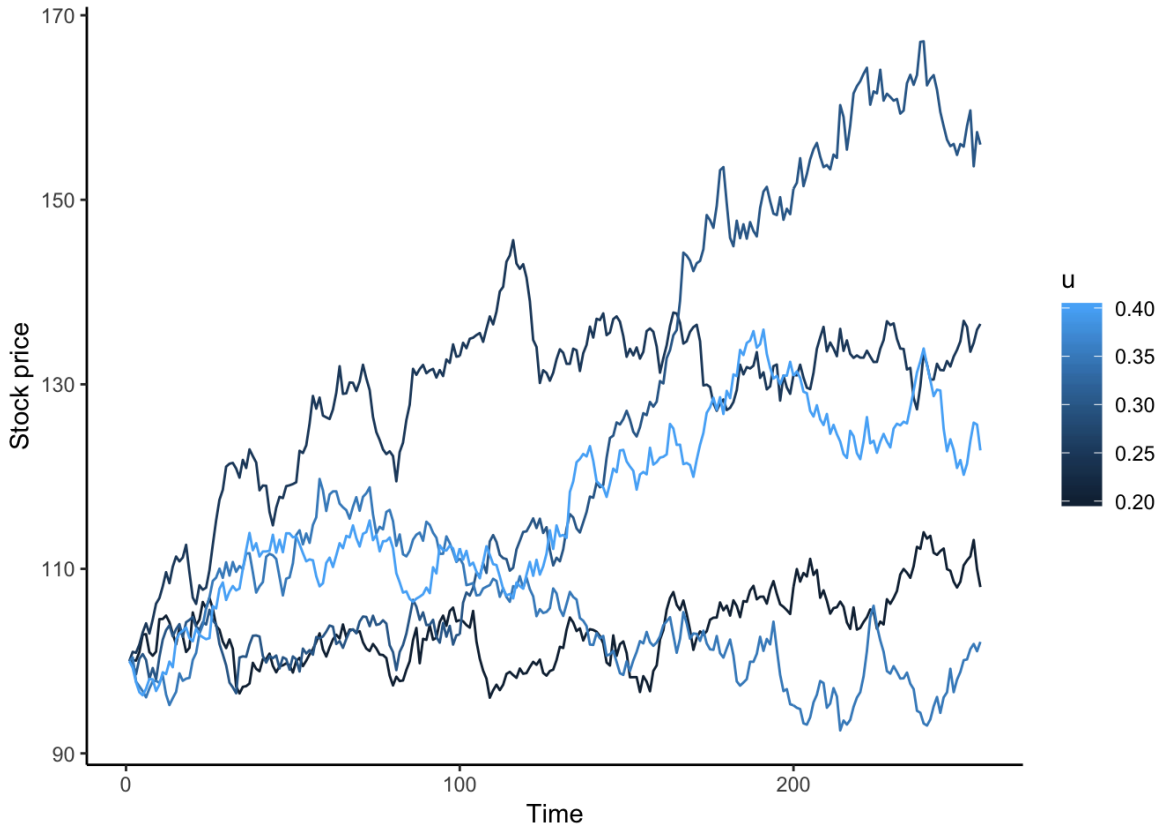
```
## The following objects are masked from 'package:reshape2':
##
##     dcast, melt
```

```
sims<-rbindlist(stoc.sims)
sims
```

```
##         t    price   u
##    1:   1 100.0000 0.2
##    2:   2 100.1194 0.2
##    3:   3 100.0526 0.2
##    4:   4 100.9790 0.2
##    5:   5 102.8860 0.2
##   ---
## 1276: 252 121.3884 0.4
## 1277: 253 123.8594 0.4
## 1278: 254 125.8446 0.4
## 1279: 255 125.6321 0.4
## 1280: 256 122.8410 0.4
```

Now we can plot them:

```
  ggplot(sims, aes(x=t, y=price, group=u, col=u))+geom_line()+theme_classic()+xlab("Time")+ylab("S
tock price")
```



Awesome. So we are going to stick with this example and expand the range of the simulations a little. Above we considered altering a single value, but as you will see the more complex and large the number of simulations gets, the more lapply() can help. Lets look at 2 parameters, and say we want to test every combination of these two parameters (which is a very common approach to take):

```
# Expected annual return (30%)
u <- seq(0.2, 0.4, length.out=5)
u
```

```
## [1] 0.20 0.25 0.30 0.35 0.40
```

```
##Expected annual standard deviation (20%)
sd <- c(0.1, 0.3)
sd
```

```
## [1] 0.1 0.3
```

great, so we have two vectors of the different values we want to iterate over. Next, lets build a data.frame containing every combination of the two vectors, using the expand.grid function:

```
all.parms<-expand.grid(u, sd)
names(all.parms)<-c("u", "sd")
dim(all.parms)
```

```
## [1] 10  2
```

```
all.parms
```

```
##       u  sd
## 1  0.20 0.1
## 2  0.25 0.1
## 3  0.30 0.1
## 4  0.35 0.1
## 5  0.40 0.1
## 6  0.20 0.3
## 7  0.25 0.3
## 8  0.30 0.3
## 9  0.35 0.3
## 10 0.40 0.3
```

