# Speeding up R code

Matteo Fasiolo

# Is R slow? It depends.

R (like Python) is an interpreted language, interpretation takes time.

Suppose that R takes 0.001s to interpret content of a `for()` loop.

Whether this 0.001 sec matters or not depends on the context:

```
start <- Sys.time()
for(ii in 1:1e3){
  Sys.sleep(0.001) # Interpretation time
  1 + 1
}
end <- Sys.time()

end - start
# Time difference of 1.170159 secs
```

Without interpretation:

```
start <- Sys.time()
for(ii in 1:1e3){
  1 + 1
}
end <- Sys.time()

end - start
# Time difference of 0.01621675 secs
```

Another example:

```
start <- Sys.time()
for(ii in 1:1e3){
  Sys.sleep(0.001) # Interpretation time
  x <- rnorm(1e5)
}
end <- Sys.time()
end - start
# Time difference of 12.38026 secs
```

Without interpretation:

```
start <- Sys.time()
for(ii in 1:1e3){
  x <- rnorm(1e5)
}
end <- Sys.time()
end - start
# Time difference of 10.03441 secs
```

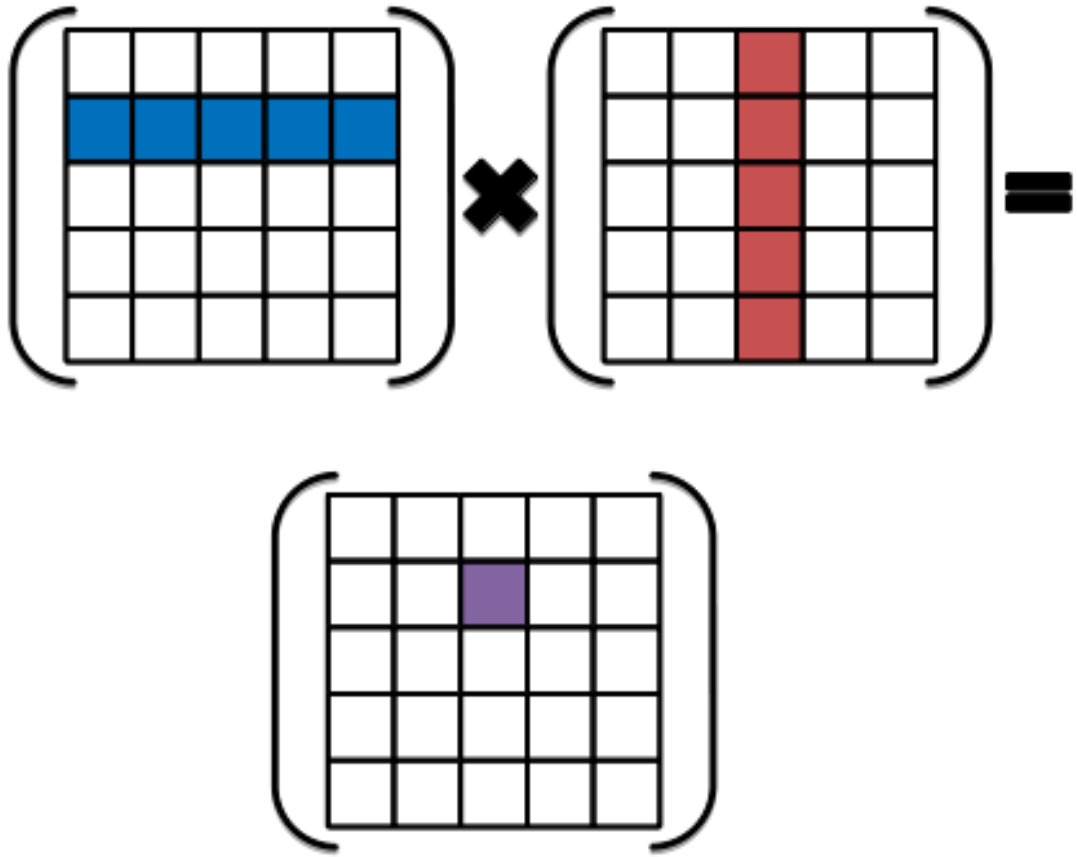
Generally speaking interpretation will be costly when:

A) you are using one or several nested `for()` loops

B) at each iteration you are doing a small computation

You need both conditions: **it is not correct to just say that `for()` loop are slow in R**!

