Scientific Programming: Assignment 1

University of Cambridge

Henrik Åhl

November 1, 2016

1 Introduction

This is an assignment report in connection to the *Scientific Programming with* `R` module in the Computational Biology course at the University of Cambridge, Michelmas term 2016. All related code is as of November 1, 2016 available on `https://github.com/supersubscript/compbio/tree/master/src/sp_assignments/assignment_1`, or available per request by contacting hpa22@cam.ac.uk. Likewise, the corresponding assignment can be found on `https://github.com/sje30/rpc2016/tree/master/assigns`.

2 Solutions

The assignment consists of three sections, related to various parts of scientific programming. The first section deals with geometric plotting, the second with developing an algorithm for solving cryptarithms, and the third with data manipulation and visualisation.

2.1 Curved Squares

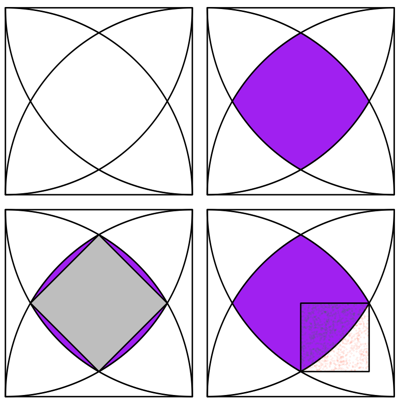

Figure 1: The arcs and corresponding geometries, as well as the area estimate.

The analytic area can be derived by tracing the two upper quarter circles to their intersection and realising that a triangle based in the two bottom corners and this point must be equilateral. We can then use two sides of the triangle to form a circle sector, which must have area $\pi/6$. Likewise, we can use the triangle to extract a circle segment, which correspondingly will have area $\pi/6 - \sin(\pi/3)/2$. Knowing that a full quarter circle is here of area $pi/4$, we can easily deduce that the area surrounding the shape must be four times the area of the circle quartile minus the segment and sector. *Summa sumarum*, we find that the enclosed area is

$$1 - (\pi/4 - \pi/6 - (\pi/6 - \sin(\pi/3)/2)) \approx 0.315.$$

```
The area of the square contained is:     0.2678984
The estimated area of the shape is:  0.3150979
```

curved_squares_out.txt

2.2 Cryptarithms

This exercise was solved using matrix manipulations, inserting character columns from a letter-value permutation matrix, parsing and evaluating. It has an approximate runtime on the order of minutes.

```
A B C D E F G H I
1 7 2 3 4 5 6 9 0
1 7 4 6 8 2 5 9 3
1 8 2 3 6 5 4 9 0
3 8 2 7 6 1 4 9 0
###################
s e n d m o r y
9 5 6 7 1 0 8 2
###################
s n o w r a i l e t
1 2 3 6 9 5 4 0 7 8
1 2 4 6 9 5 3 0 7 8
1 3 2 4 9 5 6 0 8 7
1 3 6 4 9 5 2 0 8 7
1 4 5 9 8 7 6 0 2 3
1 4 6 9 8 7 5 0 2 3
1 9 2 5 8 7 3 0 6 4
1 9 3 5 8 7 2 0 6 4
###################
o n e t w h r l v
3 9 1 8 0 4 6 7 2
###################
```

cryptarithms_out.txt

2.3 Word Processing

In the word processing part, *y* is treated as a vowel (as it should be). The dictionary does not distinguish between capitalised letters or not, so in particular in the palindrome exercise, the palindromes will be replicated if variants with different capitalisations exist.

```
Number of words:   99171
Number of ascii words:   244
Number of non-v words:   254
Number of words with apostrophes:  26141
The most frequent bigram is 's ' with a frequency of 0.03208276.
```