So you can see that we now have a data.frame comprising of all of the combinations we want to test. Now, lets convert that into a list and plug it into lapply. We can do this by using the split function, and telling it to split by every row:

```
split.parms<-split(all.parms, 1:nrow(all.parms))
head(split.parms)
```

```
## $`1`
##     u  sd
## 1 0.2 0.1
##
## $`2`
##      u  sd
## 2 0.25 0.1
##
## $`3`
##     u  sd
## 3 0.3 0.1
##
## $`4`
##      u  sd
## 4 0.35 0.1
##
## $`5`
##     u  sd
## 5 0.4 0.1
##
## $`6`
##     u  sd
## 6 0.2 0.3
```

So you can see that each object in the list is a data frame containing 1 row with the u and sd values in it.

Now lets put that into lapply. Things are going to change a little now however. Before we had a single value in each object in the list, now we have a data frame, so we need to be careful to specify that in the lapply function:

```
stoc.sims<-lapply(split.parms, function(parameters){

  ##remember "parameters" as we have called it is a data frame, so we need to pull out the u and s
d values:
  u<-parameters$u
  sd<-parameters$sd

  ##############################################################################################
############
  ###all of this remains unchanged
  ##fixed parameters for the model:
    Z <- rnorm(255,0,1)   # Random normally distributed values, mean = 0, stdv = 1
    s <- 100              # Starting price
    price <- c(s)         # Price vector
    a <- 2                # See below
    t <- 1:256            # Time. Days to put on the x axis

  ##we havent specified the u values because these are coming from the list above

  ##the stochastic simulation:
    for(i in Z)
    {
      S = s + s*(u/255 + sd/sqrt(255)*i)
      price[a] <- S
      s = S
      a = a + 1
    }
  ##this is the data frame we made earlier
  dd<-data.frame(t, price)
  ##############################################################################################
############

  ##but we now need to save both the u and sd values out:
  dd$u<-u
  dd$sd<-sd
  return(dd)
})

##how long is the list?
length(stoc.sims)
```

```
## [1] 10
```

```
##have a look at the first object in the list:
head(stoc.sims[[1]])
```

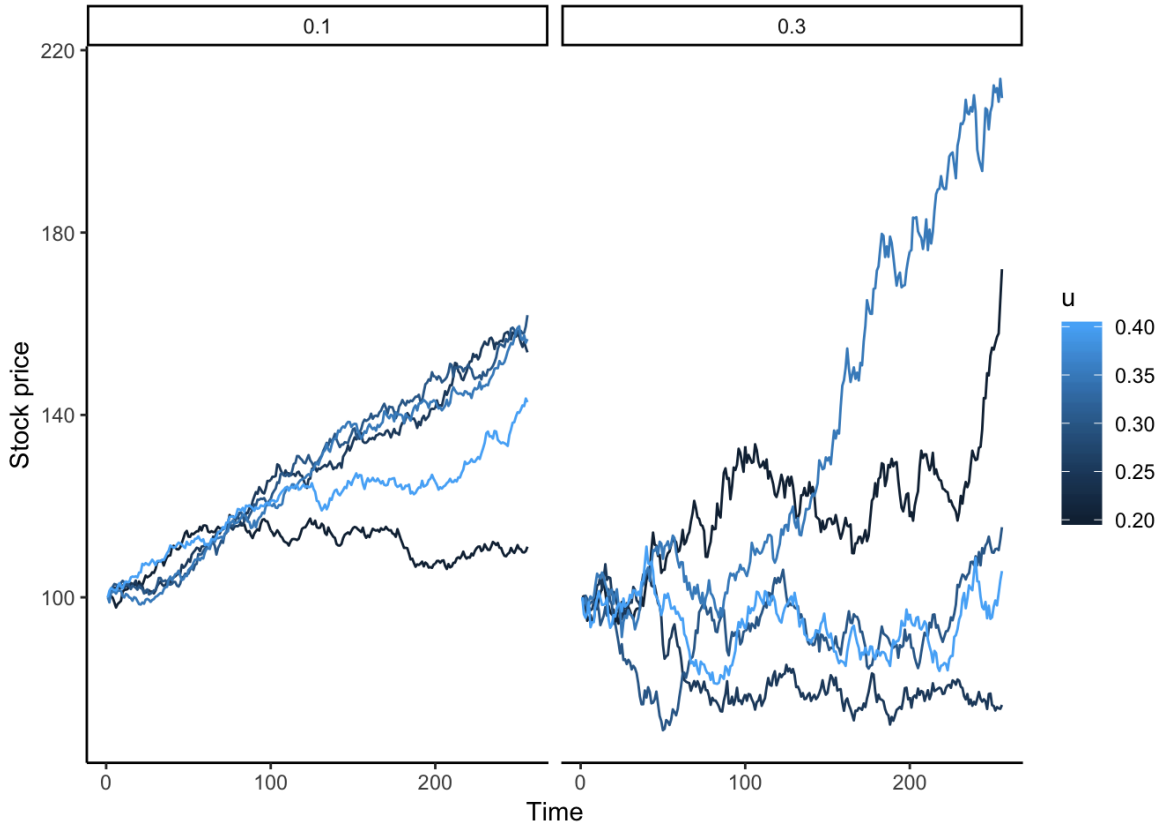
```
##   t     price   u  sd
## 1 1 100.00000 0.2 0.1
## 2 2 100.38046 0.2 0.1
## 3 3 100.27783 0.2 0.1
## 4 4  99.88901 0.2 0.1
## 5 5  98.88677 0.2 0.1
## 6 6  97.77075 0.2 0.1
```

