

MAS115: R programming

Lecture 6: Plotting and code complexity

Sofia Karadimitriou

The University of Sheffield
School of Mathematics and Statistics

Aims

How to use the extensive set of graphics functions in **R**:

- ▶ The `plot()` function
- ▶ Adding information to the plot
- ▶ Using plots well

A short introduction to coding efficiently:

- ▶ Prime numbers and Erasthones sieve

Plotting in R

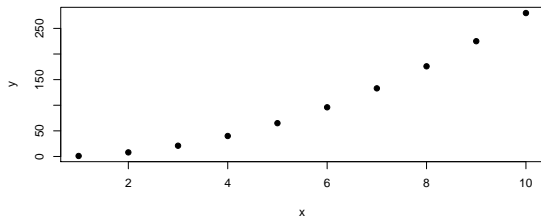
- ▶ Plotting is one of the great strengths of *R*
- ▶ The workhorse plotcommand in *R* is `plot()`. It is used as follows

```
x <- 1:10
```

```
y <- 3*x^2 - 2*x
```

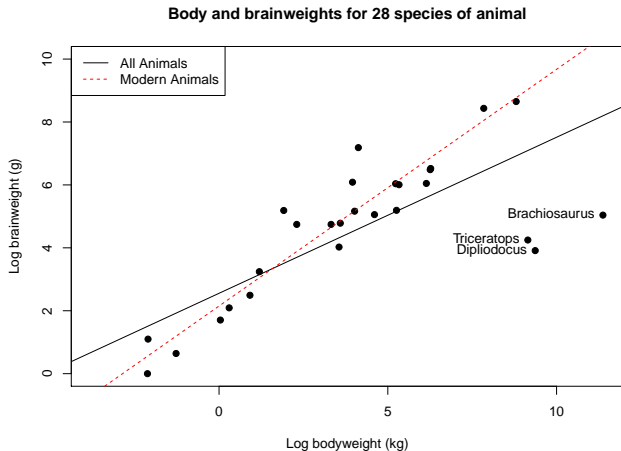
```
plot(x,y, pch = 19)
```

- ▶ Can then build up the plot with new lines/points/legend



Plotting in R

From these basic commands you can quickly create something really quite complicated



When to use plots

- ▶ Plots can sometimes represent ideas or data much more easily than words
- ▶ Visual illustrations can be used alongside text to help the reader understand new concepts
- ▶ Can represent conclusions from data analysis
- ▶ Don't just use them for the sake of it
- ▶ Be selective in what is the best plot to use and how to present it

How to include them in a report

- ▶ Every plot in a report must add something to understanding
- ▶ It must be referenced in the text
- ▶ Must be clear what the plot shows — use a caption in \LaTeX to explain what the reader should get from it.

Coding efficiently

Prime Numbers and Erasthenes Sieve

Finding all prime numbers smaller than M

A prime number is

- ▶ A natural number greater than 1
- ▶ Has no positive divisors other than 1 and itself

Suppose we want to find all the prime numbers smaller than M ?

Question: How can we code this? Does it matter how we do it?

Finding all prime numbers smaller than M

A prime number is

- ▶ A natural number greater than 1
- ▶ Has no positive divisors other than 1 and itself

Suppose we want to find all the prime numbers smaller than M ?

Question: How can we code this? Does it matter how we do it?

Answer: Many ways and yes it does matter once M becomes large

A first attempt

Consider every number $n = 1, \dots, M$ in turn

- ▶ Check if n is prime by dividing by all numbers $i = 1, \dots, n$
- ▶ If any factors found then composite, else prime

Pseudo-Code

```
INITIALIZE prime <- TRUE
```

```
FOR (i in 2:(n-1))
```

```
  IF (i divides n)
```

```
    THEN SET prime <- FALSE and break loop
```

```
  ENDIF
```

```
ENDFOR
```

```
RETURN prime
```

Need to modify slightly for the special case when $n = 2$

A first attempt — R code

I wrote some R code to check and number n individually as follows

R Code 1

```
# Inputs: n number to check
# Outputs: TRUE if found to be prime
is.prime.slow <- function(n) {
  # Deal with special cases first
  if(n == 2) return(TRUE)
  # Otherwise try dividing it by all numbers up to n
  for(i in 2:(n-1)) {
    if((n %% i) == 0) {
      return(FALSE)
    }
  }
  return(TRUE)
}
```