word_processing_out.txt

```
B B's BB's  BC's  BLT's BM's
BMW BMW's BS's  Bk  Bk's  Blvd
Br  Br's  C C's CD's  CFC's
CPR's CRT's CST's CT's  Cd  Cd's
Cf  Cf's  Cl  Cl's  Cm  Cm's
Cr  Cr's  Cs  D D's DC's
DD's  DDS's DP's  Dr  F F's
FDR FDR's FM's  Fm  Fm's  Fr
Fr's  G G's GHQ's GMT's GNP's
GP's  Gd  Gd's  H H's HF's
HP's  HQ's  HSBC  HSBC's  HTML's  Hf
Hf's  Hg  Hg's  Hz  Hz's  J
J's JFK JFK's Jr  Jr's  K
K's KFC KFC's KKK's Kr  Kr's
L L's LBJ LBJ's LCD's LPN's
LSD's Ln  Lr  Lt  Ltd M
M's MB's  MD's  MGM MGM's MHz
MP's  MS's  MSG's MST's MT's  MVP's
Mb  Md  Md's  Mg  Mg's  Mn
Mn's  Mr  Mr's  Mrs Ms  Mt
N N's NW's  Nb  Nb's  Nd
Nd's  Np  Np's  P P's PBS's
PC's  PM's  PMS's PS's  PST's PVC's
Pb  Pb's  Pd  Pd's  PhD PhD's
Pl  Pm  Pm's  Pt  Pt's  Q
R R's RN's  RV's  Rb  Rb's
Rd  Rh  Rh's  Rn  Rn's  Rx
S S's SC's  SW's  Sb  Sb's
Sc  Sc's  Sgt Sm  Sm's  Sn
Sn's  Sq  Sr  Sr's  St  T
T's TB's  TLC's TNT's TV's  Tb
Tb's  Tc  Tc's  Th  Th's  Tl
Tl's  Tm  Tm's  V V's VCR's
VD's  VHF's VLF's W W's WWW's
Wm  Wm's  X X's XL's  Z
Z's Zn  Zn's  Zr  Zr's  b
brr c cs  d dB  f
g gs  h h'm j k
kHz kW  kc  ks  l ls
m ms  n nth p pH
pj's  psst  q r rs  s
sh  t ts  v vs  w
x z
```

word_processing_noVowelWords.txt

```
A Ada Ala Ana Anna Ara
Ava B Bib Bob C D
E Eve F G Gog H
Hannah I J K L Laval
M MGM Malayalam N Nan O
Ono Otto P Q R S
S's Salas T Tet Tevet Tut
U V W X Y Z
a aha b bib bob boob
c civic d dad deed deified
did dud e eke ere eve
ewe eye f g gag gig
h hah huh i j k
kayak kook l level m ma'am
madam minim mom mum n non
noon nun o oho p pap
peep pep pip poop pop pup
q r radar redder refer rotor
s sagas sees sexes shahs sis
solos sos stats t tat tenet
tit toot tot u v w
wow x y z
```

word_processing_palindromes.txt

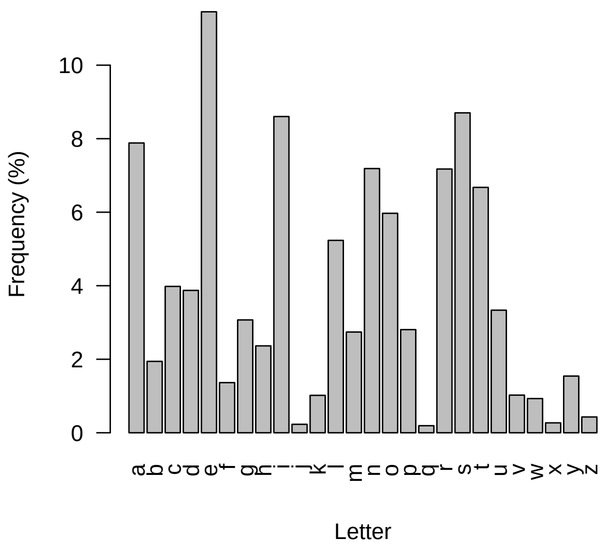

Figure 2: Letter frequencies for the Ubuntu English (UK) dictionary.