And then we can do as we did before to plot it out:

```
sims<-rbindlist(stoc.sims)
sims
```

```
##         t     price   u  sd
##    1:   1 100.00000 0.2 0.1
##    2:   2 100.38046 0.2 0.1
##    3:   3 100.27783 0.2 0.1
##    4:   4  99.88901 0.2 0.1
##    5:   5  98.88677 0.2 0.1
##   ---
## 2556: 252  99.30698 0.4 0.3
## 2557: 253  98.23880 0.4 0.3
## 2558: 254 100.91738 0.4 0.3
## 2559: 255 103.67036 0.4 0.3
## 2560: 256 105.76198 0.4 0.3
```

```
##note the slight change in the "group" section:
ggplot(sims, aes(x=t, y=price, group=interaction(u, sd), col=u))+geom_line()+theme_classic()+facet
_grid(.~sd)+xlab("Time")+ylab("Stock price")
```



Now I think it is fairly easy to see how doing lots of combinations of parameters can easily lead to a large number of simulations (and actually here we are just doing 1 stochastic simulation per set of parameters, which is weird). Here is an example with 1000 reps of each of the u and sd values we used above:

```
# Expected annual return (30%)
u <- seq(0.2, 0.4, length.out=5)
u
```

```
## [1] 0.20 0.25 0.30 0.35 0.40
```

```
##Expected annual standard deviation (20%)
sd <- seq(0.1, 0.3, length.out=5)
sd
```

```
## [1] 0.10 0.15 0.20 0.25 0.30
```

```
##number of replicates
nreps<-c(1:1000)

all.parms<-expand.grid(u, sd, nreps)
names(all.parms)<-c("u", "sd", "rep")
dim(all.parms)
```

```
## [1] 25000     3
```

You can now see that we have 25000 simulations to run, and even if they only take 0.1 seconds each, thats still 4 minutes to run. And these are simple simulations. Fortunately we can solve this (and other processing heavy tasks) by making our code parallel

mclapply()

Parallel computing is the norm. Just not in R. Basically, R was developed when computers had single processors, and so could only do one calculation at a time. Now we tend to have lots, but most of the time in R we dont take advantage of this fact. mclapply() and other parallel functions make this possible, by shipping off calculations to different processors, and then combining all the results at the end. On a mac this is all dealt with for you, on Windows….well I have no idea, sorry!

mclapply() is a multicore version of lapply(). It basically looks at your computer to see how many cores you have, divides the list you are passing to lapply evenly between those cores (e.g. if you have 5 cores, and a list of 10 objects, it will pass 2 objects to each core), and then on each core simultaneously it works through the smaller lists and applies the function specified.

Implimenting it is ridiculously simple. Here is our example from above:

```
##package for mclapply()
require(parallel)
```

```
## Loading required package: parallel
```

```
##mclapply implimentation. just change lapply to mclapply!
stoc.sims<-mclapply(split.parms, function(parameters){

  ##remember "parameters" as we have called it is a data frame, so we need to pull out the u and s
d values:
  u<-parameters$u
  sd<-parameters$sd

  ##############################################################################################
############
  ###all of this remains unchanged
  ##fixed parameters for the model:
    Z <- rnorm(255,0,1)   # Random normally distributed values, mean = 0, stdv = 1
    s <- 100              # Starting price
    price <- c(s)         # Price vector
    a <- 2                # See below
    t <- 1:256            # Time. Days to put on the x axis

  ##we havent specified the u values because these are coming from the list above

  ##the stochastic simulation:
    for(i in Z)
    {
      S = s + s*(u/255 + sd/sqrt(255)*i)
      price[a] <- S
      s = S
      a = a + 1
    }
  ##this is the data frame we made earlier
  dd<-data.frame(t, price)
  ##############################################################################################
############

  ##but we now need to save both the u and sd values out:
  dd$u<-u
  dd$sd<-sd
  return(dd)
})
```

Doing this in parallel is as easy as adding mc to the beginning of our code!

