Numerical Quadrature using (Quasi) Monte-Carlo Methods

Ahmer Nadeem Khan

10 February 2025

1 Abstract

In this project, we will test three methods for numerical quadrature; a Monte-Carlo Method using the Mersenne Twister random number generator, and two quasi Monte-Carlo Methods using Halton Sequences. We first provide some mathematical preliminaries and theory concerning random number generation using these methods, and how it relates to quadrature. We only simply present certain important theorems and state them without proof. We will then consider a specific one dimensional example and evaluate our methods numerically, providing statistical results and inference. We then substantiate theoretical predictions about the strength of these methods. (Homework problems are marked in red).

2 Objectives

Our objective is to numerically estimate the definite integral of a function in a given interval. We first give a description of the necessary mathematical notions and then we perform the estimation using three different methods:

- Mersenne Twister (MC)
- Randomly Shifted Halton Sequences (RQMC)
- Randomly Started Halton Sequences (RQMC)

For each method we compute a sequence of N terms in the given interval (with certain properties) and then calculate

$$\theta_k = \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

This gives a single estimate of the integral. We then compute m of these estimates $\theta_1, \ldots, \theta_m$, and compute an average as our final estimate. We then

present summary statistics about our computations and test the algorithms for different parameter values to demonstrate behavior of the methods. We finally present comparisons and conclusions about our results.

3 Random Number Generators

Our goal is to generate pseudo-random numbers in the interval $(0, 1)$ i.e. to simulate the uniform random distribution $Unif\,(0, 1)$. Computationally, true random sequences from the distribution are untractable (in the sense of a Marin-Löf sequence); we can only construct algorithms to generate numbers that pass a finite number of tests for *typicality* or *compressibility*, two equivalent formulations of randomness. The most basic of these pseudo-random numbers is a class of recursive methods known as Linear Congruential Generators, introduced by Lehmer in 1949.

3.1 Linear Congruential Generators (LCG)

Definition 1. *A **Linear Congruential Generator** (LCG) with parameters a, c and m is a recursive algorithm that generates a sequence of pseudo-random numbers $(x_i)_{i=0}^{\infty}$ that satisfy the following linear congruence:*

$$x_{n+1} = ax_n + c \pmod{m}$$

such that $m > 0$, $0 < a < m$ and $0 \leq c < m$.

It is clear that a number sequence produced by an LCG can only take values in the set $\{0, 1, \ldots, m-1\}$. To obtain pseudo-random samples in the interval $(0, 1)$ we use the transformation

$$u_n = \frac{x_n}{M}$$

There is also obviously a period associated with each LCG since x_{n+1} is determined by x_n, and there can only be a finite number of distinct values in the sequence. Thus, in practice we only generate a finite sequence. The sequence (and thus, the period) also depends on the chosen value of x_0 which is called the **seed** of the LCG. The following theorem characterizes the LCGs with a full period.

Theorem 3.1 (**Greenberger; Hull, Dobell 1962**). *The period of an LCG is m if and only if*

1. $(c, m) = 1 \;\; (c \neq 0)$;
2. $a \equiv 1 \pmod{p}$ *for each prime factor p of m;*
3. $a \equiv 1 \pmod{4}$ *if* $4 \mid m$.

where (a, b) is defined to be $\gcd(a, b)$, *and $a \mid b$ means a divides b.*

We now consider simple definitions and state results from elementary number theory to be used later. We first state, without proof, the following theorem.

Theorem 3.2. *Let a and b be integers that are not both zero. Then* $\gcd(a, b)$ *is the least positive integer that is a linear combination of a and b.*

(Proof Sketch). We use the **Division Algorithm** and the **Euclidean Algorithm**, and elementary facts about division for the proof. □

Theorem 3.3. *Let a and b be integers, not both zero. Then* $\gcd(a, b) = 1$ *if and only if there exist integers s and t such that*

$$1 = as + bt.$$

Proof. If $\gcd(a, b) = 1$, then by Theorem 2.2 there exist integers s and t such that

$$as + bt = 1.$$

Conversely, suppose that there are integers s and t such that $as + bt = 1$. Since $\gcd(a, b)$ is the smallest positive integer that is a linear combination of a and b, and 1 is a linear combination of a and b, it follows that $\gcd(a, b) = 1$. □

It will also be useful to define a linear congruence in general.

Definition 2. *A **linear congruence** is*

$$ax \equiv b \pmod{m}$$

where $a, b \in \mathbb{Z}$ and m is a positive integer ≥ 2.

Definition 3. *An **inverse** of $a \pmod{m}$ is any integer a' such that $aa' \equiv 1 \pmod{m}$.*

The following are the main results of use.

Lemma 3.4. *Solving the congruence $ax \equiv b \pmod{m}$ is equivalent to solving the linear Diophantine equation $ax + my = b$.*

Proof. $ax \equiv b \pmod{m}$ is equivalent $ax - b \equiv 0 \pmod{m}$ which is equivalent to the linear equation $ax - b = m(-y)$ for some integer $-y$ (since y is arbitrary). Then we get the Diophantine equation $ax + my = b$. □

Theorem 3.5. *Let $m, a, b \in \mathbb{Z}$ and $m > 1$. Let $d = \gcd(a, m)$.*

- *If $d \nmid b$, then the linear congruence $ax \equiv b \pmod{m}$ has no solution.*
- *If $d \mid b$, then the linear congruence $ax \equiv b \pmod{m}$ has exactly d solutions (mod m).*

Proof. If $d \nmid b$, then suppose there is some $x \in \mathbb{Z}$ such that $ax \equiv b \pmod{m}$, i.e.

$$ax = b + my \quad \text{for some} \quad y \in \mathbb{Z}$$

which implies

$$ax - my = b.$$

Since $ax - my$ is a linear combination of a and m, by the definition of gcd, it must be a multiple of $\gcd(a, m)$. Therefore:

$$d \mid b.$$

which is a contradiction. Hence, the linear congruence $ax \equiv b \pmod{m}$ has no solution when $d \nmid b$.

Now, if $d \mid b$, then since $d = \gcd(a, m)$, there exist integers $x_1, y_1 \in \mathbb{Z}$ such that

$$\begin{aligned} & d = ax_1 + my_1 \\ \implies & d = (ax_1 + my_1) \mid b \\ \implies & b = (ax_1 + my_1)x_0, \qquad x_0 \in \mathbb{Z} \\ \implies & b = ax_1x_0 + my_1x_0 \\ \implies & a(x_1x_0) \equiv b \pmod{m} \end{aligned}$$

Thus $x = x_1x_0$ is a solution to the original linear congruence $ax \equiv b \pmod{m}$. We claim that:

$$x = x_0 + \frac{m}{d}t \quad \text{and} \quad y = y_0 + \frac{a}{d}t$$

are solutions of the Diophantine equation $ax - my = b$, where (x_0, y_0) is any particular solution. Substituting, we get:

$$a\left(x_0 + \frac{m}{d}t\right) - m\left(y_0 + \frac{a}{d}t\right) = ax_0 - my_0 + \left(a \cdot \frac{m}{d} - m \cdot \frac{a}{d}\right)t.$$

This simplifies to:

$$ax_0 + my_0 = b \quad (\text{since } (x_0, y_0) \text{ is a solution}).$$

Since t runs through the values $0, 1, 2, \ldots, d-1$, the congruence has d solutions modulo m. □

We immediately have the following result.

