Performance and Benchmarking - Intro

Why Benchmark ? Well maybe you don't need to if your code runs well and fast enough for your requirements.

On the other hand there are many different ways to do things in R and sometimes you wind up doing things that take a very long time.

So you need ways to do comparisons and pick the approach that is "best" and most efficient.

Performance and Benchmarking - Intro

C, C++, FORTRAN are "compiled" languages (there are many others)

FORTRAN was introduced in 1957, C was introduced in 1969, C++ in 1983

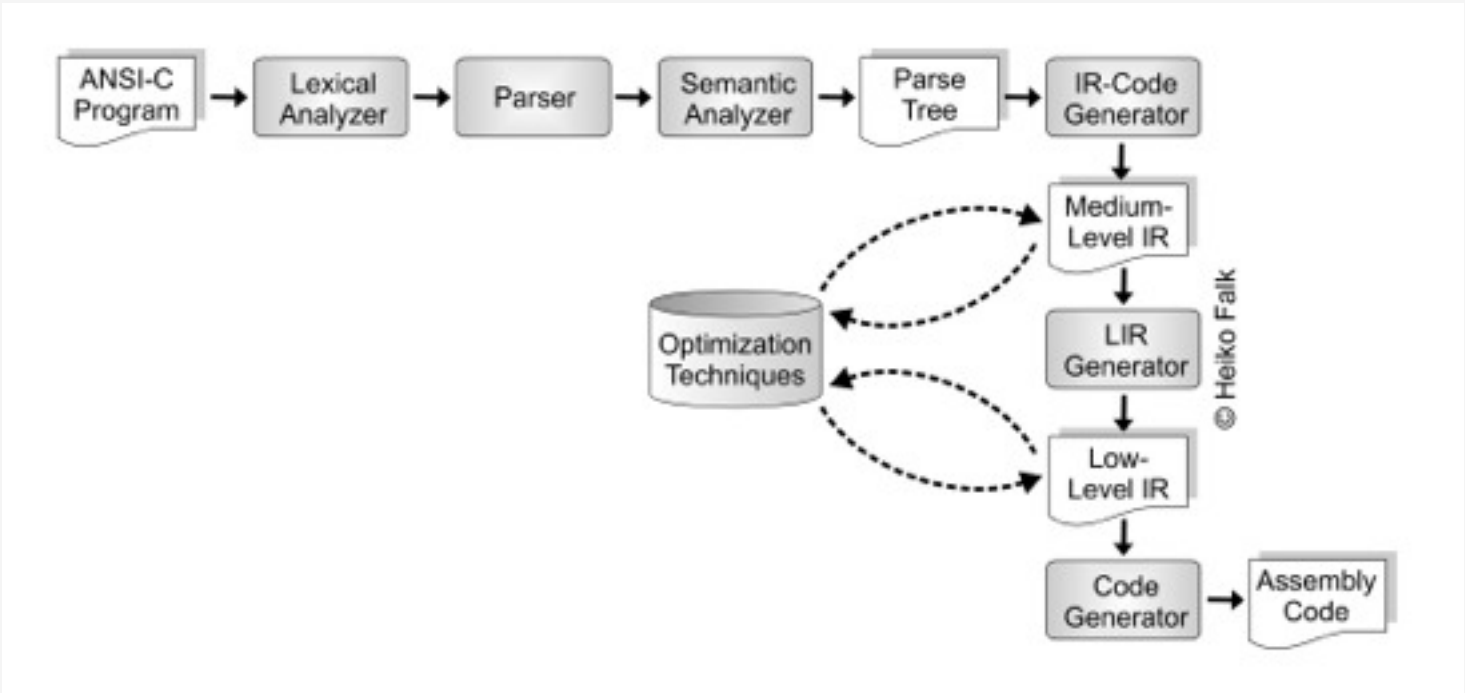
Programs/"source code" created in these languages are "read" by another program, called a compiler, that translates the entire program in to "machine code" that can then be passed to the operating system for rapid execution. T

Modern compliers can make improvements and optimizations to the code automatically giving even better performance.

A problem with this approach is that users need to recompile their programs for every change they make to the code.

Performance and Benchmarking - Intro

This shows us what is involved in compilation. Note that the compiler handles all of the steps for us so its not like we have to do them all separately. But it takes a lot to take "source code" and turn it into "machine code" - a computer's "native language".



http://ls12-www.cs.tu-dortmund.de/daes/de/forschung/source-code-optimization/motivation.html

Performance and Benchmarking - Intro

R is an "interpreted" language as is Python, Perl, Ruby, and many others.

