Data Science for Economists

Lecture 5: Loops in R

Drew Van Kuiken
University of North Carolina | ECON 370

Table of contents

Introduction

Agenda

This class will cover loops in R and their quirks

While loops are fundamental to programming, they are slow in R and sh
when possible.

There is a nice family of functions called the apply family that can conde
better looking code.

- However, they are still ultimately loops and are just as slow as stan

When in doubt, make sure your code is vectorized when possible!

Drew's thoughts: iiiiiiiii don't know? Loops are not necessarily slow. They

They're also easier than putting stuff in a matrix and less error prone.

Overall: develop your own intuitions about when to vectorize vs when to
towards vectorization, especially when you're learning.

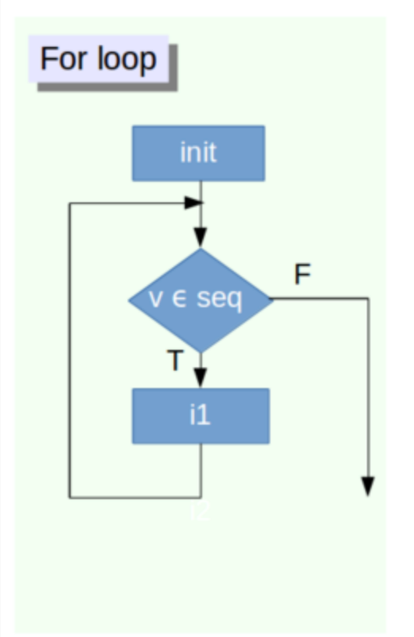
Loops

Motivation: For loop

At times we would look to do the same task multiple times that only cha
iteration.

This can be done with a for loop.

With for loops, you need something to "loop over" and an index that ind
iteration you're on.



Our First Loop

```
sum_val = 0
for(i in 1:10){
  sum_val = sum_val + i
}
sum_val
```

```
## [1] 55
```

```
sum(1:10)
```

```
## [1] 55
```

```
11*10/2
```

```
## [1] 55
```

Note: for sequence of integers, Sum = n(a+l)/2 = 10*(1+10)/2

Two Approaches to Loops

There are two approaches to loops, and more specifically, what to loop o

1. Loop over objects in a vector, list, data.frame, etc.
2. Loop over indexes for that vector, list, data.frame, etc.

For ease of understanding *what* is being looped over, 1. is usually best.

- However, it requires keeping track of indexes in another variable.

For ease of storing variables, 2. is usually easier.

- However, what exactly is being looped over can be obscured.

Neither is always better than another and at some point comes down to
preference.

Two Approaches: An Example

```
Nsim       = 100         #set number of simulations/draws
norm_draws = rnorm(Nsim) #draw N(0,1) random variables
out1       = rep(0,Nsim) #initialize output 1: MORE ON THIS LATER
out2       = rep(0,Nsim) #initialize output 2: MORE ON THIS LATER
n          = 1           #initialize counter

for(draw in norm_draws){
  out1[n] = draw^2       #square the draw and store it
  n       = n + 1        #advance the counter
} #NOTE THAT A COUNTER IS NEEDED

for(i in 1:Nsim){
  out2[i] = norm_draws[i]^2 #square the ith draw and store it
} #notice no counter needed

all.equal(out1,out2) #test to see if these approaches are the same
```

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

More Examples: Advanced Sums

Suppose we wanted to calculate

$$\sum_{a=1}^{20} \sum_{b=1}^{15} \frac{e^{\sqrt{a}} \log\left(a^5\right)}{5 + \cos(a)\sin(b)}$$

```r
val = 0
for(a in 1:20){
  for(b in 1:15){
    val = val + (exp(sqrt(a))*log(a^5))/(5+cos(a)*sin(b))
  }
}
val
```

```
## [1] 25922.81
```

The loop works, but it is not needed in R!

- Will return to this at the end of the lecture.

Preallocation

Many times you'll want to use a loop to "fill up" a matrix or vector.

It is best practice to "preallocate" this object to the correct size before fi

There are a few reasons for this, but it ultimately comes down to speed:

- Changing the size of the object inside the loop each iteration makes than they already are in R!