Corollary 3.5.1. *An inverse of $a \pmod{m}$ exists if and only if $(a, m) = 1$.*

Proof. The converse is a simple application of Theorem 2.5 with d=1, and we get a single, unique solution to the linear congruence $ax \equiv 1 \pmod{M}$. This is by definition the inverse of a.

For the forward direction assume that there exists a' such that $aa' \equiv 1 \pmod{M}$. By Lemma 2.4, this is equivalent to the equation

$$aa' + my = 1$$

This is a linear combination of a and m which equals 1. Then by Theorem 2.3, $\gcd(a, m) = 1$. □

We may now prove the following results (Homework 1.1). Note the difference between congruence and equality in the equations; we say in the LCG equation that x_{n+1} is *equal*, by definition, to a linear equation modulo m. Note also that it can be proved too as a simple corollary of Theorem 2.5 (with $d = 1$) and its proof, but we give an equivalent proof here using Corollary 2.5.1 instead.

Corollary 3.5.2. *For a LCG the term x_n can be uniquely determined by x_{n+1} if $(a, m) = 1$*

Proof. Assume WLOG, that $\exists\ x_n, y_n$ s.t

$$x_{n+1} = ax_n + c \pmod{m} \tag{1}$$

$$x_{n+1} = ay_n + c \pmod{m} \tag{2}$$

Subtracting (1) and (2), we get

$$0 = ax_n - ay_n + (c - c) \pmod{m}$$

where we have used the property that $(a + b) \pmod{m} = a \pmod{m} + b \pmod{m} \pmod{m}$, which is easy to show. We therefore get

$$0 = a(x_n - y_n) \pmod{m} \tag{3}$$

where we simply use linearity as two equal numbers must be equal modulo m. By assumption $(a, m) = 1$, so by Corollary 2.5.1, there exists a unique inverse of $a \pmod{m}$, say a'. Multiplying both sides of the equation (3) by a', we get

$$0 = a'a\,(x_n - y_n) \pmod{m}$$

$$0 = (x_n - y_n) \pmod{m}$$

which then implies that

$$x_n \equiv y_n \pmod{m}$$

Note the congruence relation replaces the equality now, since what what we mean is that x_n and y_n have the same remainder when divided by m. Then by construction of a LCG, we have that $x_n, y_n \in \{0, 1, \ldots, m - 1\}$, and they are congruent modulo m. Then, they must be the same number i.e.

$$x_n = y_n$$

□

To complete our characterization, we prove the following corollary.

Corollary 3.5.3. *If* $(a, m) > 1$*, then*

$$a\left(x_n + k\frac{m}{\gcd(a,m)}\right) + c \equiv x_{n+1} \pmod{m}$$

for any integer k*.*

Proof. Set $d = \gcd(a, b)$. Then consider the equations (1) and 2 in the previous proof i.e.

$$x_{n+1} = ax_n + c \pmod{m} \tag{1}$$

and

$$x_{n+1} = ay_n + c \pmod{m} \tag{2}$$

with no restriction on y_n. These equations give again an equivalent equation

$$0 \equiv a(y_n - x_n) \pmod{m}. \tag{3}$$

We now set $z_n = y_n - x_n$ to get the linear congruence

$$az_n \equiv 0 \pmod{m} \tag{4}$$

with $(a, m) = d > 1$. We also know that $z_n = 0$ is a solution to the congruence, since this gives us $x_n = y_n$ and then both equations (1) and (2) are true by construction of the LCG, which we assume. This we take to be the particular solution $z'_n = 0$. By the proof of Theorem 2.5, we get that there are $d > 1$ solutions to the congruence (4), where the solutions are given by the form

$$z_n = z'_n + k\frac{m}{d}.$$

Substituting for the particular solution $z'_n = 0$, we get the form

$$z_n = k\frac{m}{d}$$

and substituting back for z_n and $d = \gcd(a, b)$ we get

$$y_n = x_n + k\frac{m}{\gcd(a,b)}$$

Plugging this into (2) gives us the required result.

□

This result means there are multiple values for x_n that can yield x_{n+1} (exactly d unique values, in fact).

3.2 Mersenne Twister

We can generalize the LCG in the following way to get a larger class of generators.

Definition 4. *A **Multiple Recursive Generator** has the form*

$$x_n \equiv a_1 x_{n-1} + \ldots + a_k x_{n-k} \pmod{p}$$

where p is a prime.

It is possible to find $a_1, \ldots, a_k$ such that the sequence has period length $p^k - 1$. Initial values $x_0, \ldots, x_{k-1}$ can be chosen arbitrarily, but not all zero. A special case, $p = 2$, is called a *feedback shift register generator*. The period of:

$$x_n \equiv a_1 x_{n-1} + \cdots + a_k x_{n-k} \pmod{2}$$

is $2^k - 1$ if and only if the polynomial:

$$f(z) = z^k - (a_1 z^{k-1} + \cdots + a_k)$$

defined over the field of integers $\{0, 1\}$ is irreducible (provided not all initial values x_i are zero). The sequence of binary digits x_n is then partitioned into blocks, and each block is transformed to an integer. For example, if blocks of size 4 are formed:

$$[0, 1, 1, 0],\ [1, 1, 1, 0],\ \ldots$$

then the first block is converted to an integer as:

$$0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 6.$$

If the partition into blocks is done with a delay between blocks, then we obtain the so-called *generalized feedback shift register generators* (GFSR). An example of these generators is the **Mersenne Twister**, developed by Matsumoto and Nishimura in 1997. The name comes from the period length form $2^k - 1$ which is chosen to be a prime (primes of this form are called Mersenne primes). The most common version of the algorithm is based on the Mersenne prime $2^{19937} - 1$, known at MT19937.

The algorithmic details of the Mersenne Twister are beyond the scope of this project.

3.3 Randomized Quasi Monte Carlo Methods

Quasi Monte-Carlo Methods are concerned with low discrepancy sequences. We want to define the Halton sequence, which is a generalization of the Van der Corput sequence. We first begin with a definition of uniformity.