# Matrix Multiplication in R



Perfect example of what would be slow in R.

Three nested loops, in the inner-most loop you are doing very little.

```r
mmul1 <- function(A, B){
  # Create a zero matrix C
  C <- matrix(0, nrow = nrow(A), ncol = ncol(B))
  # Loop over rows of A
  for (i in 1:nrow(A)){
    # Loop over cols of B
    for (j in 1:ncol(B)){
      # Loop over cols of A
      for (k in 1:ncol(A)){
        C[i,j] <- C[i,j] + A[i,k]*B[k,j]
      }
    }
  }
  return(C)
}
```

Generating two 1000 $\times$ 1000 random matrices:

```
A <- matrix(rnorm(1000*1000), nrow = 1000)
B <- matrix(rnorm(1000*1000), nrow = 1000)
```

Perform matrix multiplication:

```
start <- Sys.time()
C <- mmul1(A, B)
end <- Sys.time()

end - start
# Time difference of 6.435844 mins
```

Solution: use %*%, R's build-in matrix multiplication function:

```
# built-in matrix multiplication
start <- Sys.time()
C <- A%*%B
end <- Sys.time()

end - start
# Time difference of 0.5414867 secs
```

Over 700 times faster than our code!

It takes C $\approx$ 5 second (using code from lab 9).

Why?

Because R's %*% is calling C code for matrix multiplications.

It is calling a highly optimised library: Basic Linear Algebra Subprograms (BLAS).

There are many libraries: Linear Algebra PACKage (LAPACK), OpenBLAS, ...

R calls code written in C (or other fast compiled languages) for many standard task.

E.g., simulating random variables:

```
x <- rnorm(1000)
```

sorting:

```
x <- sort(x)
x[1:4]
```

```
[1] -3.351201 -3.100005 -3.003718 -2.875334
```

Try typing on the R console:

```
rnorm
# function (n, mean = 0, sd = 1)

# .Call(C_rnorm, n, mean, sd)

# <bytecode: 0x5634390979a0>

# <environment: namespace:stats>
```

`.Call` is an R function allowing you to call C/C++ code from R.

If interested in R/C++ interface see this (PhD level!) course material.

# How to speed up your R code

If you are faced with a routine task e.g.:

- vector/matrix ops: adding, multiplying, decompositions ...
- simulating random numbers, sorting, ...

Try to find whether R has a specific function for it (Google it!).

That function will often call compiled code.

**WARNING**: The CRAN repository contains over 20K packages.

You can install them by doing

```
install.packages("package_name")
```

You will find lots of useful functions, but not all of them are computationally efficient.

It's important to be clear about where the speed up comes from.

Consider again matrix multiplication.

We saw that %*% was faster than our R and C code.

The sources of the speed up are:

1) The overhead of interpreting R loops is avoided (6min -> 5sec);
2) %*% calls a highly optimised C++ library (5sec -> 0.5sec).

Focusing on 1), the process of rearranging code to avoid R loops is called **vectorisation**.

A simple example is:

```
n <- 1000
x <- numeric(n)
for(ii in 1:n){
  x[ii] <- rnorm(1)
}
```

Vectorised version:

```
x <- rnorm(n) # Much faster
```

In the second case the `for` loop has not really disappeared: it now takes place in C.

Here the speed up comes only from 1), looping in C rather than R.

Another example:

```
z <- 0
for(ii in 1:n){
  z <- z + x[ii]
}
```

Vectorised version:

```
z <- sum(x) # Much faster
```

Now we get speed up from

1) no for loop in R
2) calling efficient C code.