Preallocation: Example

```r
N            = 100000
my_vec       = c(0)
my_vec_pre = rep(0,N)

for(i in 1:N){
  my_vec[i] = i^2
}

for(i in 1:length(my_vec_pre)){
  my_vec_pre[i] = i^2
}
```

Both run! But let's look at the speed.

Preallocation: Comparing Speed

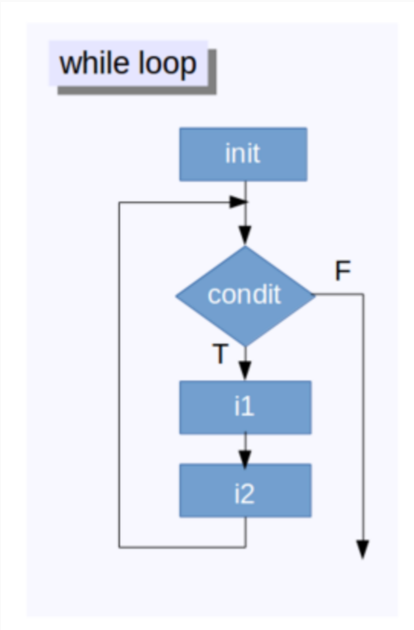
```
mbm = microbenchmark(
  "no_preal"={
    my_vec     = c(0)
    for(i in 1:N){
      my_vec[i] = i^2
      }},
  "preal"={
    my_vec_pre = rep(0,N)
    for(i in 1:length(my_vec_pre)){
      my_vec_pre[i] = i^2
      }},times=1000)
mbm
```

```
## Unit: milliseconds
##      expr       min        lq      mean    median        uq
##  no_preal 29.230398 33.371610 37.326908 36.693882 39.136462 253.
##     preal  7.515862  8.042438  8.973442  8.390997  9.075575  72.
```

Bottom line: Preallocate objects whenever possible!

While loops

- For loops are not the only types of loops in R!
- Another type is while loops.
- Instead of looping through objects or indexes, we continue to do so condition is no longer met.
- This can be really useful for some of the things we will use later on.
- Can be dangerous though: infinite loop!
  - Not so much in RStudio, though.



Our First While Loop

```
val = 0
n   = 1
while(n < 31){
  val = val + n
  n   = n + 1
}
val
```

```
## [1] 465
```

```
30*31/2
```

```
## [1] 465
```

The while loop continues to increase n by one and add it to `val` until n

Econ Application: English Auctio

Art and other valuable objects are often sold in an ascending auction wl
low and bidders continue to increase the price until only one bidder ren

We can "simulate" these auctions using while loops.

Suppose there are N bidders and each bidder i has a "valuation" v_i tha
much she values the object.

As well, suppose there is some fixed amount Δ that bidders must increa
round (no more and no less) and bidders continue to alternate until onl
remains. Also suppose that the initial price $p_0 = \Delta$.

So each round t, if a bidder is willing to bid, the price increases to $p_t =$

Bidder i is willing to bid if $v_i \geq p_t$ or $v_i \geq p_{t-1} + \Delta$.

Let's go to R.

Another Application: An Easy Fix

In math, a fixed point, x^*, of a function f is defined as a value where $f($

- So a fixed point is a point where when we apply the function to it, w
value back!

While loops can be used to calculate these when they exist.

The idea is to keep applying the function over and over again until the v
enough"

So $x_{n+1} = f(x_n)$. If $|x_{n+1} - x_n|$ is "small," we stop.

If not, replace x_n with x_{n+1}, and continue.

- So $x_{n+2} = f(x_{n+1})$ and then compare x_{n+2} and x_{n+1}.

We will use the function $f(x) = \sqrt{x}$.

$\sqrt{1} = 1$ and $\sqrt{0} = 0$. So these are our candidate fixed points.

However, we will only get to one of them no matter which starting value

Calculating Fixed Points

```
eps   = .Machine$double.eps #set tolerance
x_n   = 5000                #starting guess
x_np1 = sqrt(x_n)           #apply function

while(abs(x_n - x_np1) > eps){
  x_n   = x_np1     #update guess
  x_np1 = sqrt(x_n) #apply function
}
x_np1
```

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

```
x_n   = 0.00001             #starting guess
x_np1 = sqrt(x_n)           #apply function

while(abs(x_n - x_np1) > eps){
  x_n   = x_np1     #update guess
  x_np1 = sqrt(x_n) #apply function
}
x_np1
```

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

Implementing Fail-Safes

When writing while loops, it is often good practice to implement a fail-sa
loop doesn't run for forever.

