Monte Carlo Methods
Homework 1

Daniel Gonzalez
30th of January, 2019

1 Problems

1. *Given a Linear Congruential Generator with parameters $a, c, m \in \mathbb{Z}$, show using mathematical induction that*

$$x_{n+k} \equiv a^k x_n + \frac{a^k - 1}{a - 1} c \mod m$$

where $a \geqslant 2$ and $k \geqslant 0$. This shows that the $(n+k)^{th}$ sequence term can be directly computed from the n^{th}.

Proof. A Linear Congruential Generator is a sequence $\langle x_n \rangle$ with parameters $a, c, m \in \mathbb{Z}$ defined by

$$x_{n+1} \equiv a x_n + c \mod m.$$

We will prove the claim by induction on $k \in \mathbb{N}$. Let $a \geqslant 2$ and $n \in \mathbb{N}$ and observe that, when $k = 0$, we trivially have

$$x_{n+0} = 1 x_n + \frac{1 - 1}{a - 1} c \equiv a^0 x_n + \frac{a^0 - 1}{a - 1} c \mod m.$$

Now, assume that the claim is true for some $k \in \mathbb{N}$. Observe that

$$\begin{aligned}
x_{n+(k+1)} &\equiv a x_{n+k} + c && \mod m \\
&\equiv a\left(a^k x_n + \frac{a^k - 1}{a - 1} c\right) + c && \mod m \\
&\equiv a^{k+1} x_n + \frac{a^{k+1} - a}{a - 1} c + \frac{a - 1}{a - 1} c && \mod m \\
&\equiv a^{k+1} x_n + \frac{a^{k+1} - 1}{a - 1} c && \mod m.
\end{aligned}$$

Therefore, for all $n \in \mathbb{N}$ and $k \in \mathbb{N}$, we can conclude that $a \geqslant 2$ implies

$$x_{n+k} \equiv a^k x_n + \frac{a^k - 1}{a - 1} c \mod m.$$

QED

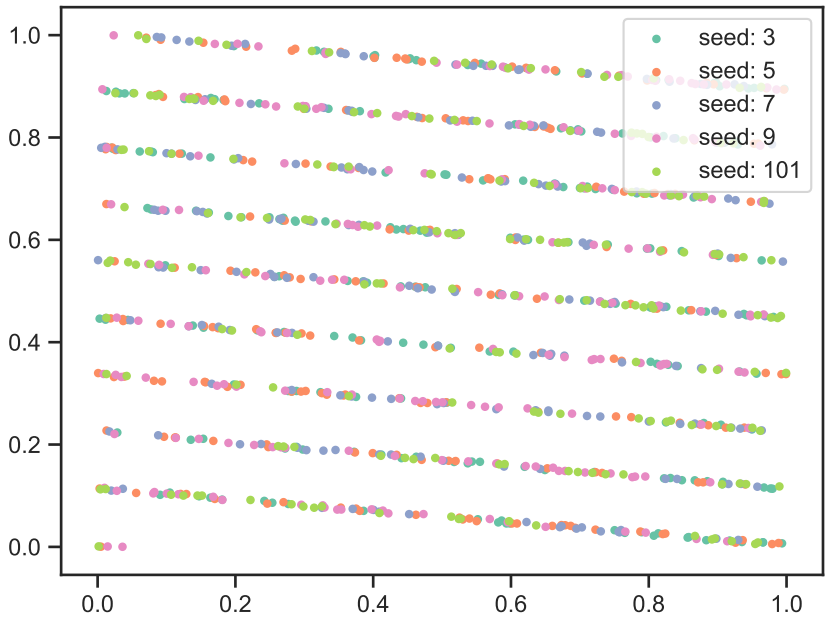
2. *Write a code that implements RANDU. For debugging purposes, print x_{1000} when the seed is $x_0 = 1$.*

   (a) *Using RANDU generate $u_1, u_2, \ldots u_{20002}$ where $u_n := {}^{x_n}/_{2^{31}}$. For all triples (u_i, u_{i+1}, u_{i+2}) such that $0.5 \leqslant u_{i+1} \leqslant 0.51$, plot the points (u_i, u_{i+1}). Comment on the pattern of your scatter plot.*

   The RANDU pseudorandom number generator was implemented in `randu.hs` using Haskell. The `randu` function takes an integer input and outputs the next element of the RANDU sequence, modified so that if $x_1 = 1$ is input, then $x_{1000} = 649091873$ is output. The `randuSeq` function takes an initial seed and a length parameter and generates the RANDU sequence of that length determined by that seed. The `uSeq` function then takes a list, such as the RANDU sequence, and divides every element of the list by 2^{31}, the modulus used for RANDU.