Definition 5. *A sequence $q_1, q_2, \ldots$ in $[0,1)$ is said to be* ***uniformly distributed modulo 1*** *(u.d. mod 1) if*

$$\lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} \mathbf{1}_{[a,b)}(q_n) = b - a$$

for any $0 \leq a < b \leq 1$, where $\mathbf{1}_{[a,b)}$ is the indicator function, that is, $\mathbf{1}_{[a,b)}(q_n) = 1$ if $q_n \in [a,b)$ and 0 otherwise.

We can generalize this to the s-dimensional unit cube in the obvious way. Then we can state the primary result for our use.

Theorem 3.6. *A sequence $q_1, q_2, \ldots$ in $[0,1)^s$ is u.d. mod 1 if and only if*

$$\lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} f(q_n) = \int_{(0,1)^s} f(x)\, dx$$

for all Riemann integrable functions f on $(0,1)^s$.

This is a theoretical result that describes a quadrature algorithm for sequences of this type. We can characterize these sequences in terms of discrepancy.

Definition 6. *The* ***extreme discrepancy*** *of the sequence $q_1, \ldots, q_N$ in $[0,1)^s$ is*

$$D_N(x_n) = \sup_{[a,b)} \left| \frac{1}{N} \sum_{n=1}^{N} \mathbf{1}_{[a,b)}(x_n) - \lambda_s([a,b)) \right|.$$

If the supremum is taken over all intervals of the form $[0,a)$, then we obtain the star-discrepancy:

$$D_N^*(x_n) = \sup_{[0,a)} \left| \frac{1}{N} \sum_{n=1}^{N} \mathbf{1}_{[0,a)}(x_n) - \lambda_s([0,a)) \right|.$$

The two discrepancy notions are related in the following way: For any finite set $P = \{q_1, \ldots, q_N\}$, we have

$$D_N^*(P) \leq D_N(P) \leq 2^s D_N^*(P).$$

Finally, the relevant theorem is the following.

Theorem 3.7. *$D_N(q_n) \to 0$ if and only if $\{q_n\}$ is a u.d. mod 1 sequence.*

Thus low-discrepancy sequences are crucial to numerical quadrature problems and define the quasi Monte-Carlo class of methods. The most basic of these sequences is the Van der Corput sequence.

Definition 7. *For a base b, if*

$$n = (a_k \cdots a_1 a_0)_b = a_0 + a_1 b + \cdots + a_k b^k,$$

then the nth element of the ***Van der Corput sequence*** *in base b is*

$$\phi_b(n) = (.a_0 a_1 \cdots a_k)_b = \frac{a_0}{b} + \frac{a_1}{b^2} + \cdots + \frac{a_k}{b^{k+1}}.$$

We can generalize the sequence to multiple dimensions by setting the Van der Corput sequence in different bases in different components of a vector. These are called Halton sequences (thus the Van der Corput sequence is a one-dimensional Halton sequence). The s-dimensional Halton sequence in the bases $b_1, \ldots, b_s$ is defined as:

$$\begin{bmatrix} \phi_{b_1}(1) \\ \vdots \\ \phi_{b_s}(1) \end{bmatrix}, \begin{bmatrix} \phi_{b_1}(2) \\ \vdots \\ \phi_{b_s}(2) \end{bmatrix}, \ldots, \begin{bmatrix} \phi_{b_1}(n) \\ \vdots \\ \phi_{b_s}(n) \end{bmatrix}, \ldots$$

To preserve uniformity, we use the following result.

Theorem 3.8. *The Halton sequence is u.d. mod 1 if its bases $b_1, \ldots, b_s$ are relatively prime.*

There are further theoretical results that can be developed, the major one being the *Koksma-Hlawka Inequality*, which provides a loose upper-bound to the quadrature error. We can also improve practical error analysis of Quasi Monte-Carlo methods by introducing techniques known as Randomized Quasi-Monte Carlo Methods; two techniques we test here are known as:

- Random shift: we take a Halton sequence and add a random shift $\mathbf{u}$, with components random numbers in $[0, 1)$ and then take the fractional part (i.e. modulo 1).
- Random start: the theory for randomly started Halton sequences is the beyond the scope here, but we in effect force a delay into the sequence.

4 A Quadrature Problem: Implementation

We will consider the quadrature of the function $f : \mathbb{R} \to \mathbb{R}$ defined by

$$f(x) = e^x$$

over the interval $I = [0.1]$
Thus, we want to compute the following definite integral:

$$\int_0^1 e^x \, dx$$

We now discuss our implementation of this problem. For each method, we fix parameter N (the number of points or order of the method) and m (the number of estimates computed). For each method we generate a sequence of the form $(x_i)_{i=1}^N$ and compute

$$\theta_k = \frac{1}{N}\sum_{i=1}^{N} e^{x_i}$$

We then compute m of these estimates for each method, and then find the mean and standard deviation of our estimation for each method (by default, we present this in a box-plot for easy comparison). We change values of m and N to determine how each method behaves.

The code is written in Python on a Jupyter Notebook to allow easy task separation (see .ipynb file). We can now discuss the algorithmic detail for generating each instance of a sequence for each method:

- **Mersenne Twister**: We use the default python library Random whose subroutine Random.random() is by default the Mersenne Twister (MT19937, with 53-bit precision specifically). This has a period of $2^{19937} - 1$. This returns a pseudo random number in the interval $[0, 1)$ by default.
- **Halton Sequence**: Since we are in 1 dimension, we simply produce the Van der Corput sequence. This is done as follows. Start with the base 2 representation of the index n, and then invert this (reflect across the decimal point), and finally, convert back to base 10. Algorithmically, we simply compute the base two representation of n, and then sum the array backwards with appropriate base coefficients.
- **Random Shift**: We use the default pseudo random number generator to generate a shift, which we then add to the Van der Corput sequence array, and then simply mod out by 1.
- **Random Start**: To compute the Random start, we again start with a pseudo random number, say u. We then compute its base 2 representation up to some given precision tolerance (e.g 2^{-53}. This is achieved by choosing the maximum allowed bits (=53). This corresponds to a close 2-adic number to u, and then we invert, and convert back to base 10 to obtain a delay k for our Van der Corput Sequence. The details of this routine are not important for our purpose.

The subroutines for these tasks are available in the Jupyter notebook as well as the appendix at the end. The code for presenting results and generating plots was generated with the help of generative AI (specifically, ChatGPT 4o).