**Be skeptical about unjustified claims that a piece of code is fast/slow.**

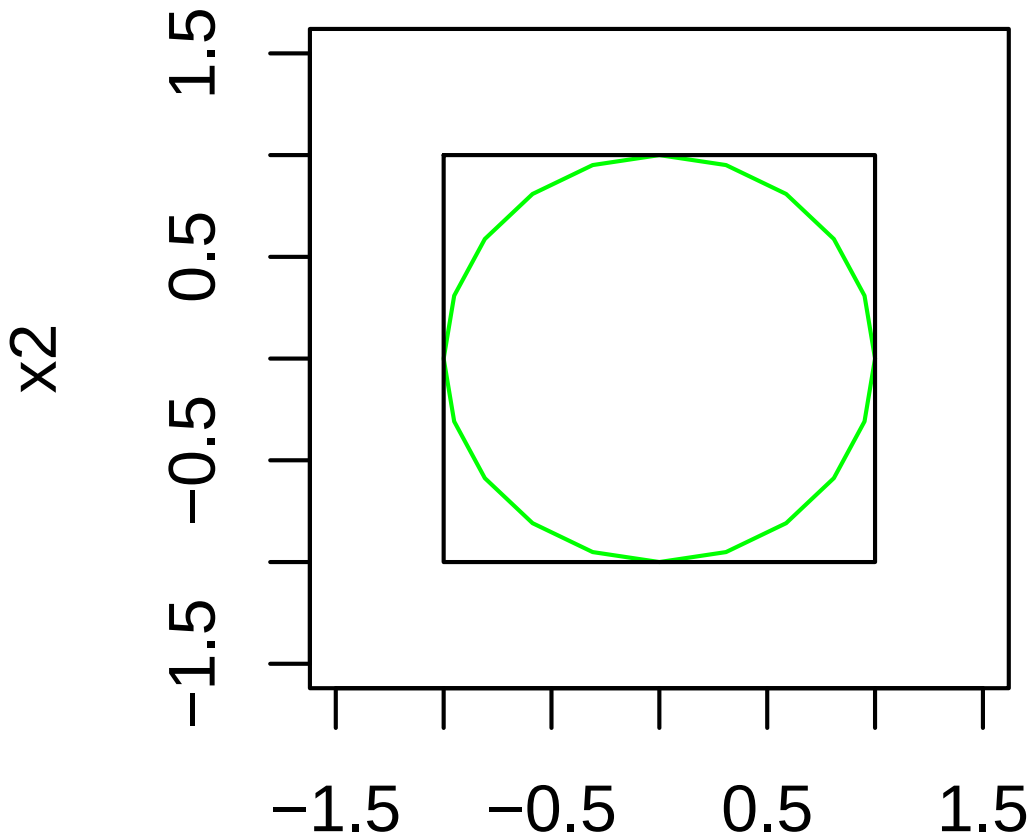
# Example on speeding up R code: estimating $\pi$

Define a two dimensional uniform random variable $X$:

$$X = (X_1, X_2) \quad \text{where} \quad X_1, X_2 \sim U(-1, 1).$$

Unit circle: a circle centered at origin with r $= 1$.

Unit square: a square centered at origin with sidelength $= 2$.



$$\mathbb{P}(X \text{ in the unit circle}) = \frac{\text{area of unit circle}}{\text{area of unit square}}.$$

Area of the unit circle $= \pi$.

Area of the unit square $= 4$.

