DECISION MODELLING FOR HEALTH ECONOMIC EVALUATION

Model Efficiency: Vectorised approach example

Supplementary file for Advanced Course

Jack Williams & Nichola Naylor

2022 Course

Simple example of vectorisation

Let us start by describing two functions - one which uses a loop, and appends to existing data, and the other which performs a function once using vectorisation. The purpose of the code is to compare two alternatives and check their speed. *Note the process used here, involving the function `system.time()` to highlight running time of functions, can also be used to compare different model running times in general, which can be handy for figuring out run times of certain sections of code to see where optimisation would be most beneficial!.*

Both functions involve taking random draws from a beta distribution. The functions requires three arguments: the number of samples to be drawn from the distribution, and the alpha and beta parameters that provide the shape of the underlying beta distribution.

The first function uses a loop. The loop runs from 1 through to the total number of samples For each loop, a single value is drawn from the beta distribution, and then saved in the results matrix.

```r
function1 <- function(sims, alpha, beta){
  results <- matrix(0, nrow = sims, ncol = 1)
  for(i in 1:sims){
    results[i,1] <- rbeta(n = 1, alpha, beta)
  }
  return(results)
}
```

The second function instead runs all samples at once (as n is equal to the number of sims in the beta distribution), and then these are saved in the matrix.

```
function2 <- function(sims, alpha, beta){
  results <- matrix(0, nrow = sims, ncol = 1)
  results[,1] <- rbeta(n = sims, alpha, beta)
  return(results)
}
```

Now that we have coded the two functions, we can check that the output is comparable, in terms of formatting (the exact numbers will differ as the samples are probabilistic). Here are 6 samples:

```
function1(6, 10, 15)
```

```
##           [,1]
## [1,] 0.3639678
## [2,] 0.4618752
## [3,] 0.3140658
## [4,] 0.4131724
## [5,] 0.4217956
## [6,] 0.2888707
```

```
function2(6, 10, 15)
```

```
##           [,1]
## [1,] 0.3751068
## [2,] 0.5042826
## [3,] 0.3346147
## [4,] 0.2955247
## [5,] 0.4390096
## [6,] 0.4478891
```

Finally, we can run one million samples using each function, and see the difference in time between using a loop and saving each output individually (loop), and running all the samples at once and saving them (vector approach). The time displayed is in seconds.

```
a <- system.time(function1(1000000, 10, 15))
b <- system.time(function2(1000000, 10, 15))

a
```

```
##    user  system elapsed
##    1.58    0.01    1.61
```

```
b
```

```
##    user  system elapsed
##    0.14    0.00    0.14
```

```
a[3]/b[3] ## vectorisation ~10x faster in terms of user time
```

```
## elapsed
##    11.5
```

From `?proc.time` *information (which is used in* `system.time()`*) - "The 'user time' is the CPU time charged for the execution of user instructions of the calling process. The 'system time' is the CPU time charged for execution by the system on behalf of the calling process. . . and the third entry is the 'real' elapsed time since the process was started"*

Whilst this is a very simplistic example, it is worth thinking about when considering how data is assigned within a model function, if that function will be run many times in probabilistic analyses (such as PSA, or EVPPI simulations - see Module 4 exercise materials).

Example of vectorisation in transition probabilities

Here we use the model created in Module 1 as the foundation for our example. The code to produce the transition probability matrices (one for each model cycle) in the course uses a loop. Whilst the loop is an easy way to code this, there are quicker ways for those who want to learn about more efficient ways to code in R.

First lets take a look at the loop approach:

```
for (i in 1:cycles) {
  ## tranisitions out of P-THR
  tm.SP0["P-THR","Death",i] <- tp.PTHR2dead
  tm.SP0["P-THR","successP-THR",i] <- 1 - tp.PTHR2dead
  ## transitions out of success-P-THR
  tm.SP0["successP-THR","R-THR",i] <- revision.risk.sp0[i]
  tm.SP0["successP-THR","Death",i] <- death.risk[i,col.key]
  tm.SP0["successP-THR","successP-THR",i] <- 1-revision.risk.sp0[i] - death.risk[i,col.key]
  ## transitions out of R-THR
  tm.SP0["R-THR","Death",i] <- tp.RTHR2dead + death.risk[i,col.key]
  tm.SP0["R-THR","successR-THR",i] <- 1 - tp.RTHR2dead - death.risk[i,col.key]
  ## transitions out of success-THR
  tm.SP0["successR-THR","R-THR",i] <- tp.rrr
  tm.SP0["successR-THR","Death",i] <- death.risk[i,col.key]
  tm.SP0["successR-THR","successR-THR",i] <- 1 - tp.rrr - death.risk[i,col.key]
  ## no transitions out of death
  tm.SP0["Death","Death",i] <- 1
}
```

Alternatively there is a vectorised approach that can be taken, which is shown below. Here, where transition probabilities are constant, R will apply the same value across all specified positions in the array. Where values are time-dependent, since the length of the vector is equal to the number of cycles (and therefore also the length third dimension for the array), the first position will be filled with the first value, and so on. . .