5 Results and Methodology

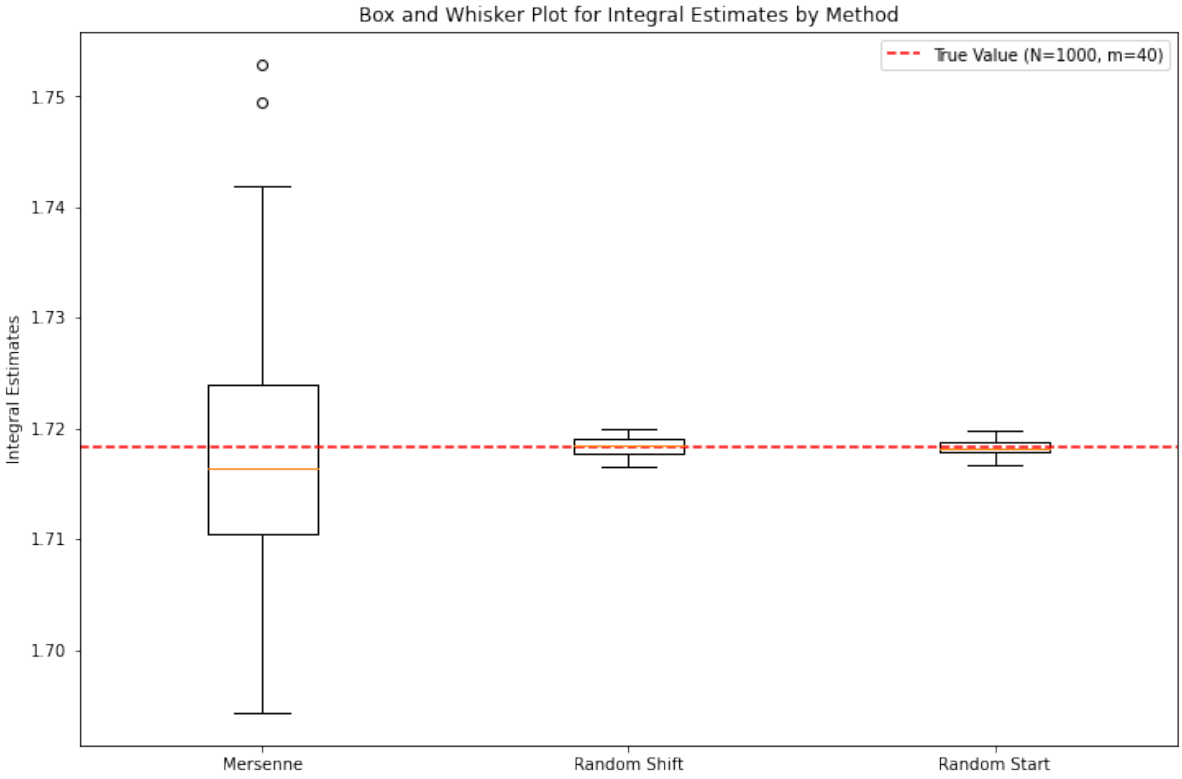
(Homework 1.2) First we compute the true value of the integral analytically:

$$\int_0^1 e^x \, dx = [\, e^x \,]_0^1 = e^1 - e^0 = e - 1$$

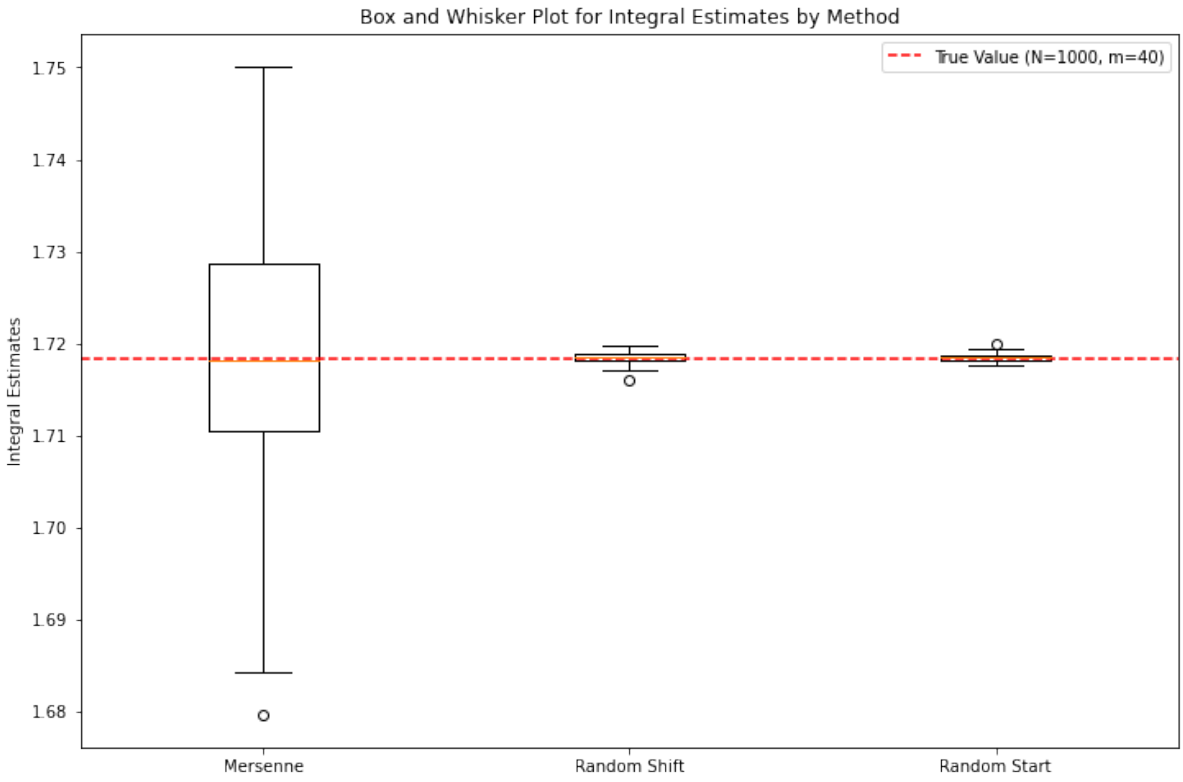
This is approximately

$$e - 1 \approx 1.7182818284590452353602874713 52$$

We present a few sample runs for our experiment for the quadrature and provide statistics for the estimates from both methods. These are presented in Box and Whisker plots for easy visualization. For this particular run, we set $N = 1000$ and $m = 40$.