This could be because the code isn't converging as quickly as we'd like o
and the code will never converge because you wrote it wrong.

I did not need one above because that question has really good converg
(and I trust my code ☺).

To implement a fail-safe, we need to create a new variable and use som
properties we talked about last lecture.

Ideas?

We want it to stop when $|x_{n+1} - x_n| < \varepsilon$ or $n > \bar{N}$ where $\bar{N}$ is some
iterations we set.

So what is the "while condition?" Hint: DeMorgan's Law!

Implementing a Fail-Safe

```
x_n    = 50000                #starting guess
x_np1 = sqrt(x_n)             #apply function
n      = 1
MaxIt = 100000

while(abs(x_n - x_np1) > eps & n < MaxIt){
  x_n    = x_np1      #update guess
  x_np1 = sqrt(x_n) #apply function
  n      = n + 1      #increase counter
}

#check to see why loop stopped
if(abs(x_n - x_np1) > eps){
  stop("Did not find fixed point!")
} else{
  print(c(x_np1,n))
}
```

```
## [1]  1 55
```

Fail-Safes (Fixed Points)

Fixed points don't always exist, and even when they do, we're not always
them via the iterative procedure I described.

That's where these fail safes can come into play.

Consider the function $f(x) = 2x$. f has a fixed point (and only one fixe
however, we are not guaranteed to ever find it via the iterative procedur

Therefore, the fail safe needs to be triggered so our loop doesn't go on f

Necessary Fail-Safes

```
x_n    = 0.0001 #starting guess
x_np1 = 2*x_n   #apply function
n      = 1      #initialize counter
MaxIt = 1000    #fix max iterations

while(abs(x_n - x_np1) > eps & n < MaxIt){
  x_n    = x_np1      #update guess
  x_np1 = 2*x_n      #apply function
  n      = n + 1      #increase counter
}

if(abs(x_n - x_np1) > eps){
  stop("Did not find fixed point!")
} else{
  print(c(x_np1,n))
}
```

```
## Error in eval(expr, envir, enclos): Did not find fixed point!
```

Repeat Loops

In R, there is a third kind of loop: the repeat loop.

The repeat loop will continue to do something until you manually break

These are slightly different than while loops; however, while loops can b
their behavior quite easily.

I would mostly recommend avoiding repeat loops.

Repeat Loops

```r
val = 0
n   = 1
repeat{
  val = val + n
  n   = n + 1
  if(val > 30) break
}
print(c(val, n))
```

```
## [1] 36  9
```

```r
val = 0
n   = 1
while(TRUE){
  val = val + n
  n   = n + 1
  if(val > 30) break
}
print(c(val, n))
```

```
## [1] 36  9
```

The Apply Family of Functi

The Apply Family

In R, there are a family of functions called the apply family.

They can be used to write loops in a much more compact format.

The idea is to have some vector-like object that you would to do someth
like manner, and then "apply" some function to each element of the obj

If you'd like to see more about the apply family, I would recommend foll
tutorial for more.

The Apply Function

The first one we will look at is the apply function.

It takes three arguments:

1. an array (matrix, vector, etc.)
2. a "margin" (which dimension to apply over)
3. a function

It takes the array and then applys the function over the dimension that
margins argument.

Apply: An Example

```
rand_mat = matrix(rnorm(3*2),ncol=3)
rand_mat
```

```
##            [,1]       [,2]         [,3]
## [1,]  1.1837459  0.7296892 0.0007641864
## [2,] -0.7715014 -0.5870856 2.2144653193
```

```
apply(rand_mat,1,sum)
```

```
## [1] 1.9141993 0.8558783
```

```
apply(rand_mat,2,sum)
```

```
## [1] 0.4122445 0.1426036 2.2152295
```

- MARGIN = 1, the sum function is applied to each row.
  - So we are summing across columns
- MARGIN = 2, the sum function is applied to each column.
  - So we are summing across rows.