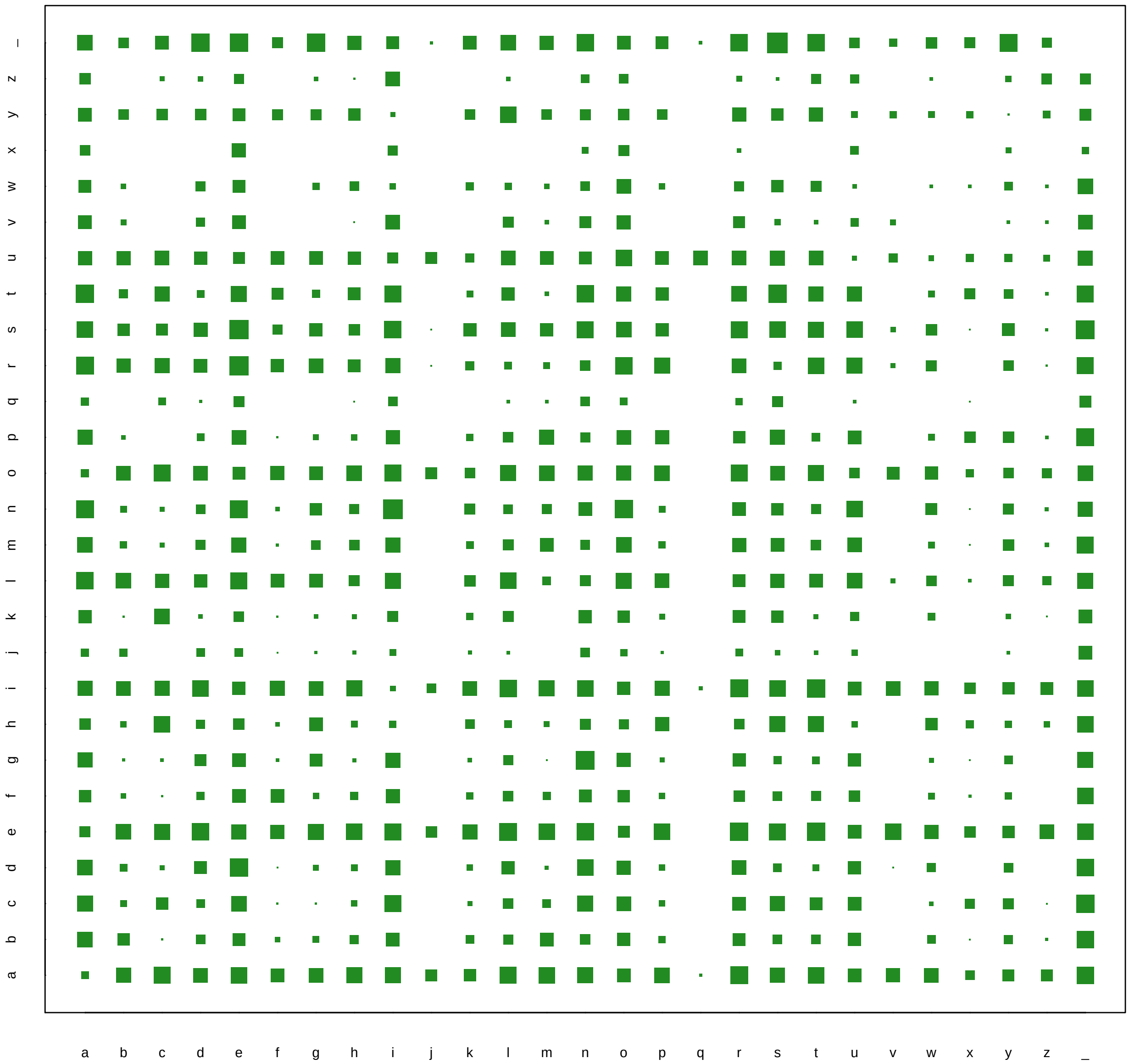

Figure 3: Bigram for the letters contained in the dictionary. Square areas are proportional to the logarithm of the number of occurences. The horizontal row marks the first character, the vertical the subsequent. (You said frequency in the exercise, but I think log(occ) makes more sense for visualisation.)