   The `partA` function constructs a list $[u_1, u_2, \ldots u_{20002}]$ using `uSeq`, on which the `parse` function acts to produce a list of ordered pairs (u_i, u_{i+1}) such that $0.5 \leqslant u_{i+1} \leqslant 0.51$. These lists of ordered pairs were computed for seeds $x_1 \in \{3, 5, 7, 9, 101\}$ and then plotted by `graph.py` using matplotlib with Python (and seaborn for aesthetics). The following figure shows the resulting plot. As we can clearly see, the points all lie on 10 parallel planes in $\mathbb{R}^2$. This shows that, even across different seeds, the RANDU sequence is highly nonrandom. When a value x_i in a RANDU sequence is observed to satisfy $0.5 \leqslant x_i \leqslant 0.51$, we can immediately deduce that the values x_{i-1} and x_{i+1} lie on one of these 10 planes.

Figure 1: Modified RANDU



(b) *Generate a sequence of length* 1002. *Use a program that plots points in* 3 *dimensions and rotates the axes to rotate the points until you can see the* 15 *planes.*

Using the `partB` function in the same `randu.hs` Haskell file, the standard RANDU sequence $x_1, x_2, \ldots x_{1002}$ was computed with $x_1 = 3$. The `triples` function then generated $(x_1, x_2, x_3), (x_2, x_3, x_4), \ldots (x_{1000}, x_{1001}, x_{1002})$, which was plotted using the plot3D library for R. The following are 15 plots highlighting the 15 different planes that the RANDU points lie on in $\mathbb{R}^3$.