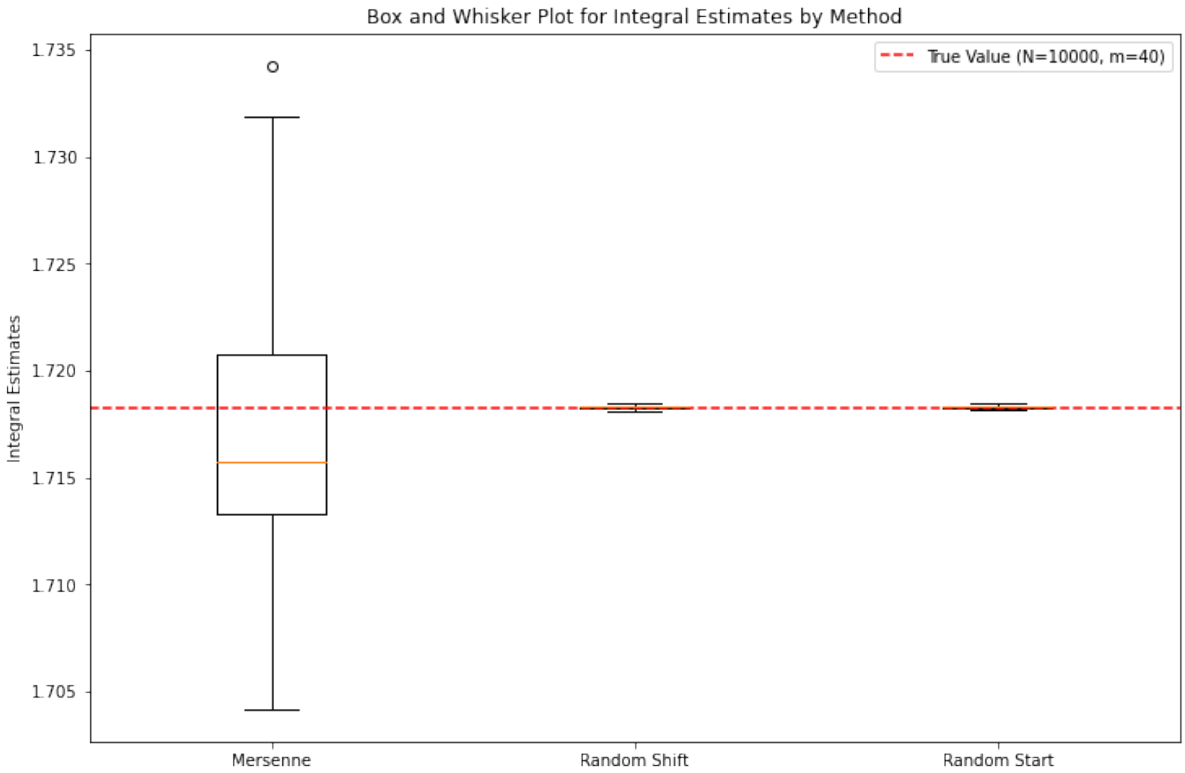


	Method	Mean	Standard Deviation
0	Mersenne	1.718280	0.013646
1	Random Shift	1.718377	0.000894
2	Random Start	1.718308	0.000720



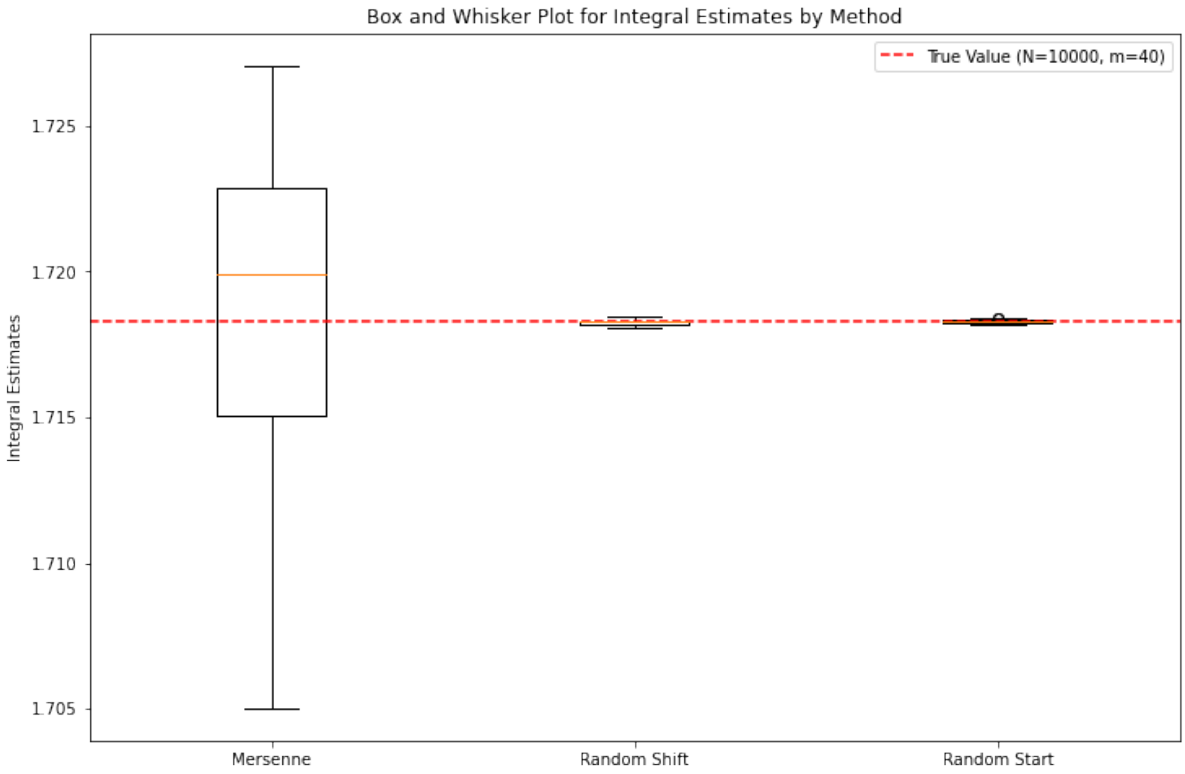
	Method	Mean	Standard Deviation
0	Mersenne	1.718743	0.014931
1	Random Shift	1.718523	0.000748
2	Random Start	1.718530	0.000520

Now we set $N = 10000$ and keep $m = 40$. The results are as follows:



	Method	Mean	Standard Deviation
0	Mersenne	1.717008	0.006345
1	Random Shift	1.718295	0.000081
2	Random Start	1.718310	0.000056

Another such run looks like:



	Method	Mean	Standard Deviation
0	Mersenne	1.717008	0.006345
1	Random Shift	1.718295	0.000081
2	Random Start	1.718310	0.000056

Important conclusions: We can immediately begin to hypothesize from these results:

- Each method approximates the integral at least up to 2 decimal places (we can see that the true value is always within the upper and lower quartile); we generally expect this result to not generalize and the condition numbers to be functions of where the interval lies, and the function we are approximating.
- The Mersenne Twister is the least reliable of these three methods. It has the largest standard deviation (orders of magnitude larger than HS), largest range and the worse outliers. The accuracy of the Mersenne Twister is also unreliable; whenever it does beat the randomized quasi Monte-Carlo sequences, it does so by chance.
- Randomly shifted and Randomly started Halton sequences are accurate to $\sim$ 3 or 4 decimal places. Accuracy improves marginally with larger sizes of the sequence (N).
- Mersenne Twister behaves similarly with larger sample sizes.
- Randomly started Halton sequences are more precise than randomly shifted ones (not necessarily more accurate); they generally have smaller standard deviation.