of course in this example it doesnt make much difference, but lets try that big simulaion of 2500 parameter values. We arent going to analyse these results, just look at the time it takes for lapply and mclapply to do their thing.

```
##############################################################################
##############################################################################
##we are going to wrap all of this model stuff up into a function. This function is just going to
 take the data.frame that contains our parameter values, and then extract them (as above) and do t
he simulation, and return our data.frame of results:
model<-function(x){
  ##the new parameters
  u<-x$u
  sd<-x$sd
  ###all of this remains unchanged
  ##fixed parameters for the model:
    Z <- rnorm(255,0,1)   # Random normally distributed values, mean = 0, stdv = 1
    s <- 100              # Starting price
    price <- c(s)         # Price vector
    a <- 2                # See below
    t <- 1:256            # Time. Days to put on the x axis

  ##we havent specified the u values because these are coming from the list above

  ##the stochastic simulation:
    for(i in Z)
    {
      S = s + s*(u/255 + sd/sqrt(255)*i)
      price[a] <- S
      s = S
      a = a + 1
    }
  ##this is the data frame we made earlier
  dd<-data.frame(t, price)
  return(dd)
}
##############################################################################
##############################################################################
```

Ok, now lets look at how long they take to run:

```
split.all.parms<-split(all.parms, 1:nrow(all.parms))

system.time(lapply(split.all.parms, function(parameters){
  return(model(parameters))
}))
```

```
##    user  system elapsed
##   6.176   0.070   6.279
```

```
system.time(mclapply(split.all.parms, function(parameters){
  return(model(parameters))
}))
```

```
##    user  system elapsed
##   0.216   0.109   3.766
```

so you can clealrly see that here mclapply() took 1/2 the time that lapply did, with basically no changes to my code. This gets exponentially more obvious as the simulations/data analysis/functions you are carrying out get slower.

Restricting the number of cores

If you just run mclapply() as above it will use as many cores (processors) as you have availble on your computer. If you are running big, RAM hungry simulations then this can cause your computer to slow down to the point it is unusable for anything else. However its easy to restrict the number of cores mclapply uses. I tend to use the number of cores my computer has -1. You can find the number of cores your computer has by doing:

```
##detectCores() is from the parallel library
detectCores()
```

```
## [1] 8
```

Then you can simply specify the number of cores mclapply is using by doing:

```
mclapply(split.all.parms, function(parameters){
  return(model(parameters))
}, mc.cores = 4)

##or
mclapply(split.all.parms, function(parameters){
  return(model(parameters))
}, mc.cores = detectCores()-1)
```

sapply()

sapply is a simplified version of lapply. in effect it does what rbindlist does when we used it above, to return a simplified version of the output of lapply. To give a really simple example, lapply returns a list, sapply returns a vector:

```
lapply(split.iris, function(k){
  mean(k$Sepal.Width)
})
```

```
## $setosa
## [1] 3.428
##
## $versicolor
## [1] 2.77
##
## $virginica
## [1] 2.974
```

```
sapply(split.iris, function(k){
  mean(k$Sepal.Width)
})
```

```
##     setosa versicolor  virginica
##      3.428      2.770      2.974
```

However it won't always return a vector, for example you could do :

```
sapply(split.iris, function(k){
  k$Length.Width<-k$Sepal.Length+k$Sepal.Width
})
```

```
##       setosa versicolor virginica
##  [1,]    8.6       10.2       9.6
##  [2,]    7.9        9.6       8.5
##  [3,]    7.9       10.0      10.1
##  [4,]    7.7        7.8       9.2
##  [5,]    8.6        9.3       9.5
##  [6,]    9.3        8.5      10.6
##  [7,]    8.0        9.6       7.4
##  [8,]    8.4        7.3      10.2
##  [9,]    7.3        9.5       9.2
## [10,]    8.0        7.9      10.8
## [11,]    9.1        7.0       9.7
## [12,]    8.2        8.9       9.1
## [13,]    7.8        8.2       9.8
## [14,]    7.3        9.0       8.2
## [15,]    9.8        8.5       8.6
## [16,]   10.1        9.8       9.6
## [17,]    9.3        8.6       9.5
## [18,]    8.6        8.5      11.5
## [19,]    9.5        8.4      10.3
## [20,]    8.9        8.1       8.2
## [21,]    8.8        9.1      10.1
## [22,]    8.8        8.9       8.4
## [23,]    8.2        8.8      10.5
## [24,]    8.4        8.9       9.0
## [25,]    8.2        9.3      10.0
## [26,]    8.0        9.6      10.4
## [27,]    8.4        9.6       9.0
## [28,]    8.7        9.7       9.1
## [29,]    8.6        8.9       9.2
## [30,]    7.9        8.3      10.2
## [31,]    7.9        7.9      10.2
## [32,]    8.8        7.9      11.7
## [33,]    9.3        8.5       9.2
## [34,]    9.7        8.7       9.1
## [35,]    8.0        8.4       8.7
## [36,]    8.2        9.4      10.7
## [37,]    9.0        9.8       9.7
## [38,]    8.5        8.6       9.5
## [39,]    7.4        8.6       9.0
## [40,]    8.5        8.0      10.0
## [41,]    8.5        8.1       9.8
## [42,]    6.8        9.1      10.0
## [43,]    7.6        8.4       8.5
## [44,]    8.5        7.3      10.0
## [45,]    8.9        8.3      10.0
## [46,]    7.8        8.7       9.7
## [47,]    8.9        8.6       8.8
## [48,]    7.8        9.1       9.5
## [49,]    9.0        7.6       9.6
## [50,]    8.3        8.5       8.9
```

And sapply will return a useful matrix with the results in it.

vapply()

in the words of the R help file:

“vapply is similar to sapply, but has a pre-specified type of return value, so it can be safer (and sometimes faster) to use.”