Figure 2: RANDU plane 1

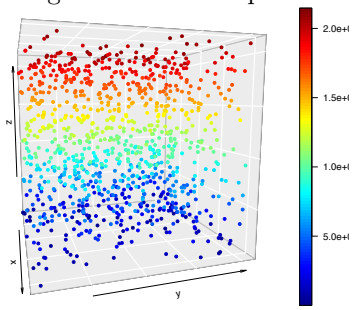


Figure 3: RANDU plane 2

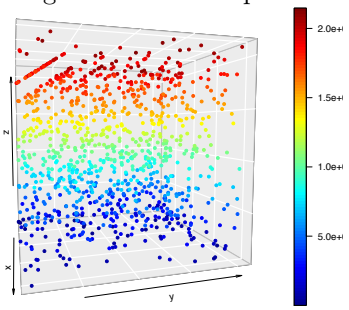


Figure 4: RANDU plane 3

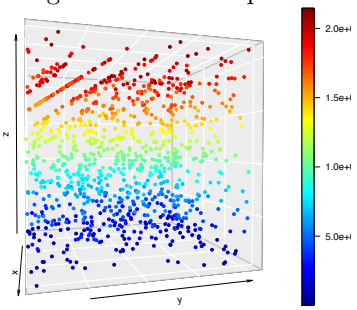


Figure 5: RANDU plane 4

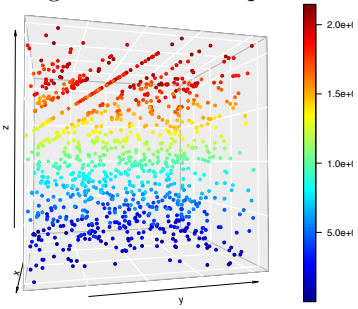


Figure 6: RANDU plane 5

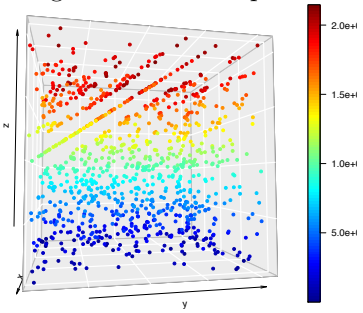


Figure 7: RANDU plane 6

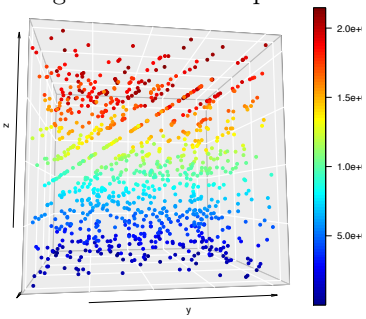


Figure 8: RANDU plane 7

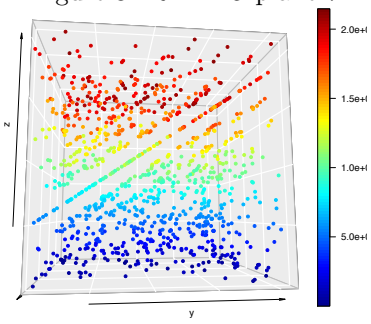


Figure 9: RANDU plane 8

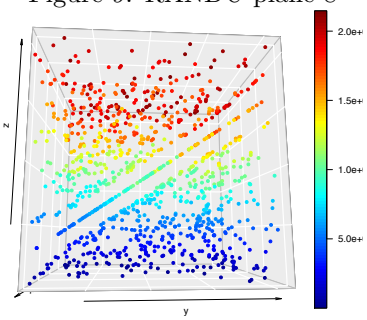


Figure 10: RANDU plane 9

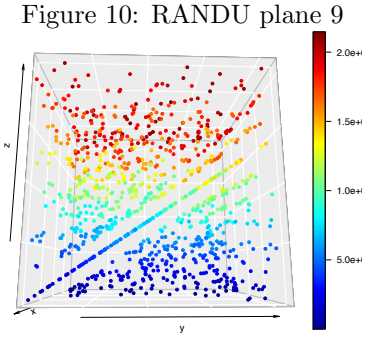


Figure 11: RANDU plane 10

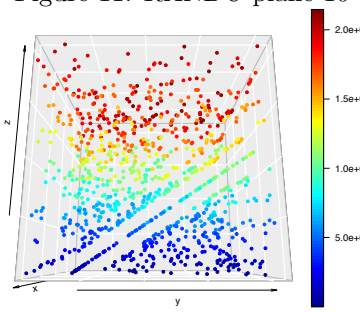


Figure 12: RANDU plane 11

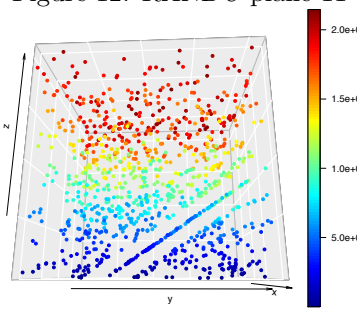


Figure 13: RANDU plane 12

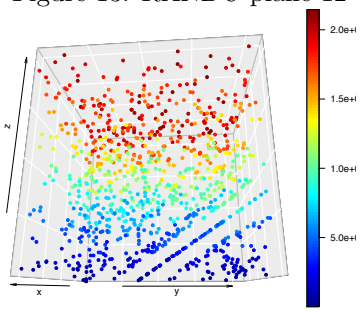


Figure 14: RANDU plane 13

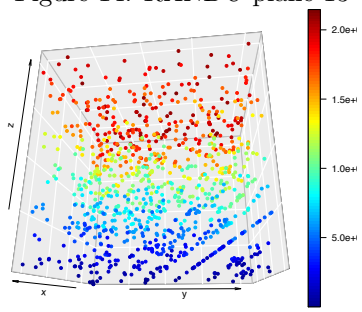


Figure 15: RANDU plane 14

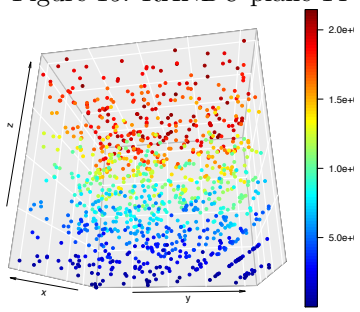
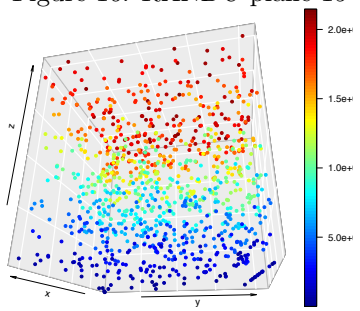


Figure 16: RANDU plane 15



3. *Download a code for the Mersenne Twister written by Mutsuo Saito and Makoto Matsumoto. Generate* 1002 *numbers, and plot pairs and triples of successive numbers for a visual inspection of randomness. Discuss your conclusions.*

   Though there are many implementations of the Mersenne Twister available for Haskell, the SIMD Fast Mersenne Twister found in System.Random.Mersenne was chosen for its efficiency and the clarity of its documentation.

   The `mersenne.hs` file implements the `mersenne` function, which generates numbers in $[0, 1)$ using the aforementioned Mersenne Twister seeded by the current system time. The `mersenneList` function then (lazily) constructs an infinite sequence of numbers using `mersenne`, from which the first n are taken. These n numbers are then arranged into a list of either pairs or triples by the `pairs` and `triples` functions.

   The following two figures are plots of the same sequence of 1002 numbers, generated in the manner described above, split into a sequences of pairs and triples respectively. As we can clearly see, the plots appear completely random and there is no readily apparent structure in the data from visual inspection.