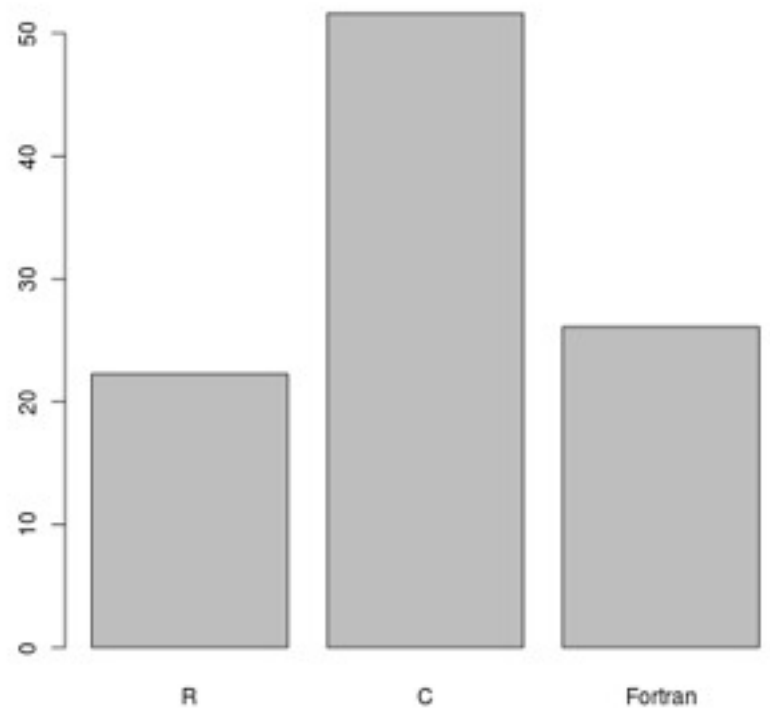
Interpreted languages have some advantages such as platform independence and a quicker development cycle since there is no compilation step involved. However, they tend to be slower since there is no direct translation into "machine code" before execution.

Each line is "interpreted" in sequence which is easier on the user but doesn't result in great efficiencies.

The language R is written in R with a large portion being written in C and FORTRAN. Certain functions are written in C and FORTRAN to speed up computation.