In effect this means that you basically use it as sapply, but it will only return a vector.

mapply()

mapply() works a little differently to the previous apply functions, for a start the arguments given to the mapply() are given in a different order:

```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
       USE.NAMES = TRUE)
```

AS you can see first off we defin the function that is going to be applied, and then we have this "…" section which allows us to define what the function is going to be applied over.

For a really simple example, we could use mapply() to create a matrix of repeate values using the rep() function:

```
##normally rep is used like this:
rep(1:4, 2)
```

```
## [1] 1 2 3 4 1 2 3 4
```

```
##which as you can see means replicate the numbers 1 to 4 twice

##we can do the same thing with mapply, but note that this now comes out as a matrix, rather than
 a long vector, which is much more useful:
mapply(rep, 1:4, 2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    1    2    3    4
```

But where mapply becomes pretty neat is when you iterate over two lists or vectors of varying lengths. What do I mean by that? Well in the above example the 3rd argument had length=1, but if we change that, say by giving it another vector of values, you can see that mapply does something a bit creative:

```
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

Now mapply has gone through both vectors an value at a time and used those values in the rep() function. This is what R's help file unhelpfully means when it says: "mapply is a multivariate version of sapply. mapply applies FUN to the first elements of each … argument, the second elements, the third elements, and so on."

Slightly more helpfully, the r-bloggers described it as:

"Its purpose is to be able to vectorize arguments to a function that is not usually accepting vectors as arguments. In short, mapply applies a Function to Multiple List or multiple Vector Arguments.""

If am I honest, I haven't ever used the mapply function in anger, so I will leave it up to you to find exciting uses for it!

rapply()

recursive apply is a function I hadn't heard of until I wrote this tutorial, but, as the name unimaginatively suggests, it recursively applies a function to a list or vector.

The easiest way to understand this is to build a list which contains sublists (lists within lists). This is a good thing to know about anyway in R, especially if you are dealing with complex data. Here is a simple example:

```
x=list(1,list(2,3),4)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [[2]][[1]]
## [1] 2
##
## [[2]][[2]]
## [1] 3
##
##
## [[3]]
## [1] 4
```

```
str(x)
```

```
## List of 3
##  $ : num 1
##  $ :List of 2
##   ..$ : num 2
##   ..$ : num 3
##  $ : num 4
```

Ok, so x is our list. the first object in the list x[[1]] is the number 1:

```
##first object in the list x
x[[1]]
```

```
## [1] 1
```

The second object in the list x[[2]] is a sub-list containing 2 objects:

```
##2nd object in the list x
x[[2]]
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [1] 3
```

the first of these objects, is the number 2, the second the number 3:

```
##second object in the first list (x), 1st object in that sub list
x[[2]][[1]]
```

```
## [1] 2
```

```
##second object in the first list (x), 2nd object in that sub list
x[[2]][[2]]
```

```
## [1] 3
```

The final object in the x list is the number 4. Easy.

Ok, so rapply will iterate over the objects in the list x, and also the sublist. here, we take the value and simply add 1 to it:

```
rapply(x, function(y)y+1)
```

```
## [1] 2 3 4 5
```

In its basic form (above) it will return a vector of values, but (more usefully) it can also return a list in the same structure as the original:

```
rapply(x, function(y)y+1, how="list")
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [[2]][[1]]
## [1] 3
##
## [[2]][[2]]
## [1] 4
##
##
## [[3]]
## [1] 5
```

You can also use this to replace items in a lists of different classes with a new value. The best example I can think of would be say to replace NA's in the lists (and sub lists) with 0's:

```
y <- list(1,list(2,NA),4)
y
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [[2]][[1]]
## [1] 2
##
## [[2]][[2]]
## [1] NA
##
##
## [[3]]
## [1] 4
```

```
rapply(y, function(y)y+1, class="numeric", how="list", deflt=0)
```

```
## [[1]]
## [1] 2
##
## [[2]]
## [[2]][[1]]
## [1] 3
##
## [[2]][[2]]
## [1] 0
##
##
## [[3]]
## [1] 5
```