Figure 17: Mersenne Twister in 2D

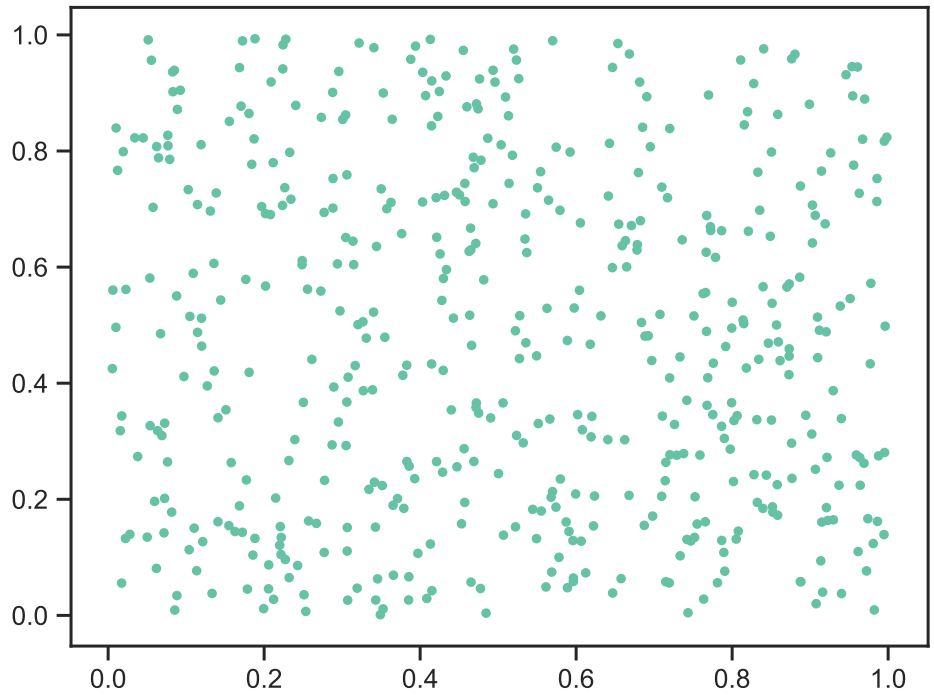
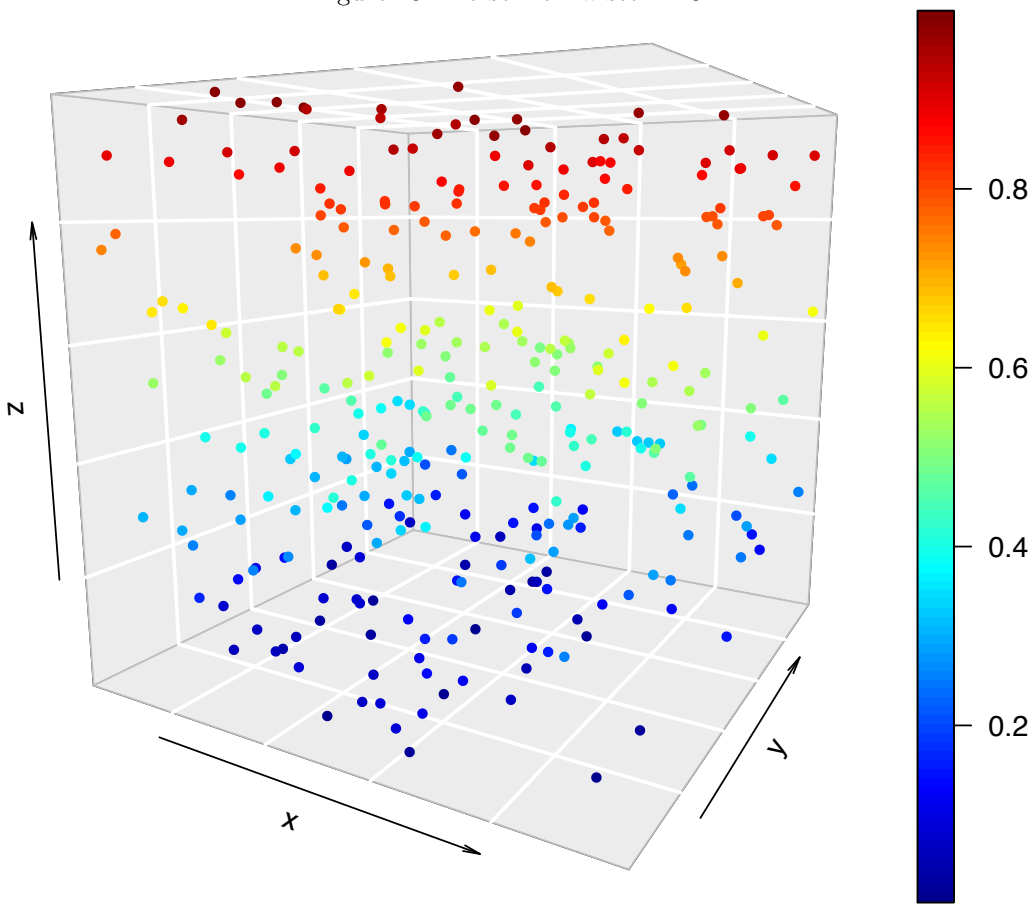