We can test these hypotheses more rigorously by conducting larger experiments and observing trends more generally.

5.1 Large Parameterized Set of Experiments

We now vary the parameter N (the number of samples) and fix m (=40) and compute the complete estimate (mean for each separate run of the experiment) and evaluate the error from the true value as follows:

$$\text{Error}(N;m) = \left| \frac{1}{m} \sum_{k=1}^{m} \theta_k^{(N)} - \text{true value} \right|$$

We then graph the error as sample grows larger for each of the methods. This gives us a more complete picture of the reliability of the algorithms.

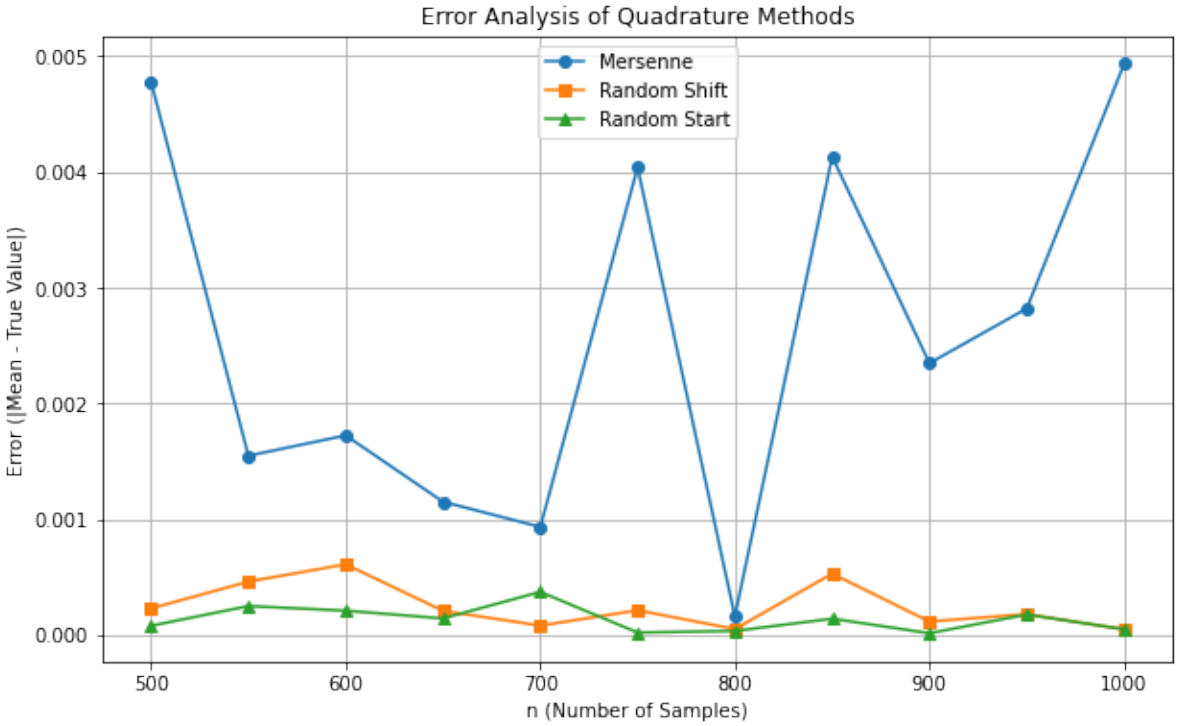


Figure 1: Error of Mean as a function of N (m=40)

We can also now remove the second layer of averaging and set m=1 to see how the algorithms behave per iteration as the number of points grows larger. Note that for each iteration on n, the estimates are still independent.

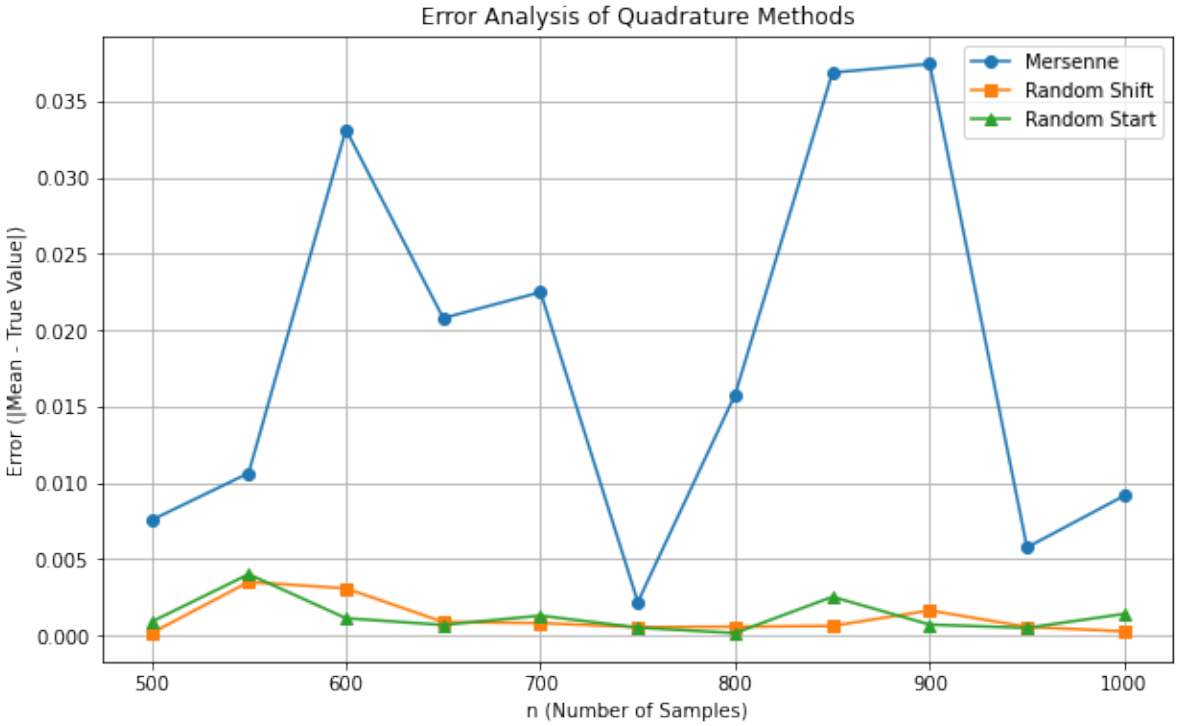


Figure 2: Error with one estimate for each N (m=1)

These plots corroborate our earlier hypotheses.
Important Conclusions (for smaller N):

- Mersenne Twister is the least reliable method, and it is also the least accurate (largest error) in general.
- HS outperforms MS in reliability and precision (more consistent results) and accuracy (smaller error) to a considerable degree, in general.
- Random shift and Random start perform similarly.
- Higher amounts of averaging (i.e. more estimates, larger m) improves results across the board by orders of magnitude; this can be observed from the magnitude on the y-axes.
- With averaging, Random start tends to be somewhat more accurate and more precise than Random shift.