```
rapply(y, function(y)y+1, class="numeric",  deflt=0)
```

```
## [1] 2 3 0 5
```

apply()

The big daddy of all of the apply functions is a hard one to wrap your head around (at least for me) but is really useful for big, complex data sets.

Matrices and arrays

Apply works with either a matrix, or array (which is just a 3D matrix). Lets just make sure we are all on the same page first with those concepts. First an example of a matrix:

```
##an example matrix, with 3 columns and 3 rows
matrix(1:9, nrow=3, ncol=3)
```

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

The main difference between a matrix and a data.frame (since you are probably wondering) is that matrices have to contain all the same kind of data (e.g. numeric) whilst data.frames can have factors, numerics, characters all mixed in.

An array is just a collection of matrices. The easiest way to imagine it is that you have a matrix, and then another one behind it, and potentially another behind that etc.

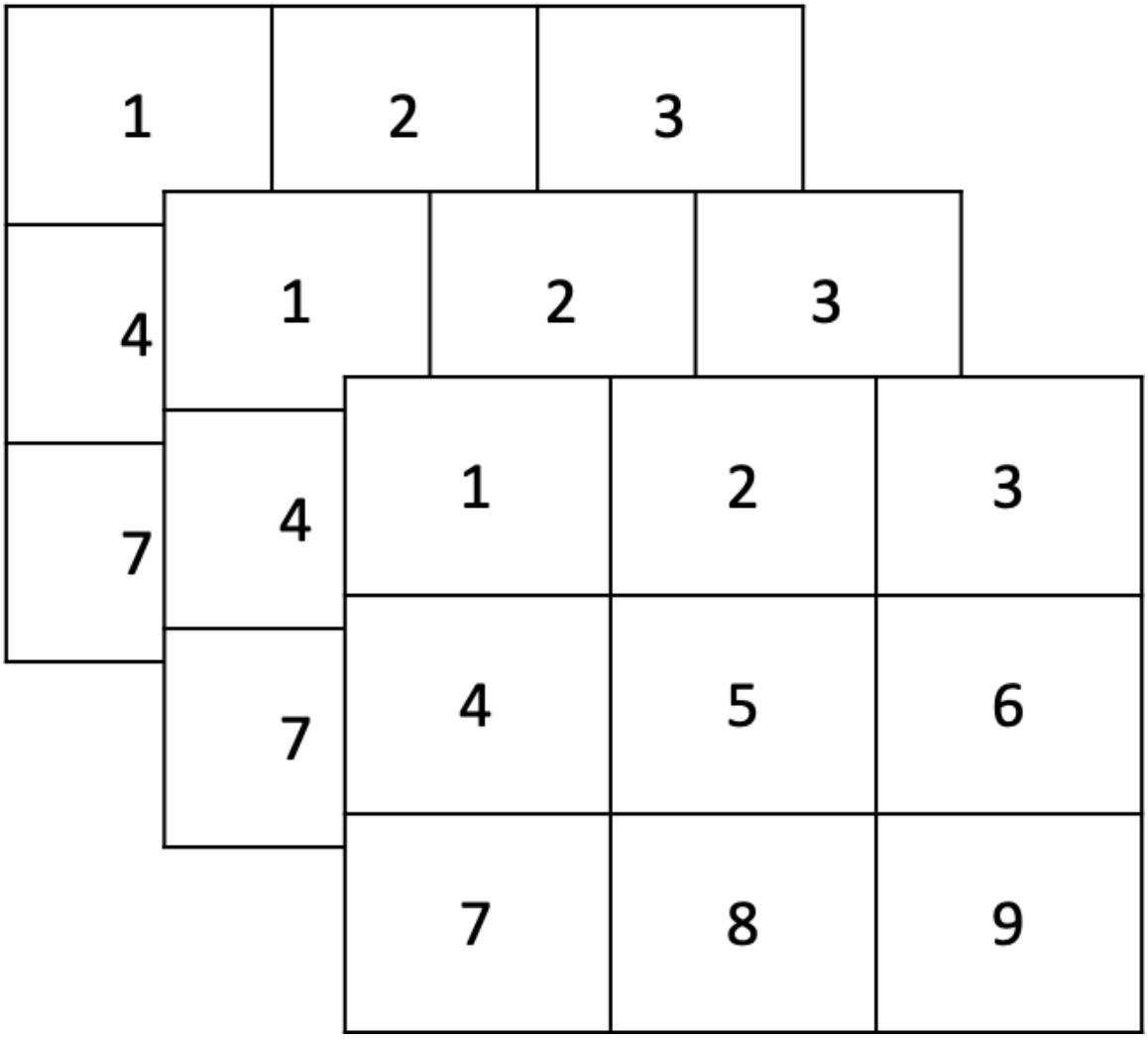


Fig. 1 - An array (sort of)

However, as with so many things this easy visualisation isnt actually whats going on. In actuality the data is sorted in an n dimensional data object, which allows you to access any rows or columns combination from any dimension.

But lets start with a simple array:

```
thisisanarray <- array(1:24, dim=c(3,4,2))
thisisanarray
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

So you can see that specifying the dimensions using dim=c() shapes the matrices and the array. We specified there were going to be three rows, four columns, and two levels to the array by writing dim=c(3,4,2).