A first attempt - R code

We now have a function which can check whether an input number n is prime so we can create a wrapper to check all the numbers $n = 2, \dots, M$

Pseudo-Code

```
INITIALIZE primes as empty
```

```
FOR (i in 2:M)
```

```
    CHECK i primeness using is.prime.slow() function
```

```
    IF (i is prime)
```

```
        THEN ADD i to list of primes
```

```
    ENDIF
```

```
ENDFOR
```

```
RETURN primes
```

A first attempt - R code

```
# Finds all the prime numbers less than a certain number
# Inputs: M find all prime numbers below this number
# Outputs: vector of all primes below M
findprimes1 <- function(M) {
  primes <- c()
  for(i in 2:M) {
    if(is.prime.slow(i)) primes <- c(primes, i)
  }
  primes
}

> findprimes1(20)
[1]  2  3  5  7 11 13 17 19
```

Speed of our first attempt code

We can check how long our code takes to run by using the `system.time()` function

```
> system.time(p1 <- findprimes1(1000))
  user  system elapsed
 0.11   0.00   0.11
> system.time(p1 <- findprimes1(10000))
  user  system elapsed
 7.730  0.030  7.783
```

We haven't even chosen a very large M and already it's taking ages. This is a real problem if we want to choose $M = 10,000,000$.

Rough theory of algorithmic complexity - worst cases

is.prime.slow Consider the worst case where we have to divide each individual n by all of the numbers $i = 2, \dots, n - 1$. Then the total number of operation we will have to perform is

$$\begin{aligned} \text{Operations} &= 0 + 1 + 2 + 3 + \dots + (M - 2) = \sum_{n=2}^M (n - 2) \\ &\approx \frac{M(M - 1)}{2} \quad [\text{summing a AP}] \\ &= O(M^2) \quad [\text{i.e. scales with } M^2] \end{aligned}$$

This is very slow

An improvement - Trial 2

There is a lot of redundancy in our coding:

- ▶ To check if n is prime we don't need to divide by all numbers $i = 1, \dots, n$ but only need check for $i = 1, \dots, \lfloor \sqrt{n} \rfloor$

Pseudo-Code

```
INITIALIZE prime <- TRUE
```

```
FOR (i in 2:sqrt(n))
```

```
  IF (i divides n)
```

```
    THEN SET prime <- FALSE and break loop
```

```
  ENDFOR
```

```
ENDFOR
```

```
RETURN prime
```

Need to modify slightly for the special case when $n = 2$ and 3

An improvement — R code

Can write this as a function in R too

R Code 2

```
# Inputs: n number to check
# Outputs: TRUE is found to be prime
is.prime.fast <- function(n) {
  # Deal with special cases first
  if((n == 2) || (n == 3)) return(TRUE)
  # Otherwise try dividing it by numbers
  for(i in 2:floor(sqrt(n))) {
    if((n %% i) == 0) {
      return(FALSE)
    }
  }
  return(TRUE)
}
```


A second attempt - R code

We now have a more efficient function to check whether an input number n is prime so we can use this in our wrapper instead

```
# Effort 2 at finding prime numbers less than M
# Inputs: M find all prime numbers below this number
# Outputs: vector of all primes below M
findprimes2 <- function(M) {
  primes <- c()
  for(i in 2:M) {
    if(is.prime.fast(i)) primes <- c(primes, i)
  }
  primes
}

> findprimes2(20)
[1]  2  3  5  7 11 13 17 19
```

Speed of our second attempt code

We can check how much faster this new code runs

```
> system.time(p2 <- findprimes2(1000))
  user  system elapsed
0.010   0.000   0.011
> system.time(p2 <- findprimes2(10000))
  user  system elapsed
0.220   0.000   0.217
> system.time(p2 <- findprimes2(100000))
  user  system elapsed
4.280   0.010   4.305
> system.time(p2 <- findprimes2(1000000))
  user  system elapsed
109.590  0.850 110.833
```

This is a lot faster but we're still going to struggle if we want a really large M e.g. $M = 10,000,000$.