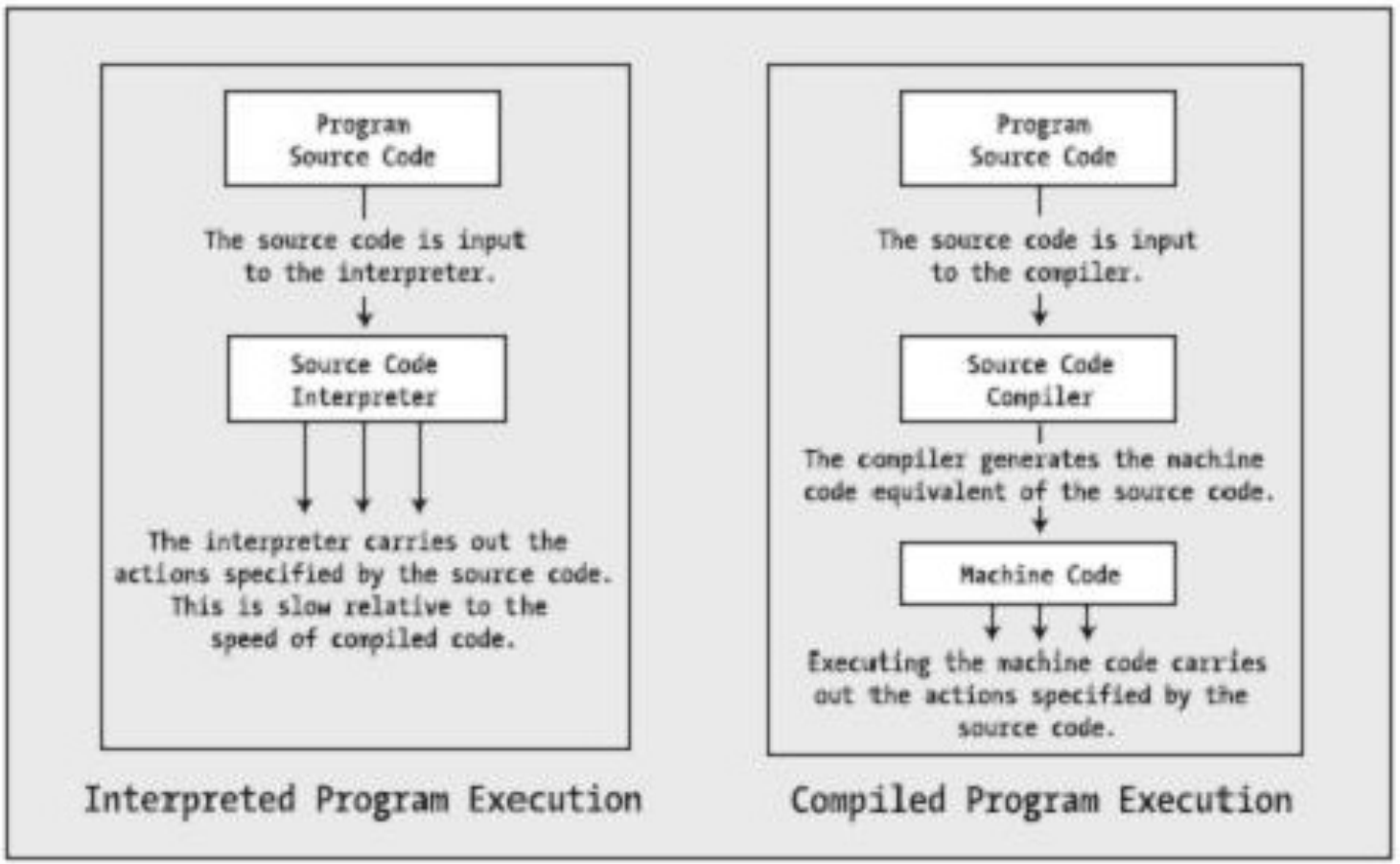
Performance and Benchmarking - Intro

So the natural question follows: *how much* of R is written in the R language? The librestats blog did an analysis of the latest R source code distribution, and found that 22% of the R's lines of code are in R.



For comparison, about 50% of R is written in C, and just under 30% in Fortran -- for the current 2.13.1 release, at least. Of course, these ratios have changed over time with each release of R, as this analysis of the R codebase from Ohloh shows:

Performance and Benchmarking - Intro



http://www.devshed.com/c/a/Practices/Basic-Ideas/1/

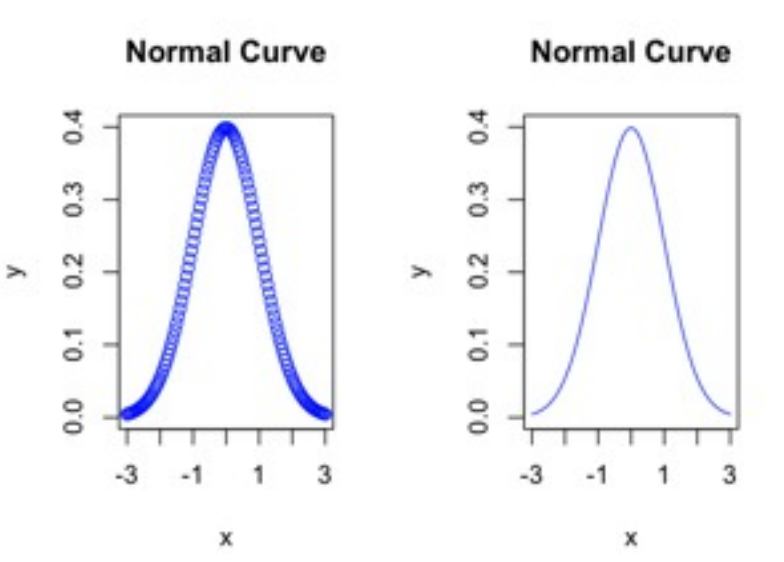
Performance and Benchmarking - Intro

Let's set up a test case:

```
x = seq(-3, 3, by=0.05)


# The "by" argument let's us specify a "step" or "interval" value


x
  [1] -3.00 -2.95 -2.90 -2.85 -2.80 -2.75 -2.70 -2.65 -2.60 -2.55 -2.50 -2.45
 [13] -2.40 -2.35 -2.30 -2.25 -2.20 -2.15 -2.10 -2.05 -2.00 -1.95 -1.90 -1.85
 [25] -1.80 -1.75 -1.70 -1.65 -1.60 -1.55 -1.50 -1.45 -1.40 -1.35 -1.30 -1.25
 [37] -1.20 -1.15 -1.10 -1.05 -1.00 -0.95 -0.90 -0.85 -0.80 -0.75 -0.70 -0.65
 [49] -0.60 -0.55 -0.50 -0.45 -0.40 -0.35 -0.30 -0.25 -0.20 -0.15 -0.10 -0.05
 [61]  0.00  0.05  0.10  0.15  0.20  0.25  0.30  0.35  0.40  0.45  0.50  0.55
 [73]  0.60  0.65  0.70  0.75  0.80  0.85  0.90  0.95  1.00  1.05  1.10  1.15
 [85]  1.20  1.25  1.30  1.35  1.40  1.45  1.50  1.55  1.60  1.65  1.70  1.75
 [97]  1.80  1.85  1.90  1.95  2.00  2.05  2.10  2.15  2.20  2.25  2.30  2.35
[109]  2.40  2.45  2.50  2.55  2.60  2.65  2.70  2.75  2.80  2.85  2.90  2.95
[121]  3.00
```