Code

```
sink("curved_squares_out.txt")

### Take off axes
pdf("curved_squares.pdf", width = 3, height = 3)
par(bty = 'n',xaxt = 'n',yaxt = 'n')
par(mfrow = c(2,2), mar = c(0,0,0,0) + 0.0)

### Define the functions, upper-to-lower, lower-to-upper
x = seq(0, 1, 10 ^ -4)
lowerUTL = function(x){-sqrt(1 - (x-1)^2) + 1}
upperUTL = function(x){sqrt(1 - x^2)}
lowerLTU = function(x){-sqrt(1 - x^2) + 1}
upperLTU = function(x){sqrt(1 - (x-1)^2)}

####### FIGURE 1
### Draw the arcs
plot(x, lowerUTL(x), type='l', ylab='', xlab='')
lines(x, lowerLTU(x))
lines(x, upperLTU(x))
lines(x, upperUTL(x))
rect(0,0,1,1)

### Indices we need
intersectionIndex = which(abs(lowerUTL(x) - upperLTU(x)) < 10 ^ -4)
midpointIndex = length(x)/2

####### FIGURE 2
### Draw the arcs
plot(x, lowerUTL(x), type='l', ylab='', xlab='')
lines(x, lowerLTU(x))
lines(x, upperLTU(x))
lines(x, upperUTL(x))
rect(0,0,1,1)

### Le polygon
polygon(c(x[seq(intersectionIndex, length(x) - intersectionIndex)],
  rev(x[seq(intersectionIndex,length(x) - intersectionIndex)])),
  c(upperLTU(x[seq(intersectionIndex, length(x) - midpointIndex)]),
  upperUTL(x[seq(midpointIndex,length(x) - intersectionIndex)]),
  rev(lowerLTU(x[seq(midpointIndex,length(x) - intersectionIndex)])),
  rev(lowerUTL(x[seq(intersectionIndex, midpointIndex)]))), col='purple')

####### FIGURE 3
### Draw the arcs
plot(x, lowerUTL(x), type = 'l', ylab = '', xlab = '')
lines(x, lowerLTU(x))
lines(x, upperLTU(x))
lines(x, upperUTL(x))
rect(0,0,1,1)
```

```
### Le polygon
polygon(c(x[seq(intersectionIndex, length(x) - intersectionIndex)],
  rev(x[seq(intersectionIndex,length(x) - intersectionIndex)])),
  c(upperLTU(x[seq(intersectionIndex, length(x) - midpointIndex)]),
  upperUTL(x[seq(midpointIndex,length(x) - intersectionIndex)]),
  rev(lowerLTU(x[seq(midpointIndex,length(x) - intersectionIndex)])),
  rev(lowerUTL(x[seq(intersectionIndex, midpointIndex)]))), col='purple')

### Le cube
y1 =  x + upperLTU(x[intersectionIndex]) - x[intersectionIndex]
y2 = -x + upperLTU(x[intersectionIndex]) - x[intersectionIndex] + 1
y3 =  x + lowerLTU(.5) - .5
y4 = -x + lowerLTU(.5) + .5

area.square = (.5 - x[intersectionIndex])^2 +
  (upperLTU(.5) - upperLTU(x[intersectionIndex]))^2
cat("The area of the square contained is:\t", area.square, "\n")

polygon(c(x[seq(intersectionIndex, length(x) - intersectionIndex)],
  rev(x[seq(intersectionIndex,length(x) - intersectionIndex)])),
  c(y1[seq(intersectionIndex, length(x) - midpointIndex)],
  y2[seq(midpointIndex,length(x) - intersectionIndex)],
  rev(y3[seq(midpointIndex,length(x) - intersectionIndex)]),
  rev(y4[seq(intersectionIndex,midpointIndex)])), col='gray')

###### FIGURE 4
### Draw the arcs
plot(x, lowerUTL(x), type='l', ylab='', xlab='')
lines(x, lowerLTU(x))
lines(x, upperLTU(x))
lines(x, upperUTL(x))
rect(0,0,1,1)

### Le polygon
polygon(c(x[seq(intersectionIndex, length(x) - intersectionIndex)],
  rev(x[seq(intersectionIndex,length(x) - intersectionIndex)])),
  c(upperLTU(x[seq(intersectionIndex, length(x) - midpointIndex)]),
  upperUTL(x[seq(midpointIndex,length(x) - intersectionIndex)]),
  rev(lowerLTU(x[seq(midpointIndex,length(x) - intersectionIndex)])),
  rev(lowerUTL(x[seq(intersectionIndex, midpointIndex)]))), col='purple')

### Naive Monte-Carlo (box around purple shape)
### Note: Our problem is symmetric, so we can limit ourselves to a quartile.
nrTries = 1e7
xHit = runif(nrTries, .5, x[length(x) - intersectionIndex])
yHit = runif(nrTries, x[intersectionIndex], .5)
hits = which(yHit > lowerLTU(xHit))
misses = which(yHit <= lowerLTU(xHit))

### Plot every 100th point.
```

```
points(xHit[hits[seq(1,length(hits),10000)]], yHit[hits[seq(1,length(hits),10000)]],
  type = 'p', cex = .01, col = 'forestgreen')
points(xHit[misses[seq(1,length(misses),10000)]], yHit[misses[seq(1,length(misses), 10000)]],
  type = 'p', cex = .01, col = 'tomato')
rect(.5,x[intersectionIndex], x[length(x) - intersectionIndex], .5)

area.estimate = length(hits)/nrTries *
  (x[length(x) - intersectionIndex] - x[intersectionIndex])^2
cat("The estimated area of the shape is:\t", area.estimate,"\n")

### Kill off graphical output
dev.off()
```