Figure 18: Mersenne Twister in 3D

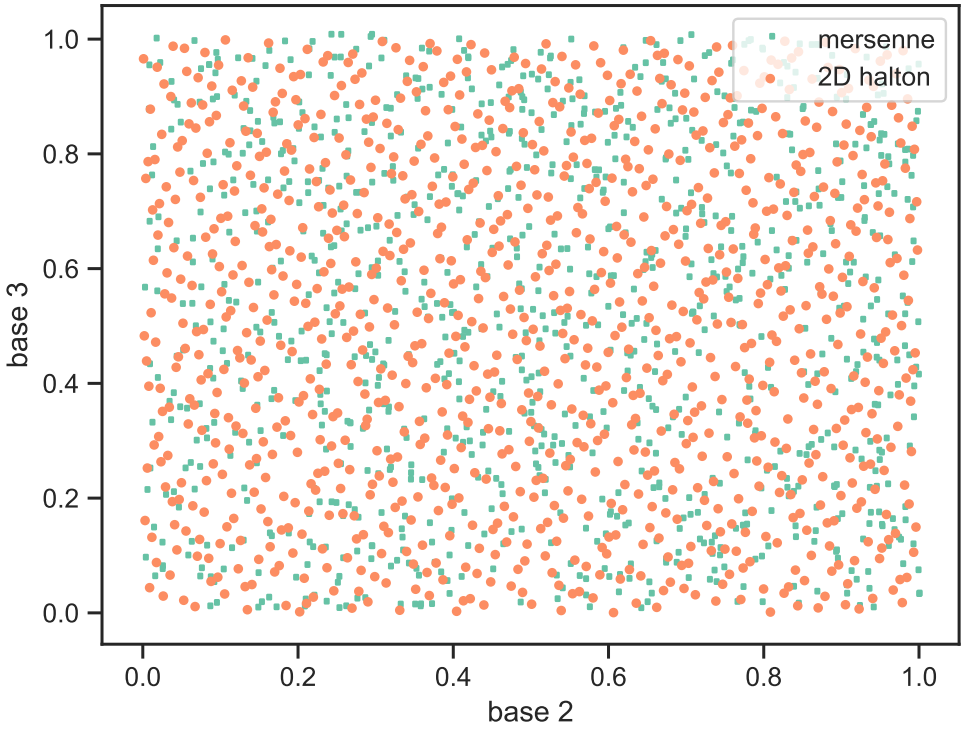


4. *Write a computer code for the Halton sequence. The input should be the dimension s and n, and the output should be the n^{th} Halton vector where the bases are the first s primes. Then, generate* 1000 *two-dimensional Halton vectors and plot them. Also plot* 1000 *two-dimensional vectors obtained by the Mersenne Twister. Can you make any observations comparing the two plots visually?*

   The `halton.hs` file implements multi-dimensional Halton sequences where the basis along dimension i is the i^{th} prime number. First, the `primes` function generates an infinite list of all prime numbers (lazily evaluated, thanks to Haskell). The `haltonList` function takes two parameters, the dimension s and the length of the list n, and then takes s primes from the list generated by `primes` and computes n terms of the Halton sequence for each prime. The resulting sequences are arranged into lists of length s and returned to the user as a sequence of n such lists.

   Using the parameters $s = 2$ and $n = 1000$, a list of 1000 two-dimensional Halton vectors was computed in the above fashion. This two-dimensional data is plotted below against 1000 points generated by the Mersenne Twister from the previous problem. We can observe that the Halton sequence seems to fill the space $[0, 1] \times [0, 1]$ more evenly than the Mersenne sequence, which appears to occasionally clump together into small clusters of between 3 and 6 points.

Figure 19: 2D Halton Sequence



5. *Consider the quadrature problem* $\int_0^1 e^x \, \mathrm{d}x$. *You will estimate this integral using Monte Carlo and Random Quasi-Monte Carlo methods by computing* $\theta_k \coloneqq \frac{1}{N} \sum_{i=1}^{N} e^{x_i}$, *where the subscript* k *refers to the* k*th estimate.*

   (a) *Use Mersenne Twister to obtain* $\theta_1, \theta_2, \ldots \theta_m$ *where* $m = 40$ *and* $N = 1000$. *Compute the mean and standard deviation of the estimates.*

   (b) *Use random shifted Halton sequences to obtain* $\theta_1, \theta_2, \ldots \theta_m$ *where* $m = 40$ *and* $N = 1000$. *Compute the mean and standard deviation of the estimates.*

   (c) *Repeat parts* (a) *and* (b) *with* $N = 10000$.

   (d) *Compute the exact value of the integral. Present all of the above data together to create a convincing argument for which method is better for approximating the integral.*

	MC		RQMC	
N	1,000	10,000	1,000	10,000
Abs. Error	6.485×10^{-4}	5.624×10^{-4}	5.907×10^{-5}	1.235×10^{-5}
Mean μ	1.718930378181906	1.718844274942174	1.71822275488340	1.718294181840698
Std. Dev. σ	1.752×10^{-2}	4.345×10^{-3}	8.024×10^{-4}	9.477×10^{-5}

The above table summarizes the results of the quadrature methods implemented in the `MC.hs` and `RQMC.hs` files. The random shifted Halton sequence used for the RQMC estimation is a Van der Corput sequence shifted by a number generated by the Mersenne Twister and truncated to lie within $[0, 1]$. The value of the integral is $\int_0^1 e^x \, \mathrm{d}x = e - 1 \approx 1.7182818284590452 35$, which was used to compute the errors in the table.