Performance and Benchmarking - Intro

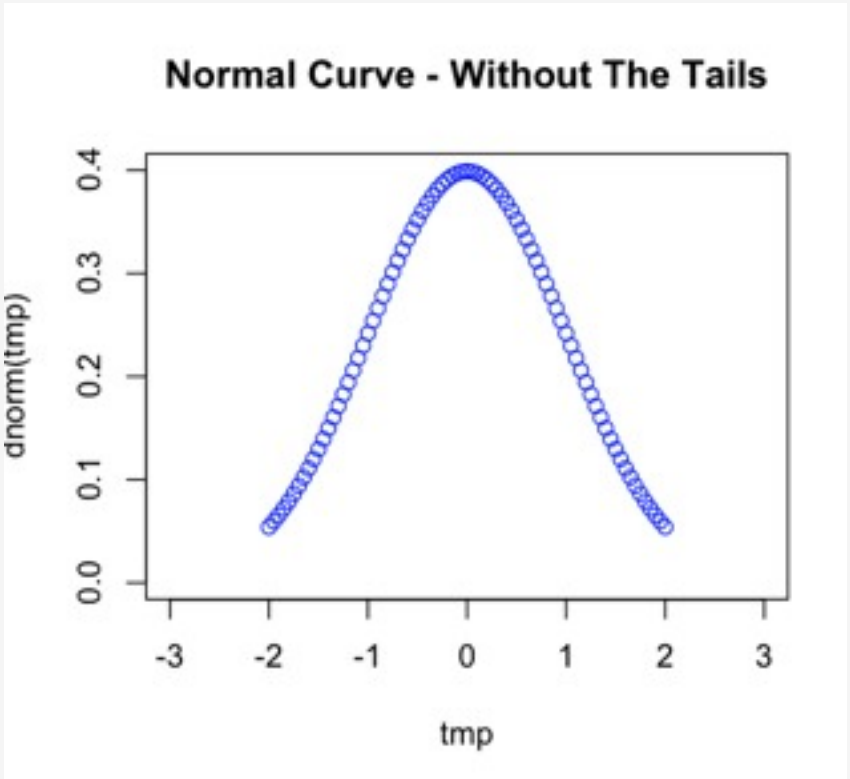
```
x = seq(-3, 3, by=0.05)
y = dnorm(x)

# Now we plot the data

par(mfrow=c(1,2))   # Set the plot region to be one row and two columns

plot(x, y, xlim=c(-3,3), ylim=c(0,0.4), main="Normal Curve", col="blue")

plot(x,y,xlim=c(-3,3),ylim=c(0,0.4),main="Normal Curve",col="blue",type="l")
```



Performance and Benchmarking - Intro

And we could look at points between -2 and 2 also just as easily.

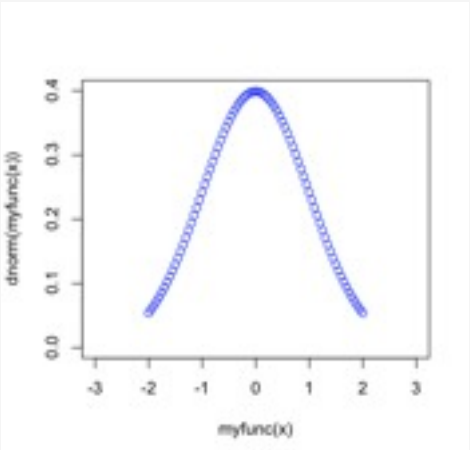
```
tmp = x[(x >= -2) & (x <= 2)]
plot(tmp,dnorm(tmp),xlim=c(-3,3),ylim=c(0,0.4),
         main="Normal Curve - Without Tails",col="blue")
```



Performance and Benchmarking - Intro

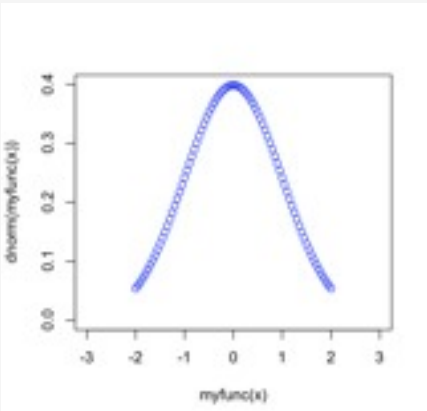
Note that in other programming languages we would use a for loop structure to do this. We can also do this in R. It isn't particularly difficult.

```
x = seq(-3, 3, by=0.05)

hold = vector()
for (ii in 1:length(x)) {
    if ( (x[ii] >= -2) & (x[ii] <= 2) ) {
       hold[ii] = x[ii]
    }
}

# Same "tails" plot as before

plot(hold,dnorm(hold),xlim=c(-3,3),ylim=c(0,0.4),col="blue")
```