../../../src/sp_assignments/assignment_1/curved_squares.R

```
sink("cryptarithms_out.txt")

####################
### FUNCTIONS
####################

### Returns possible permutations of input
permutations <- function(n) {
  if (n == 1)
  {return(matrix(1))} else
  {
    sp <- permutations(n - 1)
    p <- nrow(sp)
    A <- matrix(nrow = n * p, ncol = n)
    for (i in 1:n)
    {
      A[(i - 1) * p + 1:p,] <- cbind(i, sp + (sp >= i))
    }
    return(A)
  }
}
getUnique = function(x)
{
  if (x == 0) {
    return(1)
  }
  x * getUnique(x - 1)
}

### Which letters can't be 0?
getFirstCharacters = function(string)
{
  string = gsub("[[:punct:]]", " ", string) # remove some symbols
  string = gsub("\\s+"," ", string) # remove extra whitespaces
  string = strsplit(string," ")[[1]]
```

```
  unique(sapply(string, function(x)
    substring(x,1,1)))
}

### Which are the unique letters?
uniqueLetters = function(inString)
{
  inString = strsplit(inString, "")[[1]]
  inString = inString[!inString %in% c(" ", "+", "*", "-", "=", "/", "&")]
  inString = unique(inString)
}

####################
### MAIN
####################

### Takes in string and evaluates the corresponding cryptarithm.
### Note: Several statements are separated by '&'.
### Note: Only handles base 10 at the moment.
### Note: Very inefficient; struggles with larger strings (=> larger matrices)
crypta = function(string.in) {

  ### Get the number of row duplicates we will have
  m = permutations(10) - 1
  inString = string.in
  letters = uniqueLetters(inString)
  firstCharacters = getFirstCharacters(inString)
  inString = strsplit(inString, "")[[1]]
  colnames(m) = c(letters, rep("NA", 10 - length(letters)))
  if(length(letters) != 10)
  {
    m = m[, -grep("NA", colnames(m))] # Take off edge.
  }
  m = m[seq(1,nrow(m),getUnique(10 - length(letters))),] # Remove duplicates.
  for (ii in firstCharacters) {
    m = m[-which(m[, ii] == 0),] # Kill off those w 0's at beginning.
  }
  inString = inString[!inString %in% c(" ")] # Take out essentials
  strings = matrix(
    inString, ncol = length(inString), nrow = nrow(m), byrow = TRUE
  )
  colnames(strings) = strings[1,]

  ### Adjust our columns correspondingly.
  count = 1
  for (ii in colnames(strings)) {
    if (ii %in% colnames(m)) {
      strings[, count] = m[,ii]
    }
```

```
    else if (ii == "=")
    {
      strings[, count] = rep("==", nrow(m))
    }
    count = count + 1
  }

  ### Add strings back together
  strings = matrix(do.call(paste0, as.data.frame(strings)))
  output = sapply(strings, function(x)
    eval(parse(text = x)))
  print(m[which(output == TRUE),])
  cat("################################","\n")
}

### Run program with our different setups.
crypta("AB * C = DE & DE + FG = HI")
crypta("send + more = money")
crypta("snow + rain = sleet")
crypta("one + two + two + three + three = eleven")
```