Alternatively we could do

```
array(1:24, dim=c(3,2,4))
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
##
## , , 3
##
##      [,1] [,2]
## [1,]   13   16
## [2,]   14   17
## [3,]   15   18
##
## , , 4
##
##      [,1] [,2]
## [1,]   19   22
## [2,]   20   23
## [3,]   21   24
```

And have four smaller matrices of three rows and two columns.

But because arrays are n dimensional, we can also do:

```
a.sill.array <- array(1:24, dim=c(3,2,4,2))
dim(a.sill.array)
```

```
## [1] 3 2 4 2
```

```
a.sill.array
```

```
## , , 1, 1
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2, 1
##
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
##
## , , 3, 1
##
##      [,1] [,2]
## [1,]   13   16
## [2,]   14   17
## [3,]   15   18
##
## , , 4, 1
##
##      [,1] [,2]
## [1,]   19   22
## [2,]   20   23
## [3,]   21   24
##
## , , 1, 2
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2, 2
##
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
##
## , , 3, 2
##
##      [,1] [,2]
## [1,]   13   16
## [2,]   14   17
## [3,]   15   18
##
## , , 4, 2
##
##      [,1] [,2]
## [1,]   19   22
## [2,]   20   23
## [3,]   21   24
```

using apply() with an array

So we have build an array, how does apply work?

So remember our array has 3 rows, 4 columns, and 2… other dimensions,

```
dim(thisisanarray)
```

```
## [1] 3 4 2
```

if we look at the help file for apply() you will see that the arguments taken by apply() are:

```
apply(X, MARGIN, FUN, ...)
```

where:

X is an array, including a matrix. (good to note here that a matrix is a 2d array) MARGIN is a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns. This I find utterly confusing, but a better way to think of this is to say if you specify MARGIN=1 then it will apply by row, if MARGIN=2 then it will apply by column, etc FUN is the function to be applied

Ok, so we have our array (helpfully titled “thisisanarray”), so lets apply a function to it, using MARGIN=1

```
apply(thisisanarray, MARGIN=c(1), function(x)x^2)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    9
## [2,]   16   25   36
## [3,]   49   64   81
## [4,]  100  121  144
## [5,]  169  196  225
## [6,]  256  289  324
## [7,]  361  400  441
## [8,]  484  529  576
```

So by the above rational what have we done? well, take a look at the first column of the output first, and our original array along side one another:

```
apply(thisisanarray, MARGIN=c(1), function(x)x^2)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    9
## [2,]   16   25   36
## [3,]   49   64   81
## [4,]  100  121  144
## [5,]  169  196  225
## [6,]  256  289  324
## [7,]  361  400  441
## [8,]  484  529  576
```

```
thisisanarray
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

you can see that, as suggested above, we have applied the function BY ROW. So, the first row of the first matrix in thisisanarray gets passed to the function, and then poushed into the first column of the output. then the first row of the second matrix goes to the function, and becomes the second half of the 1st column of the output. And exactly the same happens to the second row of the first matrix, which becomes the first half of the 2nd column in the output, and so on. Or, schematically:

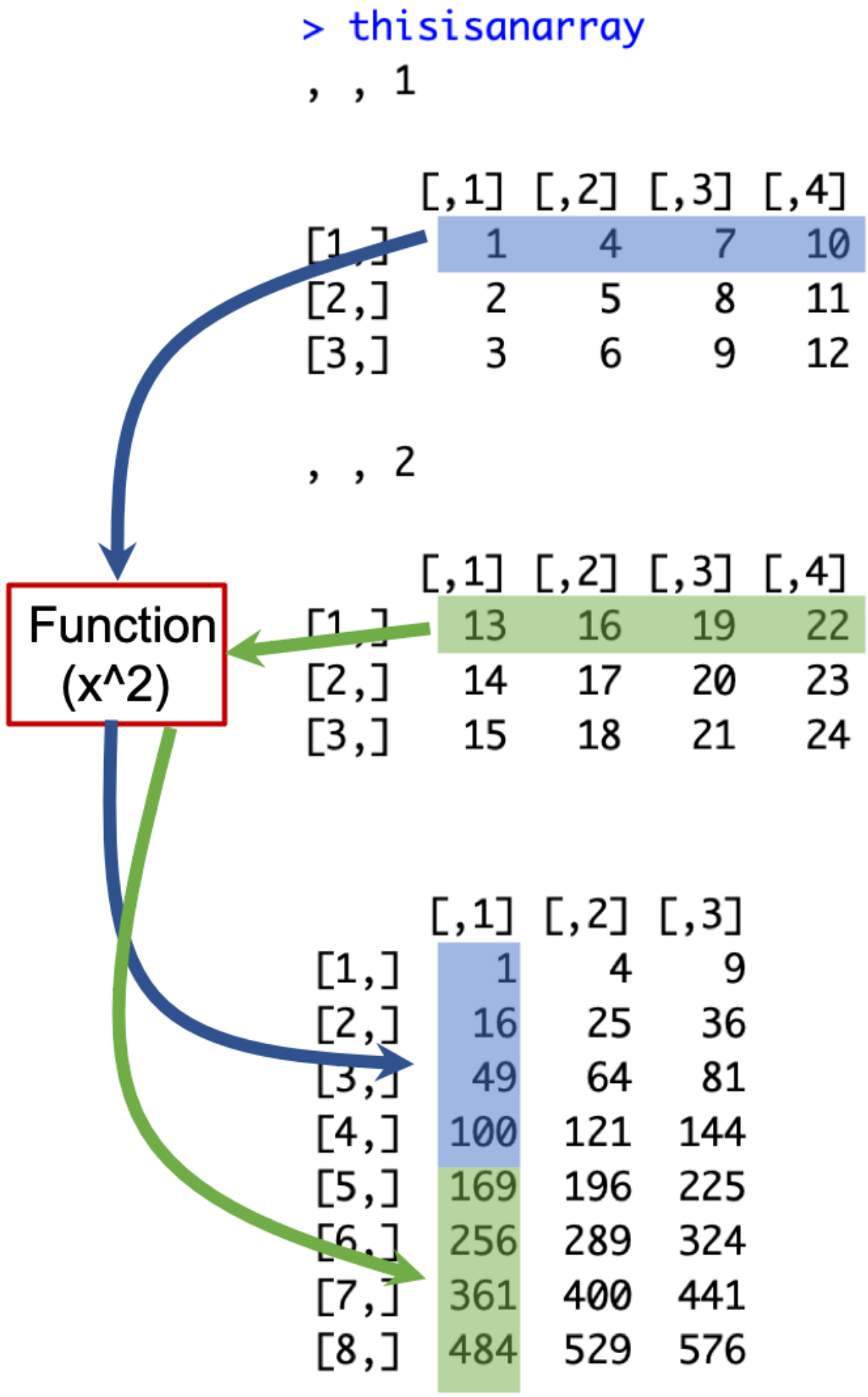


Fig. 2 - how apply works

Follow? No, me neiter. but there you go. Here is the first example reversed, i.e. going by column, which might make things a little clearer:

```
apply(thisisanarray, MARGIN=c(2), function(x)x^2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1   16   49  100
## [2,]    4   25   64  121
## [3,]    9   36   81  144
## [4,]  169  256  361  484
## [5,]  196  289  400  529
## [6,]  225  324  441  576
```

```
thisisanarray
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

Whereby you are working through “thisisanarray” by column now (so the first column of the first matrix is now the first half of the first column of the output, the 1st column of the second matrix is now the second half of the first column of the output and so on).

Finally, remember that our array has the following dimensions:

```
dim(thisisanarray)
```

```
## [1] 3 4 2
```

i.e. it has 3 rows, 4 columns, and 2 matrices, so we can also iterate over it by doing:

```
apply(thisisanarray, MARGIN=c(3), function(x)x^2)
```

```
##       [,1] [,2]
##  [1,]    1  169
##  [2,]    4  196
##  [3,]    9  225
##  [4,]   16  256
##  [5,]   25  289
##  [6,]   36  324
##  [7,]   49  361
##  [8,]   64  400
##  [9,]   81  441
## [10,]  100  484
## [11,]  121  529
## [12,]  144  576
```

Right. Deeply confusing. So this is just an example of applying a function once to each item, either by row, by column, or by matrix. However, we can also specify apply to do the following:

```
apply(thisisanarray, MARGIN=c(1,2,3), function(x)x^2)
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1   16   49  100
## [2,]    4   25   64  121
## [3,]    9   36   81  144
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]  169  256  361  484
## [2,]  196  289  400  529
## [3,]  225  324  441  576
```

```
thisisanarray
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

Now the function is being applied by row, and then column, and then matrix. In this instance we now get back our original array shape, but with the function applied such that all the information is squared. If our array had four dimensions, then we could specify:

```
apply(thisisanarray, MARGIN=c(1,2,3,4), function(x)x^2)
```

and we would get our original shape back again.

Apply is not an easy function to get your head around, at least for me, but if you are working with matrices (e.g. size spectra models) then it is invaluable.

Competition!

As is (very recent) tradition we will finish up with a little challenge to see if you can apply (hahahahah) what you have learned.

Load in the whale data and have a look at it. You can see there are 4 columns, the first specifys the year a whale was caught in, the second the species, the third a code to dermine the units used to record the length of the individual (4th column).

Each row is an individual whale.

The Length codes are as follows:

length.code==1 -> cm

length.code==2 -> meters

length.code==3 -> feet

So using the apply functions I want you to sort all the data out so it is comparable. then plot (using ggplot) the trends in the mean body size of each species of whale over the time period sampled. Include as much information in the plots as you can!

Prizes will be based on neatness/conciseness of code, and possibly also prettiness of the plots.

Have at it!