Performance and Benchmarking - Intro

```
# We could put this into a function also. It doesn't change the results

myfunc <- function(x) {
  hold = vector()
  for (ii in 1:length(x)) {
    if ( (x[ii] >= -2) & (x[ii] <= 2) ) {
       hold[ii] = x[ii]
    }
  }
   return(hold)
}

plot(myfunc(x),dnorm(myfunc(x)), xlim=c(-3,3),ylim=c(0,0.4),col="blue")
```



Performance and Benchmarking - Intro

So now let's generate a lot more points and pull out the tails. We also want to get an idea about how long it takes to do this. What could we do to get the timings ? One way is to use Sys.time() as a "stopwatch".

```
Sys.time()
[1] "2013-11-09 20:01:00 EST"

# Wait 5 seconds

Sys.time()
[1] "2013-11-09 20:01:05 EST"

now = Sys.time()

# Wait for some period of time

later = Sys.time()

later - now
Time difference of 5.272433 secs
```

Performance and Benchmarking - Intro

This is just okay. However, there is some delay between the first call to `Sys.time()` and when `tmp = x[(x >= -2) & (x <= 2)]` is executed, and then the last call to `Sys.time()`. But overall it does give us some idea of how long the process took.

```
Sys.time()
[1] "2013-11-08 15:57:53 EST"

x = seq(-3, 3, by=0.0001)
length(x)
[1] 60001

start.time = Sys.time()

tmp = x[(x >= -2) & (x <= 2)]

Sys.time() - start.time

Time difference of 0.008060932 secs
```

Performance and Benchmarking - Intro

```
system.time                package:base                R Documentation

CPU Time Used

Description:

     Return CPU (and other) times that 'expr' used.

Usage:

     system.time(expr, gcFirst = TRUE)
     unix.time(expr, gcFirst = TRUE)

Arguments:

    expr: Valid R expression to be timed.

 gcFirst: Logical - should a garbage collection be performed
          immediately before the timing?  Default is 'TRUE'.

Details:

     'system.time' calls the function 'proc.time', evaluates 'expr',
     and then calls 'proc.time' once more, returning the difference
     between the two 'proc.time' calls.

     'unix.time' is an alias of 'system.time', for compatibility with
     S.

     Timings of evaluations of the same expression can vary
     considerably depending on whether the evaluation triggers a
     garbage collection.  When 'gcFirst' is 'TRUE' a garbage collection
     ('gc') will be performed immediately before the evaluation of
     'expr'.  This will usually produce more consistent timings.

:
```

Performance and Benchmarking - Intro

system.time() is better. It allows us to pass R code directly as an argument so the timing is more precise. And we can repeat the testing to see if the timing is consistent.

```
x = seq(-3, 3, by=0.0001)

length(x)
[1] 60001

system.time( tmp <- x[(x >= -2) & (x <= 2)] )
   user  system elapsed
  0.003   0.000   0.003

system.time( tmp <- x[(x >= -2) & (x <= 2)] )
   user  system elapsed
  0.004   0.000   0.004

system.time( tmp <- x[(x >= -2) & (x <= 2)] )
   user  system elapsed
  0.003   0.000   0.003
```

Performance and Benchmarking - Intro

If we wanted to replicate this many times, say 5, R has a function just for that purpose. Each column represents one execution of the system.time command. The results are stored in the columns.

```
x = seq(-3, 3, by=0.0001)

length(x)
[1] 60001

replicate(5, tmp <- system.time( x[(x >= -2) & (x <= 2)] ))
            [,1]  [,2]  [,3]  [,4]  [,5]
user.self  0.003 0.003 0.003 0.003 0.003
sys.self   0.000 0.000 0.000 0.000 0.000
elapsed    0.003 0.003 0.003 0.004 0.003
user.child 0.000 0.000 0.000 0.000 0.000
sys.child  0.000 0.000 0.000 0.000 0.000
```

Performance and Benchmarking - Intro

Okay, how long would the comparable for loop take ?

```
x = seq(-3, 3, by=0.0001)

length(x)
[1] 60001

myfunc <- function(x) {
  hold = vector()
  for (ii in 1:length(x)) {
    if ( (x[ii] >= -2) & (x[ii] <= 2) ) {
       hold[ii] = x[ii]
    }
  }
   return(hold)
}

system.time( myfunc(x) )
   user  system elapsed
  6.362   3.160   9.528

system.time( x[(x >= -2) & (x <= 2)] )  # Big difference !
   user  system elapsed
  0.004   0.000   0.004
```

Performance and Benchmarking - Intro

Actually, the bracket character is a function written in C which is why it is fast

```
> `[`
.Primitive("[")
```

Let's double the size of the vector and see how fast/slow the timings are.

```
x = seq(-3,3,by=0.000025)

length(x)
[1] 240001


system.time( x[(x >= -2) & (x <= 2)] )
   user  system elapsed
  0.014   0.000   0.014


system.time( myfunc(x) )
   user  system elapsed
109.284  45.376 154.782
```

Performance and Benchmarking - Intro

Is there a tool that helps us do benchmarking ? Yes. Several in fact. We will focus only on one here which is the rbencmark function.

