YSC4204: STATISTICAL COMPUTING
PROBLEM SET 3

EAST: SEAN SAITO, JOHN REID, HRISHI OLICKEL, CHONG WOON HAN

1. Problem 1

(1) (a) **Option 1:** R-devel
Link: https://stat.ethz.ch/mailman/listinfo/r-devel
The web page describes itself as a list that "is intended for questions and discussion about code development in R. Questions likely to prompt discussion unintelligible to non-programmers or topics that are too technical for R-help's audience should go to R-devel". Thus, this seems to be a suitable place to ask about the technicalities behind the decision to use the Ahrens-Dieter algorithm.

Option 2: R-package-devel
Link: https://stat.ethz.ch/mailman/listinfo/r-package-devel
This is the mailing list to all package developers or R to get "help about package development in R".

(b) Here is a draft of our question:
In src/nmath/sexp.c, the Ahrens-Dieter algorithm is being used to generate random numbers from an exponential distribution.
I am curious as to why this algorithm is favored over the inverse transform method, which is the straightforward method of producing random numbers from a particular probability distribution function, used as the default for normal random numbers. Since the cumulative distribution function of the exponential function is $F_X(x) = 1 - e^{-\lambda x}$, one would use its inverse, $F_X^- 1(u) = -(1/\lambda)\log(1-u)$, to produce samples. In R, this is as easy as

```
-log(runif(n)) / lambda
```

.

I have tried to hypothesize some benefits of the Ahrens-Dieter algorithm myself. However there does not seem to be any asymptotic time benefits (the Ahrens-Dieter algorithm involves a single for-loop), nor any obvious improvements in precision.

2. Problem 2

(2) We use the following R code to implement Rizzo's functions to generate bivariate normal distributions:

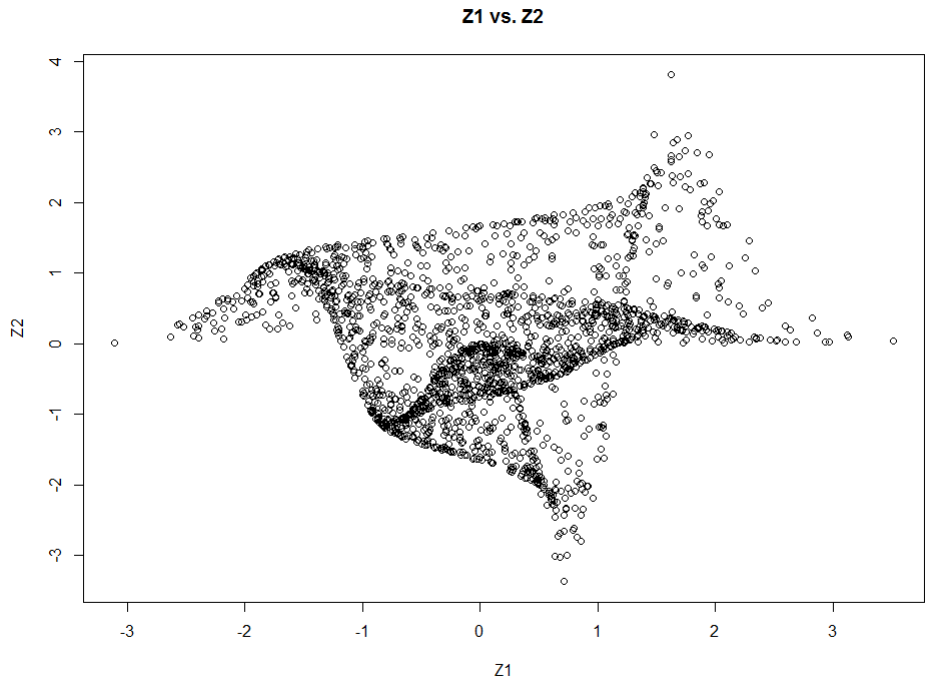
```
# Problem 2
# Assessing a proposed alternative to the Box-Muller method

library(MVN)
library(mvtnorm)
set.seed(0)

# Helper functions
z1_gen <- function(u, v) {
  return(sqrt(-2 * log(u)) * cos(2 * pi * v))
}

z2_gen <- function(u, v) {
  return(sqrt(-2 * log(v)) * sin(2 * pi * u))
}

# Initialization
n = 2000
u <- runif(n)
v <- runif(n)

# Get z1 and z2
```

Date: September 19, 2016.

```
z1 <- z1_gen(u, v)
z2 <- z2_gen(u, v)

# Plot the distributions
plot(z1, z2, main="Z1 vs. Z2", xlab="Z1", ylab="Z2")
```

.