../../../src/sp_assignments/assignment_1/cryptarithms.R

```
rm(list = ls())
sink("word_processing_out.txt")
### 1 Number of words
words = read.table(
  "../../../src/sp_assignments/assignment_1/usr-share-dict-words")
words = unlist(words)
#words = tolower(words)
#words = unique(words)
cat("Number of words:\t", length(words), "\n")

### 2 Ascii
words.ascii = words[-grep("[^\x21-\x7e]", words)]
words.ascii = as.character(words.ascii)
cat("Number of ascii words:\t", length(words) - length(words.ascii),
    "\n")

### 3 Vowels
noVowelWords = words.ascii[-grep("[aeiouy]", words.ascii,
  ignore.case = TRUE)]
cat("Number of non-v words:\t", length(noVowelWords), "\n")
write(noVowelWords, "word_processing_noVowelWords.txt", sep="\t", ncolumns =
     6)

### 4 Palindromes
reverseString = function(x) paste(substring(x, nchar(x):1,
  nchar(x):1), collapse = "")
palindromes = words.ascii[which(sapply(words.ascii,
  function(x) tolower(x) == tolower(reverseString(x))) == TRUE)]
```

```
write(palindromes, "word_processing_palindromes.txt", sep="\t", ncolumns = 6)

### 5 Apostrophes
no.apostrophes = grep("[']", words.ascii, value = TRUE,
  invert = TRUE)
cat("Number of words with apostrophes:\t", length(words.ascii)
    - length(no.apostrophes), "\n")

### 6 Probability distribution
letter.frequency = paste(no.apostrophes, collapse = "")
letter.frequency = summary(factor(tolower(strsplit(
  letter.frequency, "")[[1]])))
letter.frequency = letter.frequency / sum(letter.frequency)
pdf("word_processing_letter_frequency.pdf", width = 5, height = 5)
par(mfrow = c(1, 1), las = 2)
barplot(letter.frequency * 100, xlab = 'Letter',
  names.arg = letters[1:26], ylab="Frequency (%)")
x = dev.off()

### 7 Bigram
bigram.matrix = matrix(0, length(letters) + 1, length(letters) + 1)
bigram.data   = tolower(paste(c("", no.apostrophes, ""),
  collapse = " "))
colnames(bigram.matrix) = c(letters, " ")
rownames(bigram.matrix) = c(letters, " ")
pairs = substring(bigram.data, 1:(nchar(bigram.data) - 1),
  2:nchar(bigram.data))
pairs.data = table(pairs)

cat("The most frequent bigram is \'",
  names(pairs.data)[which(pairs.data == max(pairs.data))],
  "\' with a frequency of ",
  pairs.data[which(pairs.data == max(pairs.data))]
  / sum(pairs.data), ".", sep = "")

for(ii in names(pairs.data))
{
  chars = strsplit(ii,"")[[1]]
  bigram.matrix[chars[1], chars[2]] =
    bigram.matrix[chars[1], chars[2]]  + pairs.data[ii]
}

pdf("word_processing_bigram.pdf", width = 10, height = 10)
par(mfrow=c(1,1))
nrs = as.numeric(factor(rownames(bigram.matrix)))
plot(0, 0, type='n', xlim = c(1,27), ylim = c(1,27),
     xaxt = 'n', yaxt = 'n', xlab = '', ylab = '')
for(ii in nrs)
{
  for(jj in nrs)
```

```
  {
    points(ii, jj, cex = 0.22 * max(0, log(bigram.matrix[ii,jj])),
           pch = 15, col='forestgreen')
  }
}
axis(1, at=1:27, labels=c(letters[1:26], '_'),tck= 0,cex.axis = .7)
axis(2, at=1:27, labels=c(letters[1:26], '_'),tck= 0,cex.axis = .7)
x = dev.off()
```

../../../src/sp_assignments/assignment_1/word_processing.R