You will have to install it since it is an add-on package. Use RStudio or the Windows GUI to do this or you can use the **install.packages** command. This package contains the function called **benchmark** which allows you to time the execution of any R function including those that you have written.

Performance and Benchmarking - Intro

It replicates the function a specified number of times. You can also compare the performance of more than one function at a time. Let's start out simple. Here we will time how long it takes to square the x vector.

As mentioned this will be replicated some number times with the default being 100. We will get back a variety of information.

```
install.packages("rbenchmark")  # do this once

library(rbenchmark)

x = seq(-3,3,by=0.0001)

length(x)
[1] 60001

benchmark( square = x^2)
    test replications elapsed relative user.self sys.self user.child sys.child
1 square          100   0.026        1     0.015    0.011          0         0
```

Performance and Benchmarking - Intro

We can compare how long it takes to square it vs cubing it. We can also specify the information to be returned

```
benchmark( square = x^2, cube = x^3, replications=5, columns=c("test","elapsed"))
    test elapsed
2   cube   0.006
1 square   0.001
```

So it takes longer to cube the vector than it does to square it - That makes sense. Let's look back at our original problem now.

```
x = seq(-3,3,by=0.01)
length(x)
[1] 601
```

We can use the benchmark function to time the performance of the bracket notation vs. the for loop although we already have some ideas about that.

Performance and Benchmarking - Intro

```
# Okay let's try it on a larger vector

x = seq(-3,3,0.01)
length(x)
[1] 601

t600 = benchmark(bracket = x[(x >= -2) & (x <= 2)],
            forloop = myfunc(x),replications=10, columns=c("test","elapsed"))
t600
     test elapsed
1 bracket   0.000
2 forloop   0.023
```

Performance and Benchmarking - Intro

```
# Okay let's try it on a larger vector

x = seq(-3,3,0.001)
# length(x)
# [1] 6001

t6K = benchmark(bracket = x[(x >= -2) & (x <= 2)],
            forloop = myfunc(x),replications=10, columns=c("test","elapsed"))

t6K
      test elapsed
1 bracket   0.003
2 forloop   0.511
```

Performance and Benchmarking - Intro

```
# Okay let's try it on a larger vector

x = seq(-3,3, 0.0001)
length(x)
[1] 60001

t60K = benchmark(bracket = x[(x >= -2) & (x <= 2)],
            forloop = myfunc(x),replications=10, columns=c("test","elapsed"))

t60K
     test elapsed
1 bracket   0.034
2 forloop  46.021
```