This produces the following plot of z1 and z2:

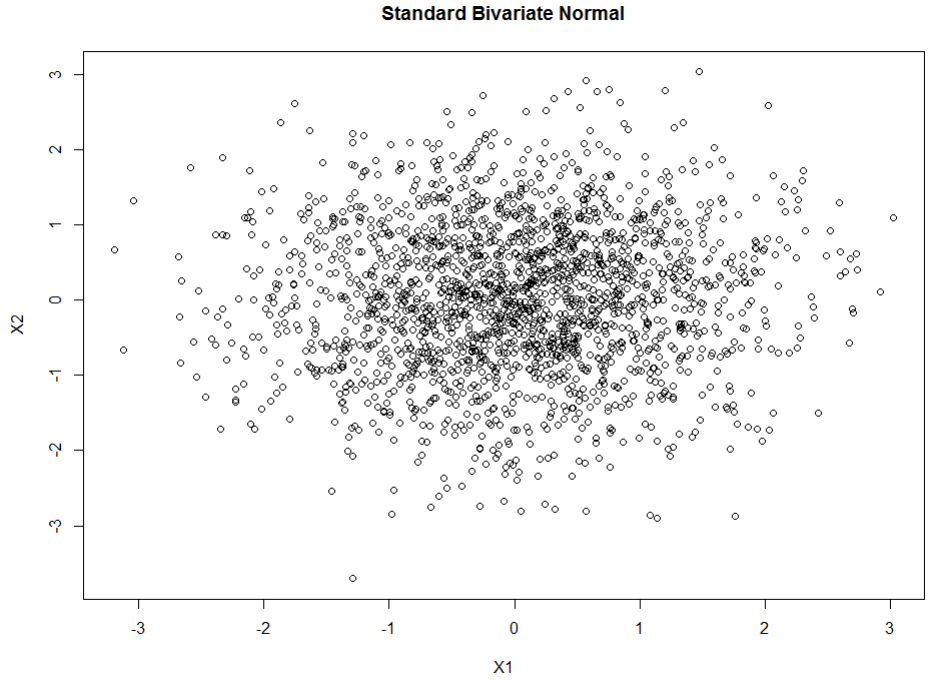


We can compare this graph with one produced by a standard bivariate normal distribution by running the following code:

```
# Standard Bivariate Normal
biv <- rmvnorm(n,c(0,0))

plot(biv[,1],biv[,2],main="Standard Bivariate Normal",xlab="X1",ylab="X2")
```

.

This produces the following graph:



The first graph plotted using Rizzo's functions seems quite different from this new graph, which gives us reason to suspect that something may not be right with Rizzo's functions.

We now run hypothesis tests on the points generated by Rizzo's functions to check if they are multinomial using the following code:

```
pair <- matrix(c(c(z1), c(z2)), ncol=2)
pair_df <- data.frame(pair)

# Assess the pair using MVN
# Useful documentation here: https://cran.r-project.org/web/packages/MVN/vignettes/MVN.pdf
mardiaResult <- mardiaTest(pair_df,qqplot=TRUE)
# Skew is too big, and thus is not multivariate normal

hzResult <- hzTest(pair_df)
# HZ statistic is too high - not multivariate normal

roystonResult <- roystonTest(pair_df)
# Here, the Royston Test thinks the data is multivariate normal
```

.