Apply's Connection to Loops

It might not be entirely obvious apply's connection to loops.

When MARGIN = 1, this is what apply is doing:

```
out = rep(0,nrow(rand_mat))
for(i in 1:nrow(rand_mat)){
  out[i] = sum(rand_mat[i,])
}
out
```

```
## [1] 1.9141993 0.8558783
```

Likewise, when MARGIN = 2, this is what apply is doing:

```
out = rep(0,ncol(rand_mat))
for(i in 1:ncol(rand_mat)){
  out[i] = sum(rand_mat[,i])
}
out
```

```
## [1] 0.4122445 0.1426036 2.2152295
```

Beyond Apply

As seen above, apply can simplify loops and results in much cleaner coc

- Though, is it more readable?

While the apply function is useful, it has its limitations.

1. It can only be used on array-like objects.
2. It will only return a vector or array.

There are other functions that can be used on a wider class of objects a
non-arrays.

- lapply: returns a list the same length as the object
- sapply: returns the "most simple" version of the output of lapply tha
  - I know, this sounds ambiguous because it is!
- vapply: the same as sapply, but an output type must be specified.
  - Generally, safer to use.
- tapply
  - I have never used this one. Just know it exists.

x-apply Examples

```
my_list = list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FA
my_list
```

```
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $beta
## [1]  0.04978707  0.13533528  0.36787944  1.00000000  2.71828183
## [7] 20.08553692
##
## $logic
## [1]  TRUE FALSE FALSE  TRUE
```

```
lapply(my_list, mean)
```

```
## $a
## [1] 5.5
##
## $beta
## [1] 4.535125
##
## $logic
## [1] 0.5
```

x-apply Examples (Cont.)

```
sapply(my_list, mean)
```

```
##        a     beta    logic
## 5.500000 4.535125 0.500000
```

```
lapply(my_list, quantile, probs = (1:3)/4)
```

```
## $a
##  25%  50%  75%
## 3.25 5.50 7.75
##
## $beta
##       25%       50%       75%
## 0.2516074 1.0000000 5.0536690
##
## $logic
## 25% 50% 75%
## 0.0 0.5 1.0
```

x-apply Examples (Cont.)

```
sapply(my_list, quantile)
```

```
##           a        beta logic
## 0%     1.00  0.04978707   0.0
## 25%    3.25  0.25160736   0.0
## 50%    5.50  1.00000000   0.5
## 75%    7.75  5.05366896   1.0
## 100% 10.00 20.08553692   1.0
```

x-apply Examples (Cont.)

By default, sapply will apply functions to columns (across rows) of data.f
2 in the apply function.

Note, there is no MARGIN argument for sapply, lapply, or vapply.

```
data(mtcars)
sapply(mtcars,summary)
```

```
##              mpg    cyl     disp       hp     drat      wt      q
## Min.    10.40000 4.0000  71.1000  52.0000 2.760000 1.51300 14.50
## 1st Qu. 15.42500 4.0000 120.8250  96.5000 3.080000 2.58125 16.89
## Median  19.20000 6.0000 196.3000 123.0000 3.695000 3.32500 17.71
## Mean    20.09062 6.1875 230.7219 146.6875 3.596563 3.21725 17.84
## 3rd Qu. 22.80000 8.0000 326.0000 180.0000 3.920000 3.61000 18.90
## Max.    33.90000 8.0000 472.0000 335.0000 4.930000 5.42400 22.90
##              am   gear   carb
## Min.    0.00000 3.0000 1.0000
## 1st Qu. 0.00000 3.0000 2.0000
## Median  0.00000 4.0000 2.0000
## Mean    0.40625 3.6875 2.8125
## 3rd Qu. 1.00000 4.0000 4.0000
## Max.    1.00000 5.0000 8.0000
```