We can repeat this experiment with more frequent data points:

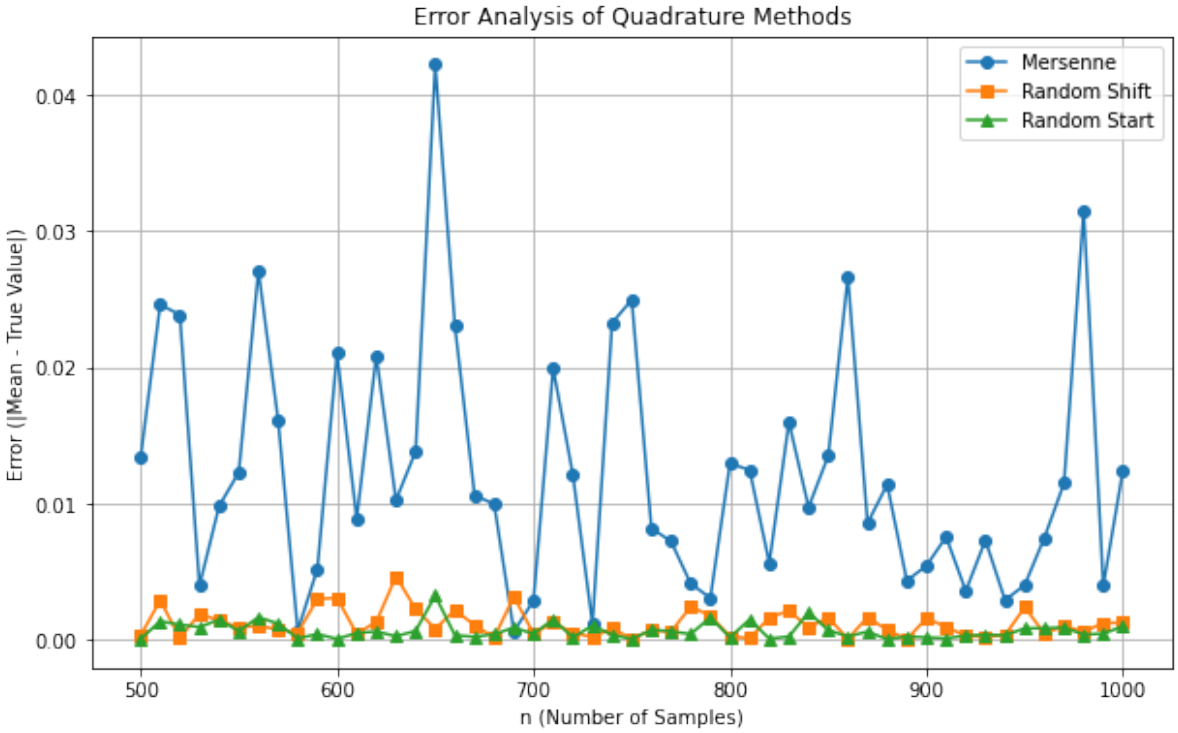


Figure 3: Data point at every 10 (m=1)

These results provide even more **evidence** at this level ($500 \leq N \leq 1000$) for our earlier claims.
Finally, we regenerate the plot for larger values of N to look at a complete picture (from 5000 - 10000 with a data point every 500): At this level we observe similar results. Accuracy is improved across the board by about double the significant figures as before. Random Start and Random Shift behavior becomes more uniform. Mersenne Twister is still less reliable and worse performing.

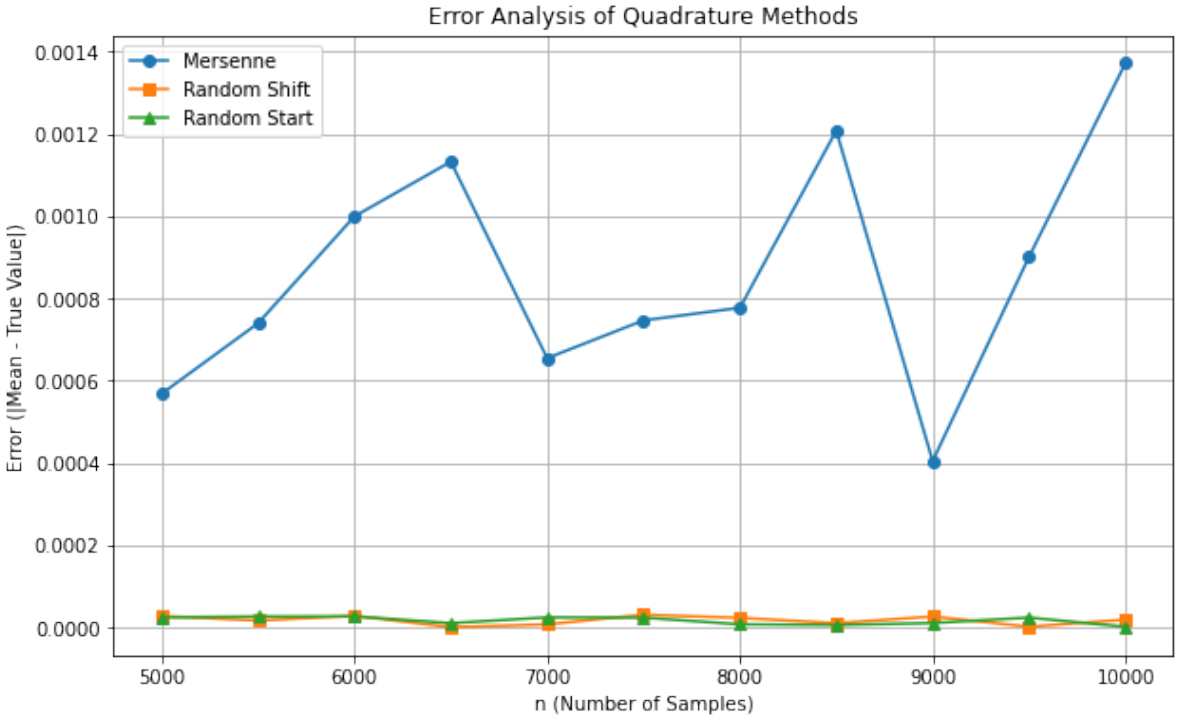


Figure 4: Large N (with m=40)

5.2 Conclusions

The behaviors we have observed allows us to establish a hierarchy of these algorithms which is in line with theoretical prediction. Since RQMC sequences are low discrepancy, they are more suited to numerical quadrature. Convergence results still hold in general as we see improvement in accuracy as $N \to \infty$. These results have been shown rigorously and successfully through numerical experiments.