As we can see from the table, the RQMC method produces, with the same number of sample points, estimates of the integral which are more accurate by an order of magnitude (10^{-4} versus 10^{-5} for MC and RQMC respectively). Not only that, but when $N = 1,000$ the RQMC method has a standard deviation an order of magnitude lower than the MC method. The same is true for $N = 10,000$, showing that RQMC dominates MC in all metrics when it comes to estimating the integral. However, there is a caveat with using RQMC: when estimating with $N = 10,000$, generating the RQMC estimates took more than twice as long as the MC. This shows us that the performance of RQMC comes at the cost of computation time.

These findings are further supported by the following plots, which also summarize the data.

Figure 20: MC vs. RQMC with $N = 1,000$

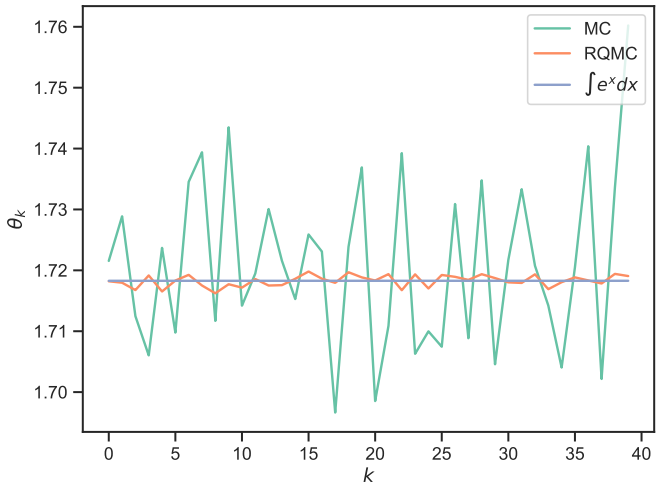
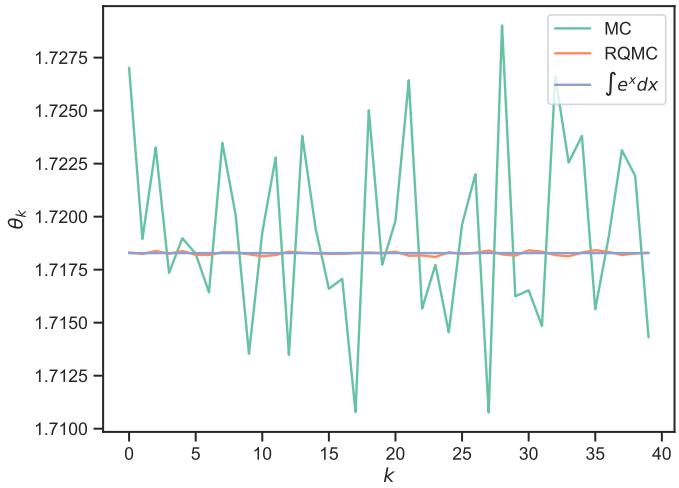


Figure 21: MC vs. RQMC with $N = 10,000$



2 Appendix

2.1 RANDU Haskell Code (`randu.hs`)

```haskell
main = partA

partA = sequence (map print (parse list []))
    where list = (uSeq (randuSeq length [] seed))
          seed = 3
          length = 20002

partB = sequence (map print (triples list []))
    where list = (randuSeq length [] seed)
          seed = 3
          length = 1002

randu :: (Integral a) => a -> a
randu 1 = mod (65539 ^ 1000) (2 ^ 31)
randu x = mod (65539 * x) (2 ^ 31)

randuSeq :: (Integral a) => a -> [a] -> a -> [a]
randuSeq 1 seq seed = [randu seed]
randuSeq n seq seed = (randu (head rs)):rs
    where rs = randuSeq (n-1) seq seed

uSeq :: (Integral a, Fractional b) => [a] -> [b]
uSeq rs = [(fromIntegral r) / (fromIntegral (2 ^ 31)) | r <- rs]

parse :: (Fractional a, Ord a) => [a] -> [(a, a)] -> [(a, a)]
parse [] result = result
parse (x:[]) result = result
parse (x:y:[]) result = result
parse (x:y:z:us) result
    | (0.5 <= y) && (y <= 0.51) = parse (y:z:us) ((x, z):result)
    | otherwise = parse (y:z:us) result

triples :: (Num a) => [a] -> [(a, a, a)] -> [(a, a, a)]
triples [] result = result
triples (x:[]) result = result
triples (x:y:[]) result = result
triples (x:y:z:rs) result = triples (y:z:rs) ((x, y, z):result)
```