The following are the results of three hypothesis tests we used:
Mardia's Multivariate Normality Test

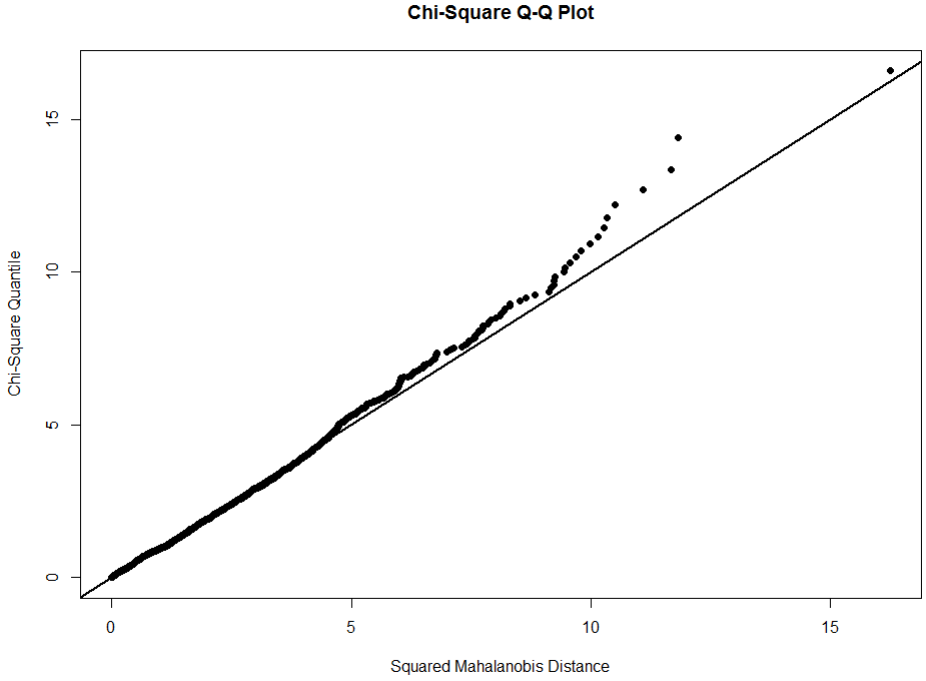
```
> print(mardiaResult)
   Mardia's Multivariate Normality Test
---------------------------------------
   data : pair_df

   g1p             : 0.8873402
   chi.skew        : 295.7801
   p.value.skew    : 8.81132e-63

   g2p             : 7.606148
   z.kurtosis      : -2.2017
   p.value.kurt    : 0.02768651

   chi.small.skew  : 296.5201
   p.value.small   : 6.101279e-63

   Result          : Data are not multivariate normal.
---------------------------------------

> print(hzResult)
  Henze-Zirkler's Multivariate Normality Test
---------------------------------------------
  data : pair_df

  HZ      : 33.42431
```

```
  p-value : 0

  Result  : Data are not multivariate normal.
-------------------------------------------

> print(roystonResult)
  Royston's Multivariate Normality Test
-------------------------------------------
  data : pair_df

  H       : 2.871799
  p-value : 0.2379013

  Result  : Data are multivariate normal.
-------------------------------------------
```

.

We observe that Mardia's Multivariate Normality Test and Henze-Zirkler's Multivariate Normality Test both conclude that the data are not multivariate normal, whereas Royston's Multivariate Normality Test concludes that the data are multivariate normal.

We can also check the Q-Q plot generated by Mardia's Multivariate Normality Test, which plots the theoretical and observed quantiles of the different distributions.



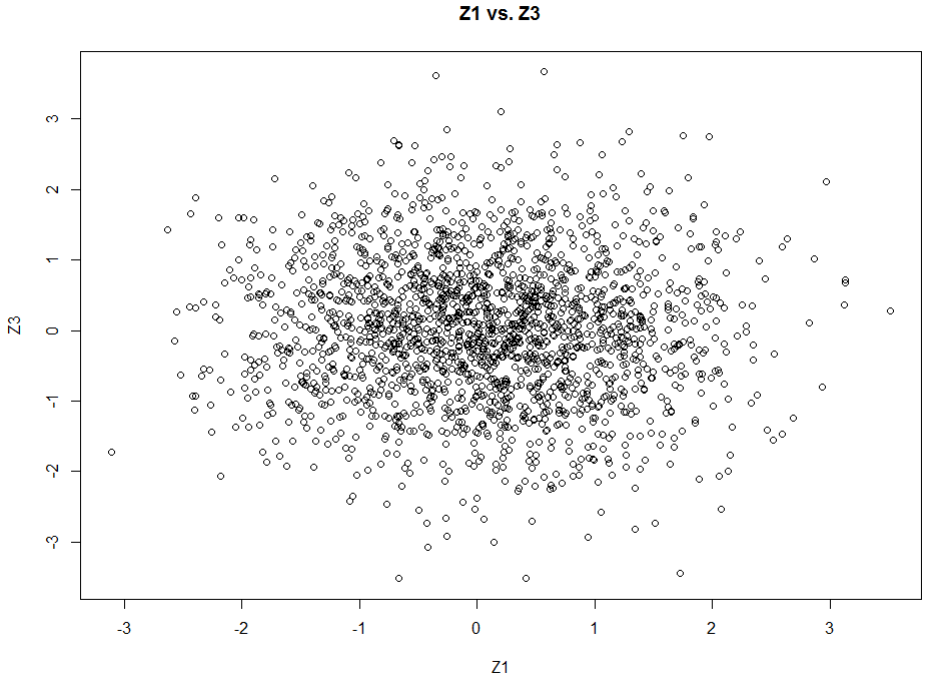
The plot indicates that the points deviate from the line $y = x$ which indicates multivariate normality, thus showing that the points are not multivariate normal.