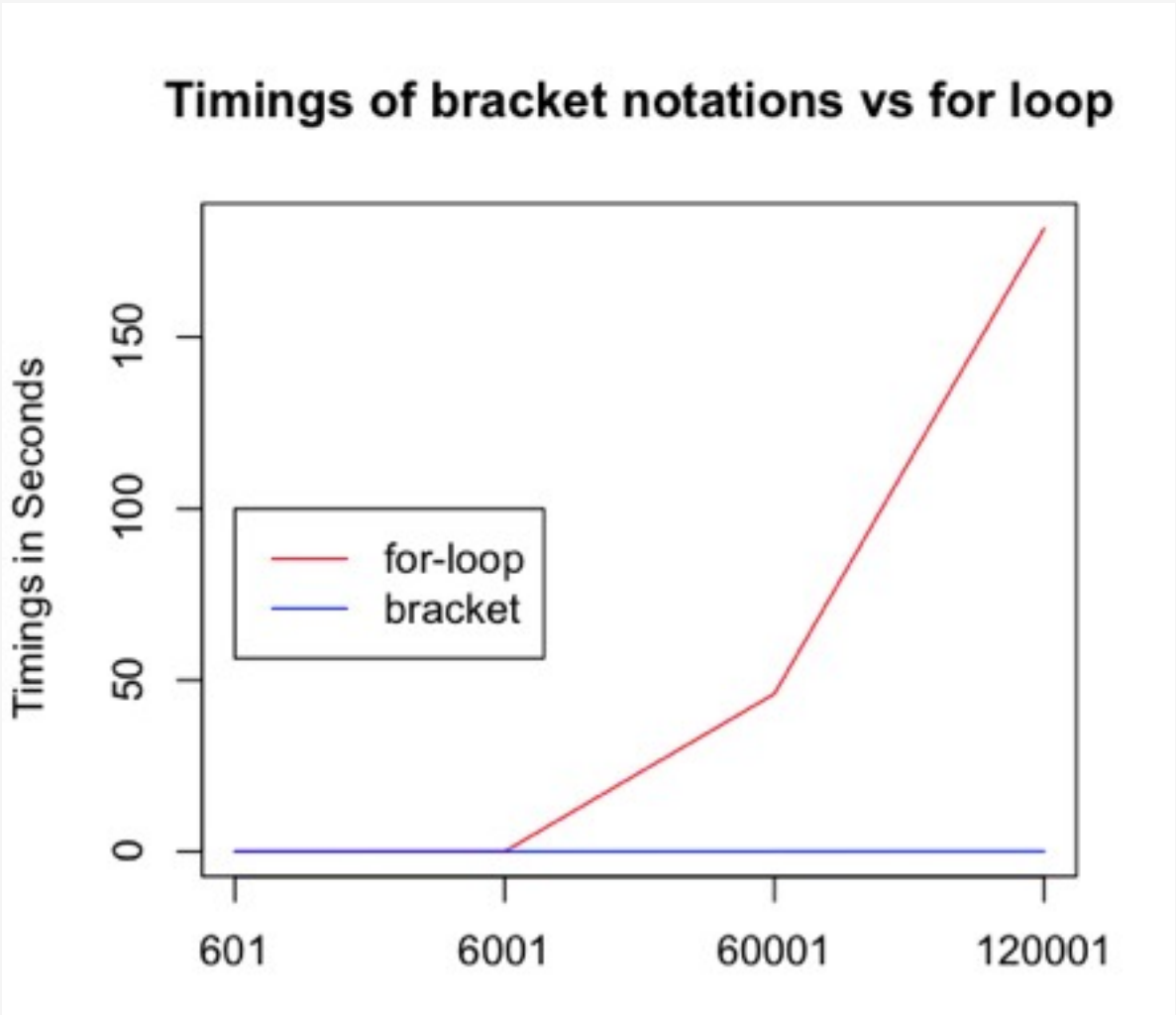
Performance and Benchmarking - Intro

```
# Okay let's try it on a larger vector

x = seq(-3,3, 0.00005)
length(x)
[1] 120001

t120K = benchmark(bracket = x[(x >= -2) & (x <= 2)],
            forloop = myfunc(x),replications=10, columns=c("test","elapsed"))

t120K
     test elapsed
1 bracket   0.067
2 forloop 179.109
```

Performance and Benchmarking - Intro

```
mtimings = cbind(t600[,2],t6K[,2],t60K[,2],t120K[,2])


colnames(mtimings)=c("601","6001","60001","120001")


mtimings
       601  6001  60001  120001
[1,] 0.001 0.000  0.033   0.066
[2,] 0.024 0.027 46.009 181.679

plot(1:4,mtimings[2,],pch=18,col="red",type="l",xaxt="n",
          main="Timings of bracket notations vs for loop",
          ylab="Timings in Seconds")

points(1:4,mtimings[1,],type="l",col="blue")
axis(1,1:4,labels=colnames(mtimings),xlab=c("vector size in elements"))

legend(1,100,c("for-loop","bracket"),col=c("red","blue"),lty=c(1,1))
```

Performance and Benchmarking - Intro



Performance and Benchmarking

Use vectorized functions where-ever possible. These are functions that know how to handle vectors as arguments. Add up the elements of a vector:

Not vectorized:

```
total = 0
x = rnorm(100,mean=100,sd=2)
for (ii in 1:length(x)) {
    total = total + x[ii]
}
total
[1] 9985.895
```

Vectorized:

```
sum(x)
[1] 9985.895

sum
function (..., na.rm = FALSE)  .Primitive("sum") # Written in C or FORTRAN
```

Performance and Benchmarking

Given a vector find the maximum value:

```
set.seed(233)
x = rnorm(10000)
```

 # Not vectorized:

```
maxval = x[1]
for (ii in 1:length(x)) {
   if (x[ii] > maxval) {
      maxval = x[ii]
   }
}
maxval
[1] 3.983584
```

 # Vectorized:

```
max(x)
[1] 3.983584

> max
function (..., na.rm = FALSE)  .Primitive("max")  # Written in C or FORTRAN
```

Performance and Benchmarking

Given a data frame, exclude any rows that contain a missing value as indicated by the presence of NA. the built in data frame airquality has records like this:

```
airquality[5:6,]
  Ozone Solar.R Wind Temp Month Day
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

Not vectorized: Ugh !