This can be written as such:

```
## tranisitions out of P-THR
tm.SP0["P-THR","Death",] <- tp.PTHR2dead
tm.SP0["P-THR","successP-THR",] <- 1 - tp.PTHR2dead ## they go into the success THR state
## transitions out of success-P-THR
```

```
tm.SP0["successP-THR","R-THR",] <- revision.risk.sp0
tm.SP0["successP-THR","Death",] <- death.risk[,col.key]
tm.SP0["successP-THR","successP-THR",] <- 1-revision.risk.sp0 - death.risk[,col.key]
## transitions out of R-THR
tm.SP0["R-THR","Death",] <- tp.RTHR2dead + death.risk[,col.key]
tm.SP0["R-THR","successR-THR",] <- 1 - tp.RTHR2dead - death.risk[,col.key]
## transitions out of success-THR
tm.SP0["successR-THR","R-THR",] <- tp.rrr
tm.SP0["successR-THR","Death",] <- death.risk[,col.key]
tm.SP0["successR-THR","successR-THR",] <- 1 - tp.rrr - death.risk[,col.key]
## no transitions out of death
tm.SP0["Death","Death",] <- 1
}
```

This vectorised approach is easy to code, and will be quicker than using a loop. For those that are unsure about how the vectorised approach works, or how this approach results in the same data in the array, the following section goes through how the data is provided to the array, step-by-step.

Walkthrough of how the vectorised approach works

First, let's look at the transitions with a fixed probability over time (i.e. not a time-dependent transition probability). For example, the `tp.PTHR2dead` is equal to 0.02. In the loop (below), the parameter is assigned to the following position in each loop. This means that in all 60 positions in the array, the same value is assigned.

```
tm.SP0["P-THR","Death",i] <- tp.PTHR2dead
```

We can check this with the following code:

```
tm.SP0["P-THR","Death",]
```

```
##    1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16
## 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
##   17   18   19   20   21   22   23   24   25   26   27   28   29   30   31   32
## 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
##   33   34   35   36   37   38   39   40   41   42   43   44   45   46   47   48
## 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
##   49   50   51   52   53   54   55   56   57   58   59   60
## 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
```

A more efficient method, without the use of a loop, would be to assign the value as the following. This will assign the value for all 60 positions in the array.

```
  tm.SP0["P-THR","Death",] <- tp.PTHR2dead
```

Where the transition probabilities are time dependent, this can also be done as long as the length of the vector (containing time-dependent transition probabilities) is the same as the dimensions of the matrix that it is replacing. The first value in the vector will replace the first position in the array, and so on...

Here is some code to show you.

```
#  First we will create the blank array, filled with 0 values
tm.SP0 <- array(data=0,dim=c(5, 5, 60),
                dimnames= list(state.names, state.names, 1:60))

# Next lets look at the following position in the array
#  we will just show the first four positions for simplicity

tm.SP0["successP-THR","R-THR",1:4]
```

```
## 1 2 3 4
## 0 0 0 0
```

```
revision.risk.sp0[1:4]
```

```
## [1] 0.0004558791 0.0007926651 0.0010023327 0.0011684956
```

```
# Are the lengths of these values the same?
length(tm.SP0["successP-THR","R-THR",] ) == length(revision.risk.sp0)
```

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

```
# Let's assign these revision risk values to the array and see the output
tm.SP0["successP-THR","R-THR",] <- revision.risk.sp0

# Notice that the first value of the revision risk vector is now
#  applied to the first matrix (in the array)
tm.SP0[,,1:4]
```

```
## , , 1
##
##              P-THR successP-THR        R-THR successR-THR Death
## P-THR            0            0 0.0000000000            0     0
## successP-THR     0            0 0.0004558791            0     0
## R-THR            0            0 0.0000000000            0     0
## successR-THR     0            0 0.0000000000            0     0
## Death            0            0 0.0000000000            0     0
##
## , , 2
##
##              P-THR successP-THR        R-THR successR-THR Death
## P-THR            0            0 0.0000000000            0     0
## successP-THR     0            0 0.0007926651            0     0
## R-THR            0            0 0.0000000000            0     0
## successR-THR     0            0 0.0000000000            0     0
## Death            0            0 0.0000000000            0     0
##
## , , 3
##
##              P-THR successP-THR       R-THR successR-THR Death
## P-THR            0            0 0.000000000            0     0
## successP-THR     0            0 0.001002333            0     0
## R-THR            0           0 0.000000000           0     0
## successR-THR     0           0 0.000000000           0     0
## Death            0           0 0.000000000           0     0
##
## , , 4
##
##              P-THR successP-THR       R-THR successR-THR Death
## P-THR            0           0 0.000000000           0     0
## successP-THR     0           0 0.001168496           0     0
## R-THR            0           0 0.000000000           0     0
## successR-THR     0           0 0.000000000           0     0
## Death            0           0 0.000000000           0     0
```

However, as hinted above, when using the vectorisation approach you need to ensure that the length of the vectors used is correct and appropriate for the assignment processes it is being used for. As always, test and de-bug (i.e. remove errors from) your code as you go through when creating your models to make sure the outputs are looking how they are expected to.

There are also other ways to speed up code and assignments (such as using functions within the `tidyverse` or `data.table` packages), if this is something you are interested in, why not have a go at different approaches for our models and post any new super-fast methods on the discussion boards!