Finally, we can try correcting Rizzo's function and check the result:

```
# Let's try correcting Rizzo's function
z3_gen <- function(u, v) {
  return(sqrt(-2 * log(u)) * sin(2 * pi * v))
}
# get z3
z3 <- z3_gen(u, v)
# Plot z1 against z3
plot(z1, z3, main="Z1 vs. Z3", xlab="Z1", ylab="Z3")
```

.

This produces the following scatterplot:



By swapping the position of U and V in Rizza's function, the resulting distribution is much more similar in appearance as compared to our standard bivariate normal plot. Even without running hypothesis tests for the points generated by the corrected function, we are quite confident that the Rizzo's function is indeed a typo.

3. Problem 3

(3) (a) First we generate uniform and normal points

```
# Problem 3

# Part A - Box-Muller normal random numbers

RNGkind("default", "Box-Muller")
set.seed(536)
unif_rands <- runif(4)

set.seed(536)
rnorms <- rnorm(4)
```

.

According to the C file snorm.c, the following two quantities are calculated (which we implement in R), since we need to create two independent random numberes each time we draw:

```
theta = 2 * pi * unif_rands[1]
R = sqrt(-2 * log(unif_rands[2]))
```

.

The normal random numbers are then generated as follows:

```
norm1 = R * cos(theta)
norm2 = R * sin(theta)
```

.

These numbers are exactly equal to each other according to R:

```
> cat(rnorms[1], ":", norm1, "  ", rnorms[1] == norm1,"\n")
-0.6713455 : -0.6713455    TRUE
> cat(rnorms[2], ":", norm2, "  ", rnorms[2] == norm2,"\n")
1.248904 : 1.248904    TRUE
```

.

(b) The code for the default inversion method in generating random numbers from a normal distribution is found in snorm.c and qnorm.c. According to snorm.c we increase the precision by multiplying a uniform random number by 2^27, then adding another uniform number to it, and finally dividing by 2^27 again. We can test this with the following code:

```
RNGkind("default", "default")
set.seed(536)
unif_rands <- runif(4)

set.seed(536)
rnorms <- rnorm(4)

u1 = (unif_rands[1] * (2^27) + unif_rands[2])/2^27
u2 = (unif_rands[3] * (2^27) + unif_rands[4])/2^27
```

.

We can clearly see that u1 and u2 have been changed - only the first six digits of u1 remain the same for example:

```
> print(format(c(u1,unif_rands[1]), digits = 20))
[1] "0.32850040092847632" "0.32850039820186794"
> print(format(c(u2,unif_rands[3]), digits = 20))
[1] "0.92515192263276713" "0.92515191785059869"
```

.

We then call qnorm5 in the file qnorm.c with mu=0, sigma=1, lower_tail=1 and log_p=0.

```
# qnorm defines q = p - 0.5
q1 <- u1 - 0.5
q2 <- u2 - 0.5
```

.