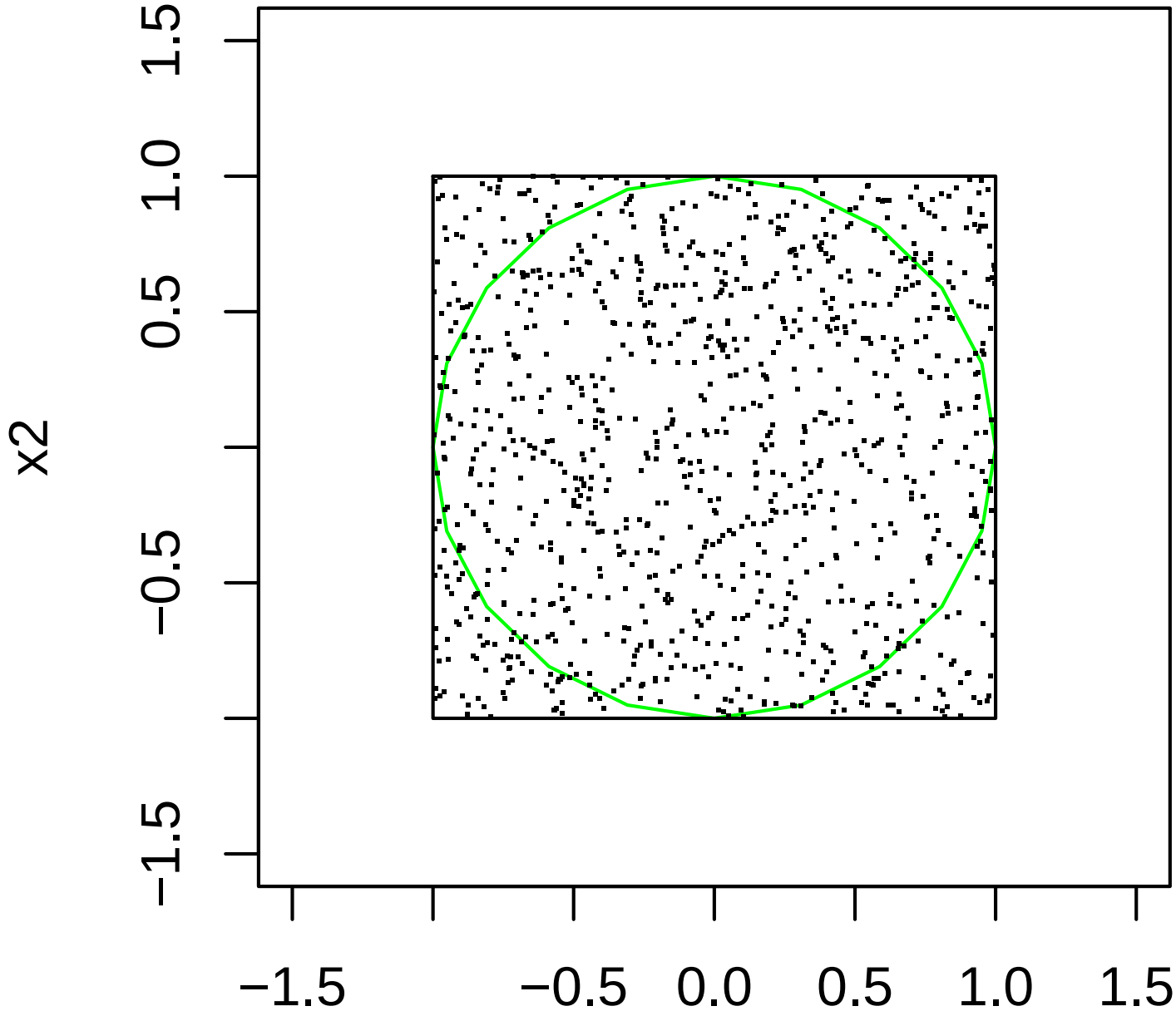
$$\mathbb{P}(X \text{ in the unit circle}) = \frac{\pi}{4}.$$

Hence $\pi = 4\mathbb{P}(X \text{ in the unit circle})$.

We can estimate $\mathbb{P}(X \text{ in the unit circle})$ via Monte Carlo.

Sample uniformly in the box $[-1, 1]^2$.

$\mathbb{P}(X \text{ in the unit circle}) \approx \frac{\#\text{samples in circle}}{\#\text{total samples}}$



# Non-vectorized Code

$X = (X_1, X_2) \in$ circle if $||X||^2 = X_1^2 + X_2^2 < 1$.

```
n <- 1e6
count <- 0
for (i in 1:n){

  x <- runif(2, -1, 1)

  if (sum(x^2) < 1){

    count <- count + 1

  }

}
count/n * 4 # (4 seconds)
```

```
[1] 3.145636
```

# Semi-vectorized Code

```
X <- matrix(runif(2*n, -1, 1), ncol = 2)

count <- 0
for (i in 1:n){

  if (sum(X[i, ]^2) < 1){

    count <- count + 1

  }

}
count/n * 4 # (1.5 seconds)
```

```
[1] 3.140932
```

# Fully-vectorized Code

```
X <- matrix(runif(2*n, -1, 1), ncol = 2)

dist <- rowSums( X^2 )

count <- sum(dist < 1)

count/n * 4 # (0.1 seconds)
```

```
[1] 3.138768
```

Ultimately we got a 40-fold speed up from:

1. Avoiding `for()` loops in R
2. Using C routine called by `rowSums`

# Conclusion

Loops (`for`, `while`...) is R are not always slow.