```
jj = 1
mylist = list()
for (ii in 1:nrow(airquality)) {
   numofnas = grep("NA", airquality[ii,])
   if (length(numofnas) == 0) {
       mylist[[jj]] = airquality[ii,]
       jj = jj +1
   } else {
       next
   }
}

nonadf = do.call(rbind, mylist)
```

Performance and Benchmarking

Given a data frame, exclude any rows that contain a missing value as indicated by the presence of NA. The built in data frame airquality has records like this:

```
airquality[5:6,]
  Ozone Solar.R Wind Temp Month Day
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6

Vectorized:  So much easier !

nonasdf = airquality[ complete.cases(airquality), ]
```

Performance and Benchmarking

Given two vectors of the same length find the maximum between the corresponding elements.

```
v1 = c(1,2,3,4,5)
v2 = c(5,4,3,2,1)

# So we would get back 5,4,3,4,5

# Not vectorized:     Please - No !

resvec = vector()
for (ii in 1:length(v1)) {
    if (v1[ii] >= v2[ii]) {
       resvec[ii] = v1[ii]
    } else {
       resvec[ii] = v2[ii]
    }
}

resvec
[1] 5 4 3 4 5
```

Performance and Benchmarking

Given two vectors of the same length find the maximum between the corresponding elements.

```
v1 = c(1,2,3,4,5)
v2 = c(5,4,3,2,1)

# So we would get back 5,4,3,4,5

# Vectorized:  Ah yes !

pmax(v1,v2)
[1] 5 4 3 4 5
```

Performance and Benchmarking

Always try to find a function that looks like it might do the job.

for loops aren't necessarily "evil" they just get slow when you use large input vectors.

knowing how to write for loops will help you when learning other languages

Many times the vectorized solution requires little or no knowledge of programming.

Study the problem - determine what is being asked of you and then try to find functions that can help. Do some googling and/or use "??" within R

```
Need to do sums ? rowSums, colSums, sum,
Need to do statistics ?  mean, sd, var, quantile, fivenum
Need to find missing values ?  complete.cases, is.na
Need to find values in a vector based on conditions ? subset or bracket
Need to do something to each row/column of a matrix ? Use apply
```

Performance and Benchmarking

Given these two vectors, find the element numbers in vec1 where the elements of vec2 match any element in vec1. So "b1" in vec2 matches the second element in vec1. "d3" in vec2 matches the fourth element in vec1. so we return 2 and 4

Not vectorized:

```
vec1 = c("a1", "b1", "c2", "d3")

vec2 = c("b1", "d3")

retvec = vector()
for (ii in 1:length(vec2)) {      # For each element in b try to match to a
   for (jj in 1:length(vec1)) {
      if (vec2[ii] == vec1[jj]) {
        retvec = c(retvec,jj)
      }
   }
}

retvec
[1] 2 4
```

Performance and Benchmarking

Given these two vectors, find the element numbers in vec1 where the elements of vec2 match any element in vec1. So "b1" in vec2 matches the second element in vec1. "d3" in vec2 matches the fourth element in vec1. so we return 2 and 4

Vectorized:

```
vec1 = c("a1", "b1", "c2", "d3")

vec2 = c("b1", "d3")

vec1 %in% vecb
[1] FALSE  TRUE FALSE  TRUE

is.element(vec1, vec2)
[1] FALSE  TRUE FALSE  TRUE


which(vec1 %in% vec2)   # Pretty cool huh ?
[1] 2 4
```

Performance and Benchmarking

In R you don't really have to think about allocating memory to structures before you use them. This is good and bad: Let's create a "blank" vector and then extend it to have 400000 elements which we will fill using a for loop:

```
somevec = vector()   #
for (ii in 1:400000) {
   somevec[ii] = ii
}
```

Is this any different, better, or worse than preallocating space for the 400000 elements ?

```
somevector = vector()
length(somevector) = 400000
for (ii in 1:400000) {
   somevec[ii] = ii
}
```

Performance and Benchmarking

```
func1 <- function(size=400000) {
   somevec = vector()   #
   for (ii in 1:size) {
     somevec[ii] = ii
   }
   return(somevec)
}

func2 <- function(size=400000) {
  somevec = vector()
  length(somevec) = size  # We tell R explicitly the size of the vector
  for (ii in 1:400000) {
    somevec[ii] = ii
  }
  return(somevec)
}
```

Performance and Benchmarking

```
system.time(func1())
   user  system elapsed
130.905  46.591 177.634

system.time(func2())
   user  system elapsed
  0.607   0.001   0.609
```

So preallocating memory does help this situation. Even if you don't know exactly how long the structure will be, you can always estimate it in advance and it will improve performance. But to really do this why not use the power of vectors:

```
system.time(somevec <- 1:400000)  # Much faster
   user  system elapsed
  0.000   0.000   0.001
```