We now approximate the integral with a rational function, with constants supplied in the source code (calculated using the Remez function). Since $|q1| < 0.425$ and $|q2| < 0.425$, we do the following:

```
r1 <- .180625 - q1 * q1
r2 <- .180625 - q2 * q2

rational_approx <- function(r, q){
  return(
  q * (((((((r * 2509.0809287301226727 +
              33430.575583588128105) * r + 67265.770927008700853) * r +
            45921.953931549871457) * r + 13731.693765509461125) * r +
          1971.5909503065514427) * r + 133.14166789178437745) * r +
        3.387132872796366608) / (((((((r * 5226.495278852854561 +
          28729.085735721942674) * r + 39307.89580009271061) * r +
        21213.794301586595867) * r + 5394.1960214247511077) * r +
      687.1870074920579083) * r + 42.313330701600911252) * r + 1.)
  )
}
final1 = rational_approx(r1, q1)
final2 = rational_approx(r2, q2)
```

.

We can check the result:

```
> print(format(c(final1,rnorms[1]), digits = 20))
[1] "-0.44405779135054374" "-0.44405779328282380"
> print(format(c(final2,rnorms[2]), digits = 20))
[1] "1.4406055388735135" "1.4406055141632930"
```

.

This looks accurate up to about nine decimal places, but there is some approximation error in the rational function.

4. Problem 4

(4) (a) Let's start with 0.1. Using the long division method, we first arrive at the repeating bicimal (not sure if this is the right term, but binary decimal sounded strange) $0.0\overline{0011}_2$. This is only an assumption, so let us try and confirm this.
The repeating bicimal $0.0\overline{0011}_2$ can be expressed as

$$\frac{11_2}{10_2}\sum_{n=1}^{\infty}\left(\frac{1}{10_2^{100_2}}\right)^n = \frac{3}{2}\sum_{n=1}^{\infty}\left(\frac{1}{2^4}\right)^n = \frac{3}{2}\times\frac{1}{15} = \frac{1}{10}$$

Therefore, the complete binary representation of 0.1_{10} is the bicimal $0.0\overline{0011}_2$.
We can derive the representations for 0.2_{10} and 0.3_{10} using the same infinite sequence.

$$\begin{aligned} 0.2_{10} = 0.1_{10} \times 2 &= 2 \times \frac{3}{2} \sum_{n=1}^{\infty} \left(\frac{1}{2^4}\right)^n \\ &= 3 \times \sum_{n=1}^{\infty} \left(\frac{1}{2^4}\right)^n \\ &= 0.\overline{0011}_2 \end{aligned}$$

$$\begin{aligned} 0.3_{10} = 0.1_{10} \times 3 &= 3 \times \frac{3}{2} \sum_{n=1}^{\infty} \left(\frac{1}{2^4}\right)^n \\ &= \frac{9}{2} \sum_{n=1}^{\infty} \left(\frac{1}{2^4}\right)^n \\ &= 0.0\overline{1001}_2 \end{aligned}$$

(b) Using the exact representations we have, we can enforce arbitrary precision on the operation $0.2 - (.3 - .1)$. With infinite precision (current representation):

$$\begin{aligned} 0.2_{10} - 0.3_{10} + 0.1_{10} &= \frac{6}{2} \sum_{n=1}^{\infty} \left(\frac{1}{2^4}\right)^n - \frac{9}{2} \sum_{n=1}^{\infty} \left(\frac{1}{2^4}\right)^n + \frac{3}{2} \sum_{n=1}^{\infty} \left(\frac{1}{2^4}\right)^n \\ &= \left(\frac{6-9+3}{2}\right) \times \sum_{n=1}^{\infty} \left(\frac{1}{2^4}\right)^n = 0 \end{aligned}$$

Once the number of digits is fixed (eg. for double precision), the representation is not so accurate anymore. Since these are repeating bicimals with the same number of digits in their representation, using $1 + (4 \times n)$ digits of precision where n is an integer will result in there being no difference between 0.2 and $0.3 - 0.1$.
However, double precision uses 52 bits of precision, which leaves the representation incomplete, which means that the error must be within two bits in the binary representation. $2.775558e-17$ is the same as $\frac{1}{2^{55}}$, which is possibly the last carry bit in the operation.

5. Problem 5

(5) (a) Inverse Transform method
Can we work through a few more examples of the inverse transform, perhaps for slightly more tricky integrals?

(b) Viewing R source code
Could we go through how we can find the source code for any given R function?

(c) Comparison of different C interfaces (.Call, .C, .Primitive, .External)
What are the advantages and disadvantages of each of these various methods? Can we work through a more complex case of outsourcing computation to C?