Rough theory of algorithmic complexity - worst cases

`is.prime.fast` Here the worst case is we have to divide each individual n by all of the numbers $i = 2, \dots, \lfloor \sqrt{n} \rfloor$. Then the total number of operations would be

$$\begin{aligned} \text{Operations} &= \underbrace{\sqrt{2} + \sqrt{3}}_{\geq \sqrt{1}=1} + \underbrace{\sqrt{4} + \sqrt{5} + \sqrt{6} + \sqrt{7} + \sqrt{8}}_{\geq \sqrt{4}=2} + \sqrt{9} + \dots + \sqrt{M} \\ &> \sum_{i=1}^{\sqrt{M}} i(2i+1) \quad \{ \text{since } (i+1)^2 - i^2 = 2i+1 \} \\ &\approx 2 \sum_{i=1}^{\sqrt{M}} i^2 \\ &= 2 \frac{2M^{3/2} + 3M + \sqrt{M}}{6} \\ &= O\left(M^{\frac{3}{2}}\right) \quad \text{Better but still not great} \end{aligned}$$

Erasthones Sieve

Erastothenes Sieve

To find all the prime numbers less than or equal to a given integer M :

1. Create a list of consecutive integers $(2, 3, 4, \dots, M)$.
2. Initially, let p equal 2, the first prime number.
3. Starting from p , mark each multiples of p : $2p, 3p, 4p$, etc.; note that some of them may have already been marked.
4. Find the first number greater than p in the list that is not marked. If no such number, stop. Otherwise, let p now equal this number (the next prime), and repeat from step 3.

As a refinement, it is sufficient to mark the numbers in step 3 starting from p^2 , as all the smaller multiples of p will have already been marked at that point. This means that the algorithm is allowed to terminate in step 4 when p^2 is greater than M .

Erasthoteles Sieve - Illustration

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Erasthones Sieve - Pseudo-code

```
INITIALIZE unsieved as 2:M
INITIALIZE sieved as empty
INITIALIZE p = 0

WHILE ( $p^2 < M$ )
  SET p to be first element in unsieved
  REMOVE all multiples of p from unsieved
  APPEND p to sieved
ENDWHILE

SET primes = concatenate(sieved, unsieved)
RETURN primes
```

Erasthones Sieve - R code

```
primesieve <- function(M)
{
  unsieved <- 2:M
  sieved <- c()
  p <- 0
  while (p^2 < M) {
    p <- unsieved[1]
    unsieved <- unsieved[unsieved%%p != 0]
    sieved <- c(sieved, p)
  }
  return(c(sieved, unsieved))
}

> primesieve(20)
[1]  2  3  5  7 11 13 17 19
```


Speed of Eratosthenes Sieve

How much faster is this algorithm?

```
> system.time(p3 <- primesieve(1000))
  user  system elapsed
0.010   0.000   0.001
> system.time(p3 <- primesieve(10000))
  user  system elapsed
0.000   0.000   0.002
> system.time(p3 <- primesieve(100000))
  user  system elapsed
0.030   0.000   0.029
> system.time(p3 <- primesieve(1000000))
  user  system elapsed
0.630   0.000   0.634
> system.time(p3 <- primesieve(10000000))
  user  system elapsed
10.610   0.340  10.988
```

Speed of the algorithms

How much difference did changing the way we coded our algorithm make?

Method	Time (s) for M				
	1000	10000	100000	1000000	10000000
is.prime.slow	0.11	7.783	N/A	N/A	N/A
is.prime.fast	0.011	0.217	4.305	110.8	N/A
Erasthotes	0.001	0.002	0.029	0.634	10.99

This a huge difference - it makes sense to think about writing your code efficiently

Rough theory of algorithmic complexity - worst cases

Erasthones Sieve With Erasthones Sieve the number of operations we need to perform is

$$O(M \log \log M)$$

This is much much faster than the other two algorithms

Conclusion

- ▶ The way that you code your algorithms up can make a huge difference to the speed that they run.
- ▶ Good to try and be as efficient as possible.
- ▶ More important to be correct (especially when starting out with programming)
- ▶ Efficiency often only makes a difference when you are working with really big numbers.