Vectorization

To Loop or Not To Loop

Generally in R, you want to avoid loops at all costs. This is because they

Developing your programming style in R requires learning when to use l

```
mbm = microbenchmark(
  "loop"={
    N   = 100000    #set size of vector
    out = rep(0, N) #preallocate vector
    for(i in 1:N){
      out[i] = i^2  #fill in vector with square of index
      }},
  "vectorized"={
    N   = 100000    #set size of vector
    out = 1:N       #preallocate vector
    out = out^2     #square each index
   },times=1000)
mbm
```

```
## Unit: microseconds
##        expr      min        lq      mean    median       uq
##        loop 7119.313 7807.3000 10539.793 8412.680 9873.459 14986
##  vectorized  272.527  727.9515  1252.171  770.984  882.128 10332
```

Returning To Advanced Sums

Earlier, we wanted to calculate the following sum:

$$\sum_{a=1}^{20}\sum_{b=1}^{15}\frac{e^{\sqrt{a}}\log\left(a^5\right)}{5+\cos(a)\sin(b)}$$

While we used a loop, it was not necessary. If we expand out every comb
then, we can use vectorized operations.

```
aANDb = expand.grid(a=1:20,b=1:15)
a     = aANDb$a
b     = aANDb$b
sum((exp(sqrt(a))*log(a^5))/(5+cos(a)*sin(b)))
```

```
## [1] 25922.81
```

Benchmarking These Sums

```
mbm = microbenchmark(
  "loop"={
    val = 0
    for(a in 1:20){
      for(b in 1:15){
        val = val + (exp(sqrt(a))*log(a^5))/(5+cos(a)*sin(b))}}},
  "vectorized"={
    aANDb = expand.grid(a=1:20,b=1:15)
    a     = aANDb$a
    b     = aANDb$b
    sum((exp(sqrt(a))*log(a^5))/(5+cos(a)*sin(b)))
   },times=1000)
mbm
```

```
## Unit: microseconds
##        expr      min        lq       mean   median        uq
##        loop 5971.643 6595.8195 12901.8845 7383.849 10190.610 302
##  vectorized  112.349  132.3815   325.7309  179.524   205.061  58
```

```
mean(mbm[mbm$expr=="loop","time"])/mean(mbm[mbm$expr=="vectorized"
```

```
## [1] 39.60903
```

When Must We Use Loops?

Sometimes, the use of a loop cannot be avoided. This might be for the f

1. Calculations depend on previous calculations.
2. The size of an "inner loop" changes based on the values of the "out
3. Too difficult to do the "prep-work" mentally for the vectorized opera

```
}
```

Calculations That Depend on Ot

An $AR(1)$ Time Series is a perfect example of an economic application
absolutely necessary.

An $AR(1)$ model says that today's value of y, called y_t, depends on yes
scaled by some value ρ, plus some constant δ, plus some error term ε_t.