They are slow when you are doing very little work at each iteration.

You can speed up your code by calling specific R functions that:

1) Implement for loops in C;
2) Call highly optimised compiled code.

Step 1 is often called **vectorisation**.

# Homework 0: Practicing rowSums

The function `rowSums` sums over each row of a matrix.

```
( A <- matrix(c(1,2,3,4,5,6), 2, 3) )
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
rowSums(A)
```

```
[1]  9 12
```

```
c(sum(A[1, ]), sum(A[2, ]))
```

```
[1]  9 12
```

Read the manual of `rowSums` and play with a few more examples.

There is a similar function called `colSums` that performs column sums. Try it yourself.

# Homework 1: Pairwise Distance Calculation

Assume we have 2 matrices:

- **A** of dimension $(500 \times 784)$;
- **B** of dimension $(784 \times 800)$.

Objective: computing a $(500 \times 800)$ matrix **D** were:

$$D[i,j] = \text{Euclidean\_Distance}(A[i,], B[,j]).$$

Steps:

1. Create two matrices of the correct size filled with random observations from a **standard** normal distribution.
   - Hint: `rnorm(n,mean,std)` produces a **vector** of size `n` filled with samples from a normal distribution with `mean` as mean and `std` as standard deviation.

# Homework 2: Scalar Computation

2. Write a function called `pdist1` that
   - takes two matrices $A, B$ as inputs
   - outputs a matrix $D$, where $D_{i,j} = \mathrm{Euc\_Dist}(A_{[i,]}, B_{[,j]})$.
   - Your code should have **3 nested for loops**, i.e., you should write non-vectorized code.
   - Test the function on matrices generated in step 1.

# Homework 2 (submit): Vectorization

3. Write another function called `pdist2`, that does exactly the same thing, using only **two for loops**.
   - Hint, consider the vectorised `dist` function from last week.
4. Write a function `pdist3`, using **only one for loop**.
   - Hint, you can compute the whole $i$-th row in $D$ by using:

$$||\mathbf{x} - \mathbf{y}|| = ((\mathbf{x} - \mathbf{y})^T(\mathbf{x} - \mathbf{y}))^{1/2} = (\mathbf{x}^T\mathbf{x} - 2\mathbf{x}^T\mathbf{y} + \mathbf{y}^T\mathbf{y})^{1/2}.$$

   - If $\mathbf{x} = A[i,]$ and $\mathbf{y} = B[,j]$, you can compute

$$D[i,] = \{||A[i,]-B[,1]||, ||A[i,]-B[,2]||, \cdots, ||A[i,]-B[,800]||\}$$

   without any `for` loop (hence only the loop over $i$ is left).

# Homework 3 (Challenge): Vectorization

5. Write another function called `pdist4`, that does exactly the same thing **without using any for loop**.
   - You should be able to do this using only `sum`, `t()` and `rowSums/colSums`.
   - Express $D$ by matrix algebra using $A$ and $B$.
   - $D = \sqrt{a\mathbf{1}_b^\top + \mathbf{1}_a b^\top - 2AB}$, where
     - `a := rowSums(A^2)`, `b := colSums(B^2)`
     - $\mathbf{1}_a$ is a column vector with `nrow(A)` elements and each element equal to 1
     - $\mathbf{1}_b$ is similar but has `ncol(B)` elements.

# Homework 3 (Challenge): Vectoriz.

6. Test all above functions using the matrices you generated in the first step:
   - How long does it take them to run?
   - Do you get the correct result?