6 Code Appendix

6.1 Library Packages

```
import numpy as np                  # For array handling
import random                       # For Mersenne Twister (uniform)
import matplotlib.pyplot as plt     # For data plotting
import pandas as pd
```

6.2 Pseudo Random Number Generator Routines

```
def float_to_binary(num, max_bits=53):
    if not (0 <= num < 1):
        raise ValueError("Number must be between 0 and 1
                         (inclusive of 0, exclusive of 1).")

    binary_list = []
    for _ in range(max_bits):
        num *= 2
        bit = int(num)
        binary_list.append(bit)
        num -= bit
        if num == 0:
            break

    return np.array(binary_list)

def random_start(b_adic):
    sum = 0
    for i in range(np.size(b_adic)):
        sum = sum + b_adic[i] * (2**i)

    return sum

def random_shift(vds, shift):
    vds_shifted = (vds + shift) % 1

    return vds_shifted

def van_der_corput(num):    # Generates the i-th element of the Van de Corput sequence
    if(num < 0):
        raise ValueError("The index for the Halton sequence has to be
                         greater than equal to 0.")

    bin_list = [int(i) for i in list('{0:0b}'.format(num))]
    bin_list.reverse()
```

```
    bin_length = np.size(bin_list)

    sum = 0
    for i in range(bin_length):
        sum = sum + ( bin_list[i] / 2**(i+1))

    return sum

def vdc_start(size, tol):   # Generates the randomly started n elements of
                            # the Van de Corput sequence
    if(size < 0):
        raise ValueError("The size of the sequence has to be greater than equal to 0.")

    u = random.random()
    max_bits = tol
    rep = float_to_binary(u, max_bits)
    start = random_start(rep)

    return np.array([van_der_corput((i+1)+start) for i in range(size)])

def vdc_shift(size):    # Generates the randomly shifted n elements
                        # of the Van de Corput sequence
    if(size < 0):
        raise ValueError("The size of the sequence has to be greater than equal to 0.")

    u = random.random()
    base_sequence =  np.array([van_der_corput((i+1)) for i in range(size)])
    shifted_sequence = random_shift(base_sequence, u)

    return shifted_sequence

def mt19937(size):      # Default Mersenne Twister (takes in size and generates a
                        # pseudo-random array of that size)
    if(size < 0):
        raise ValueError("The size of the sequence has to be greater than equal to 0.")

    return np.array([random.random() for _ in range(size)])
```

6.3 Quadrature Routines

```
def quadrature(support):
    size = np.size(support)
    if(size == 0):
       return 0
    vals = np.exp(support)                  # Change function here as a parameter
    integral = (np.sum(vals)) / size        # Compute the normalized sum
```

```
    return integral                          # return estimate

    def mt_quadrature(size,trials):
    estimates = np.zeros(trials)
    for i in range(trials):
        x_vals = mt19937(size)
        estimates[i] = quadrature(x_vals)

    return estimates

def vdc_quadrature_shift(size,trials):
    estimates = np.zeros(trials)
    for i in range(trials):
        x_vals = vdc_shift(size)
        estimates[i] = quadrature(x_vals)

    return estimates

def vdc_quadrature_start(size, trials, tol):
    estimates = np.zeros(trials)
    for i in range(trials):
        x_vals = vdc_start(size, tol)
        estimates[i] = quadrature(x_vals)

    return estimates
```

6.4 Single Experiment Run

```
n = 10000           # Size of Quadrature (number of terms)
m = 40              # Number of Trials

# For the random start algorithm
max_bits = 32       # This determines tolerence up to 2**(-max_bits)
                    # for the 2-adic number

true_value = np.e - 1.0

estimates_mt = mt_quadrature(n,m)
estimates_vdc_shift = vdc_quadrature_shift(n,m)
estimates_vdc_start = vdc_quadrature_start(n,m, max_bits)
```

6.5 Statistics and Plotting

```
# Calculate means and standard deviations
mean_mt, std_mt = np.mean(estimates_mt), np.std(estimates_mt)
```

```
mean_vdc_shift, std_vdc_shift = np.mean(estimates_vdc_shift), np.std(estimates_vdc_shift)
mean_vdc_start, std_vdc_start = np.mean(estimates_vdc_start), np.std(estimates_vdc_start)

# Create the box plot
plt.figure(figsize=(12, 8))
plt.boxplot([estimates_mt, estimates_vdc_shift, estimates_vdc_start],
            labels=['Mersenne', 'Random Shift', 'Random Start'])

# Add a line for the true value
plt.axhline(y=true_value, color='red', linestyle='--', label=f'True Value (N={n}, m={m})')

# Add labels, title, and legend
plt.ylabel('Integral Estimates')
plt.title('Box and Whisker Plot for Integral Estimates by Method')
plt.legend()

# Display the plot
plt.show()

# Create and display the table
data = {
    'Method': ['Mersenne', 'Random Shift', 'Random Start'],
    'Mean': [mean_mt, mean_vdc_shift, mean_vdc_start],
    'Standard Deviation': [std_mt, std_vdc_shift, std_vdc_start]
}
df = pd.DataFrame(data)

# Print the table to console
print(df)

# Optionally display the table with Matplotlib
plt.figure(figsize=(8, 3))
plt.axis('off')
table = plt.table(cellText=df.values, colLabels=df.columns, loc='center', cellLoc='center')
table.auto_set_font_size(False)
table.set_fontsize(12)
plt.title('Mean and Standard Deviation of Integral Estimates')
plt.show()
```

6.6 Large Parameterized Set of Experiments

```
n_values = np.arange(5000, 10001, 500)
# Arrays to store errors
errors_mt = []
errors_vdc_shift = []
errors_vdc_start = []
```

```
# Loop through each value of n, calculate means and errors
for n in n_values:
    estimates_mt = mt_quadrature(n, m)
    estimates_vdc_shift = vdc_quadrature_shift(n, m)
    estimates_vdc_start = vdc_quadrature_start(n, m, max_bits)

    mean_mt = np.mean(estimates_mt)
    mean_vdc_shift = np.mean(estimates_vdc_shift)
    mean_vdc_start = np.mean(estimates_vdc_start)

    # Calculate absolute error for each method
    errors_mt.append(abs(mean_mt - true_value))
    errors_vdc_shift.append(abs(mean_vdc_shift - true_value))
    errors_vdc_start.append(abs(mean_vdc_start - true_value))

# Plot the errors as a function of n
plt.figure(figsize=(10, 6))
plt.plot(n_values, errors_mt, marker='o', label='Mersenne')
plt.plot(n_values, errors_vdc_shift, marker='s', label='Random Shift')
plt.plot(n_values, errors_vdc_start, marker='^', label='Random Start')

# Add labels, title, and legend
plt.xlabel('n (Number of Samples)')
plt.ylabel('Error (|Mean - True Value|)')
plt.title('Error Analysis of Quadrature Methods')
plt.legend()
plt.grid(True)

# Display the plot
plt.show()
```