2.2 Mersenne Twister Code (`mersenne.hs`)

```haskell
import System.Random.Mersenne

main = (sequence . (map print)) =<< (sequence (f (take n mersenneList)))
    where f = pairs
          n = 1002

mersenne :: IO Double
mersenne = (getStdGen :: IO MTGen) >>= (random :: MTGen -> IO Double)

mersenneList :: [IO Double]
mersenneList = [mersenne | i <- [1..]]

pairs :: (Show a) => [IO a] -> [IO [a]]
pairs [] = []
pairs (x:[]) = []
pairs (x:y:ms) = (sequence [x, y]):(pairs ms)

triples :: (Show a) => [IO a] -> [IO [a]]
triples [] = []
triples (x:[]) = []
triples (x:y:[]) = []
triples (x:y:z:ms) = (sequence [x, y, z]):(triples ms)
```

2.3 Halton Sequence Code (`halton.hs`)

```haskell
main = sequence (map print (haltonList 2 1000))

primes :: (Integral a) => [a]
primes = 2 : primes'
    where isPrime (p:ps) n = p^2 > n || (mod n p /= 0 && isPrime ps n)
          primes' = 3 : filter (isPrime primes') [5, 7..]

halton :: Int -> Int -> Double
halton b i = halton' b i 1 0
    where halton' :: Int -> Int -> Double -> Double -> Double
          halton' b i f r
              | (i > 0) = halton' b i' f' r'
              | otherwise = r
              where i' = floor (fromIntegral i / fromIntegral b)
                    f' = f / fromIntegral b
                    r' = r + f' * fromIntegral (mod i b)

haltonList :: Int -> Int -> [[Double]]
haltonList s n = [[halton b i | b <- (take s primes)] | i <- [1..n]]
```

2.4 Monte Carlo Code (MC.hs)

```haskell
import System.Random.Mersenne

main = print =<< fmap stats (sequence (take m (estimate n)))
    where m = 40
          n = 10000

mersenne :: IO Double
mersenne = (getStdGen :: IO MTGen) >>= (random :: MTGen ->  IO Double)

mersenneList :: [IO Double]
mersenneList = [mersenne | i <- [1..]]

estimate :: Int -> [IO Double]
estimate n = [fmap ((\x -> x / (fromIntegral n)) . sum . (map exp)) (sequence (take n mersenneList)) | i <- [1..]]

stats :: [Double] -> (Double, Double)
stats θs = (mean, std)
    where mean = (sum θs) / (fromIntegral (length θs))
          std = sqrt (sum (map (\x -> (x - mean)^2) θs) / (fromIntegral (length θs - 1)))
```

2.5 Random Quasi-Monte Carlo Code (RQMC.hs)

```haskell
import System.Random.Mersenne

main = print =<< fmap stats (sequence (take m (estimate n)))
    where m = 40
          n = 10000

halton :: Int -> Double
halton i = halton' i 1 0
    where halton' :: Int -> Double -> Double -> Double
          halton' i f r
              | (i > 0) = halton' i' f' r'
              | otherwise = r
              where i' = floor (fromIntegral i / fromIntegral 2)
                    f' = f / fromIntegral 2
                    r' = r + f' * fromIntegral (mod i 2)

haltonList :: [Double]
haltonList = [halton i | i <- [1..]]

mersenne :: IO Double
mersenne = (getStdGen :: IO MTGen) >>= (random :: MTGen -> IO Double)

shift :: (RealFrac a) => [a] -> a -> [a]
shift hs u = map (\x -> (x + u) - fromIntegral (floor (x + u))) hs

estimate :: Int -> [IO Double]
estimate n = [fmap (estimate' . shift (take n (haltonList))) mersenne | i <- [1..]]

estimate' :: [Double] -> Double
estimate' hs = (sum hs') / fromIntegral (length hs')
    where hs' = map exp hs

stats :: [Double] -> (Double, Double)
stats θs = (mean, std)
    where mean = (sum θs) / (fromIntegral (length θs))
          std = sqrt (sum (map (\x -> (x - mean)^2) θs) / (fromIntegral (length θs - 1)))
```