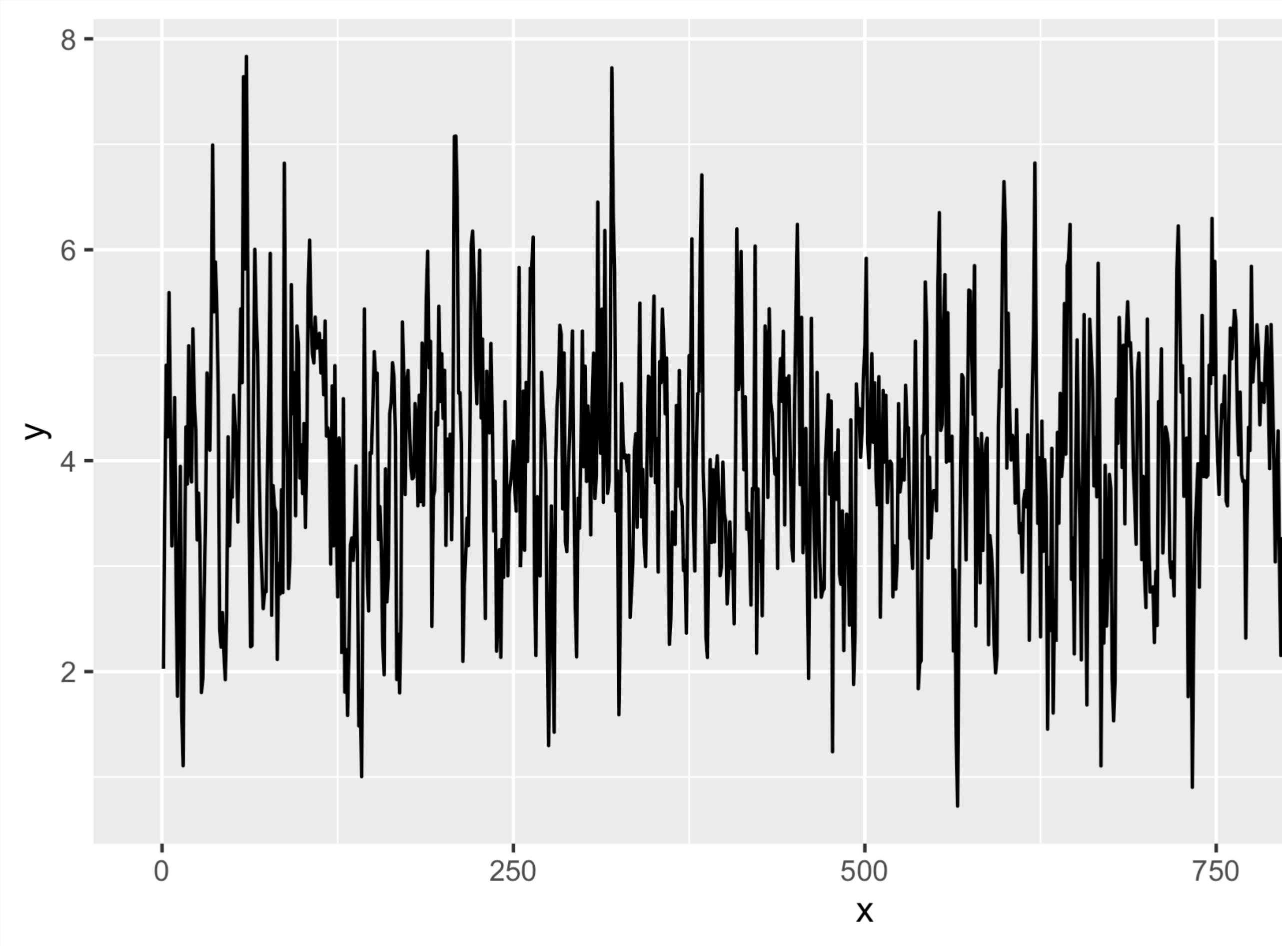
In math, that is

$$y_t = \delta + \rho y_{t-1} + \varepsilon_t$$

```
rho          = 0.5
delta        = 2
N            = 1000
AR1_ts       = rep(0,N)
AR1_ts[1] = delta + rho*(delta/(1-rho)) + rnorm(1)
for(i in 2:N){
  AR1_ts[i] = delta + rho*AR1_ts[i-1] + rnorm(1)
}
```

Note: I did not use t as the loop variable because of the function `t()`. I
cause a namespace conflict.

AR(1) Plot



Inner Loop Dependency

Sometimes when loops are nested (like our advanced sums), the inner l
values of the outer loop.

In this case, loops cannot be entirely avoided.

- Though, they can be minimized.

Consider a slight modification of the advanced sum we saw earlier

$$\sum_{a=1}^{20}\sum_{b=1}^{a}\frac{e^{\sqrt{a}}\log\left(a^5\right)}{5+\cos(a)\sin(b)}$$

Instead of looping b from 1 to 15, now the max value of b depends on th
.

In this case, a loop cannot be avoided.

- At least, without making a specialized grid which will be very tediou

Dependent Loops

```
val = 0
for(a in 1:20){
  for(b in 1:a){
    val = val + (exp(sqrt(a))*log(a^5))/(5+cos(a)*sin(b))
  }
}
val
```

```
## [1] 27100
```

With some thinking and brute force, the loops might be able to be elimi
tedious.

However, if the speed of your code matters, it is worth spending the tim

In Conclusion

Loops are very valuable to understand conceptually, but should be avoic implementing code in R.

There are variations on loops called while loops that can be very useful things.

There are a family of functions called the apply family which condense l compact syntax. However, they are still loops at heart (and just as slow).

For further experience, use `swirl` for loops and the apply functions.

For further reading, please see this link. It was very helpful when making

Next lecture(s): Miscellane