2.6 2D Graphing (graph.py)

```python
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set(style='ticks', palette='Set2')

def graph_randu(X, Y, filename):
    fig, ax = plt.subplots()
    ax.scatter(X[0], Y[0], marker='.', label='seed: 3')
    ax.scatter(X[1], Y[1], marker='.', label='seed: 5')
    ax.scatter(X[2], Y[2], marker='.', label='seed: 7')
    ax.scatter(X[3], Y[3], marker='.', label='seed: 9')
    ax.scatter(X[4], Y[4], marker='.', label='seed: 101')
    legend = ax.legend(loc='upper right')
    plt.savefig('../figures/' + filename + '.pdf', format='pdf', dpi=1000, bbox_inches='tight')

def graph_mersenne(X, Y, filename):
    fig, ax = plt.subplots()
    ax.scatter(X, Y, marker='.')
    plt.savefig('../figures/' + filename + '.pdf', format='pdf', dpi=1000, bbox_inches='tight')

def graph_halton(X1, Y1, X2, Y2, filename):
    fig, ax = plt.subplots()
    ax.scatter(X2, Y2, marker='$\cdot$', label='mersenne')
    ax.scatter(X1, Y1, marker='.', label='2D halton')
    plt.xlabel('base 2')
    plt.ylabel('base 3')
    legend = ax.legend(loc='upper right')
    plt.savefig('../figures/' + filename + '.pdf', format='pdf', dpi=1000, bbox_inches='tight')

def graph_quadrature(Y1, Y2, filename):
    fig, ax = plt.subplots()
    ax.plot(Y1, label='MC')
    ax.plot(Y2, label='RQMC')
    ax.plot([np.exp(1)-1 for i in range(1, 41)], label=r'$\int e^x dx$')
    plt.xlabel('$k$')
    plt.ylabel(r'$\theta_k$')
    legend = ax.legend(loc='upper right')
    plt.savefig('../figures/' + filename + '.pdf', format='pdf', dpi=1000, bbox_inches='tight')

def main():
    randuX = [\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[0]) for line in open('../output/randu3.dat')],\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[0]) for line in open('../output/randu5.dat')],\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[0]) for line in open('../output/randu7.dat')],\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[0]) for line in open('../output/randu9.dat')],\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[0]) for line in open('../output/randu101.dat')]]
    randuY = [\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[1]) for line in open('../output/randu3.dat')],\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[1]) for line in open('../output/randu5.dat')],\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[1]) for line in open('../output/randu7.dat')],\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[1]) for line in open('../output/randu9.dat')],\
            [float(line.strip().strip('\n').strip('(').strip(')').split(',')[1]) for line in open('../output/randu101.dat')]]

    mersenneX1 = [float(line.strip().strip('\n').strip('[').strip(']').split(',')[0]) for line in open('../output/mersenne_pairs.dat')]
    mersenneY1 = [float(line.strip().strip('\n').strip('[').strip(']').split(',')[1]) for line in open('../output/mersenne_pairs.dat')]

    haltonX = [float(line.strip().strip('\n').strip('[').strip(']').split(',')[0]) for line in open('../output/halton.dat')]
    haltonY = [float(line.strip().strip('\n').strip('[').strip(']').split(',')[1]) for line in open('../output/halton.dat')]
    mersenneX2 = [float(line.strip().strip('\n').strip('[').strip(']').split(',')[0]) for line in open('../output/mersenne.dat')]
    mersenneY2 = [float(line.strip().strip('\n').strip('[').strip(']').split(',')[1]) for line in open('../output/mersenne.dat')]

    MC1 = [float(x) for x in [line.strip().strip('\n').strip('[').strip(']').split(',') for line in open('../output/MC1.dat')][0]]
    MC2 = [float(x) for x in [line.strip().strip('\n').strip('[').strip(']').split(',') for line in open('../output/MC2.dat')][0]]
    RQMC1 = [float(x) for x in [line.strip().strip('\n').strip('[').strip(']').split(',') for line in open('../output/RQMC1.dat')][0]]
    RQMC2 = [float(x) for x in [line.strip().strip('\n').strip('[').strip(']').split(',') for line in open('../output/RQMC2.dat')][0]]

    graph_randu(randuX, randuY, 'randu')
    graph_mersenne(mersenneX1, mersenneY1, 'mersenne2D')
    graph_halton(haltonX, haltonY, mersenneX2, mersenneY2, 'halton')
    graph_quadrature(MC1, RQMC1, 'MC1')
    graph_quadrature(MC2, RQMC2, 'MC2')

main()
```

2.7 3D Graphing (graph.r)

```
library("plot3D")

# GRAPHING RANDU SEQUENCE
data ← read.csv(file="../output/randu_triples.dat", head=FALSE)
x ← as.numeric(substring(data$V1, 2))
y ← data$V2
z ← as.numeric(substring(data$V3, 1, nchar(as.character(data$V3))-1))
scatter3D(x, y, z, pch=20, theta=73, phi=7, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=74, phi=4, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=75, phi=1, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=77, phi=-1, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=79, phi=-3, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=82, phi=-5, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=84, phi=-7, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=86, phi=-9, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=88, phi=-11, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=90, phi=-13, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=93, phi=-15, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=95, phi=-17, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=97, phi=-19, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=99, phi=-21, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
scatter3D(x, y, z, pch=20, theta=102, phi=-23, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")

# GRAPHING MERSENNE TWISTER SEQUENCE
data ← read.csv(file="../output/mersenne_triples.dat", head=FALSE)
x ← as.numeric(substring(data$V1, 2))
y ← data$V2
z ← as.numeric(substring(data$V3, 1, nchar(as.character(data$V3))-1))
scatter3D(x, y, z, pch=20, theta=30, phi=10, bty="g", ticktype="simple",
          xlab="x", ylab="y", zlab="z")
```