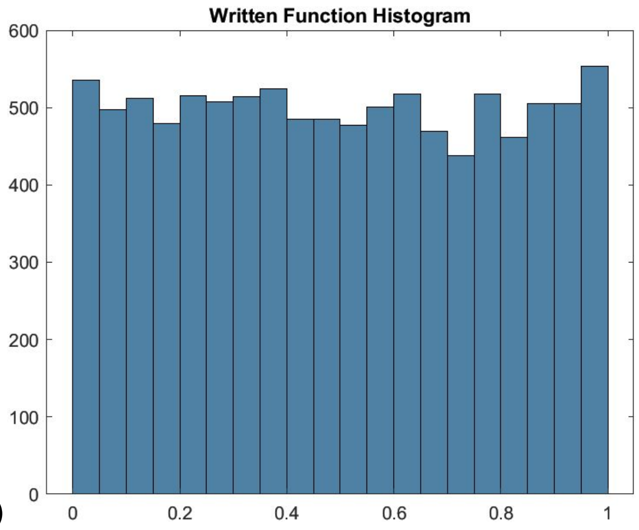
Nathan Jo, Edward Holappa, Griffin Weinhold

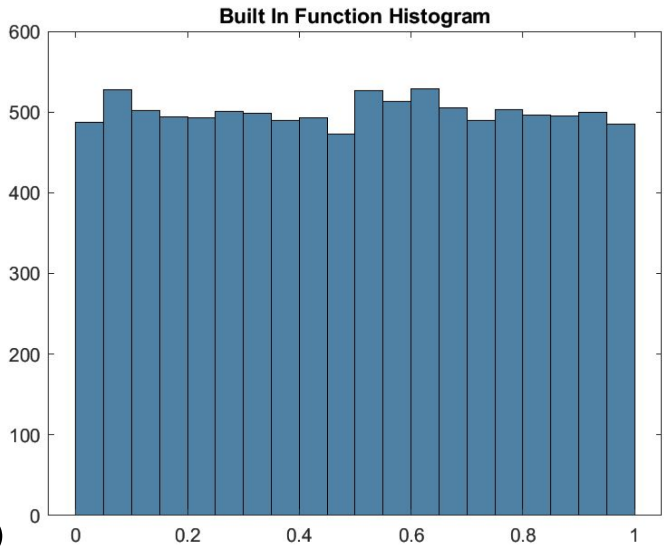
MATH 458 Final Project: Monte Carlo

QUESTION 1

We first write a program that executes the equation given: $x_n = (ax_{n-1} + c) \bmod(m)$ where $a = 7^5$, $c = 0$ and $m = 2^{31}-1$. To create the data for part a), we have to initialize the recurrence relation with a number between 0 and 1. Then we let the function continue until n (n = 10,000) numbers are generated. Using the built-in function, we just need to use rand(1,n)



a)



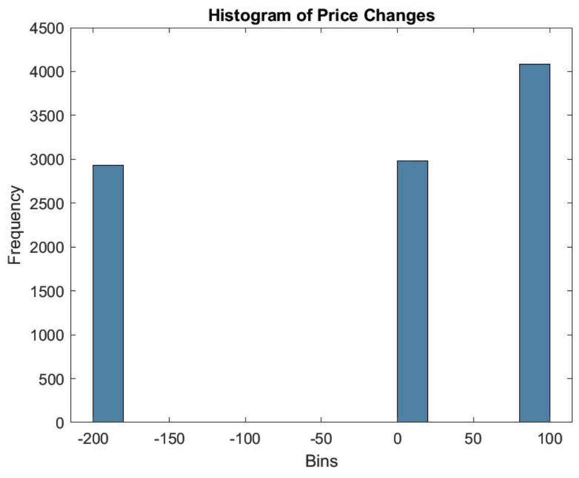
b)

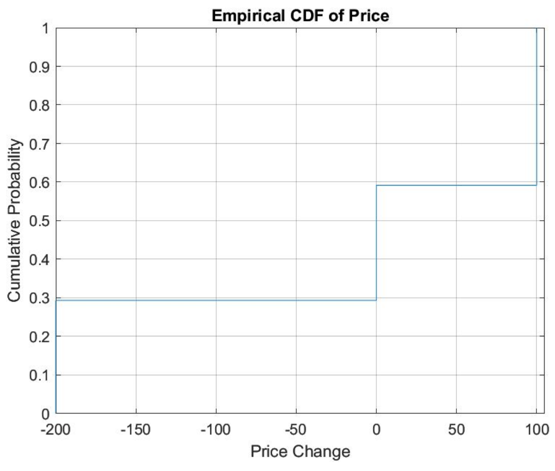
c) The two histograms are very similar. Both functions produce a uniform distribution in [0,1], so their similarity is expected.

QUESTION 2

In this problem, we use the numbers generated in part a) of question 1 to produce 10,000 instances of the daily price fluctuations of an asset. It can go up by 100 with probability 0.45, go down by 200 with probability 0.25, or stay the same with probability 0.3. Since the generated numbers are uniform between zero and one, we can complete this task by creating 3 ranges between 0 and 1 with the same magnitude as the previously mentioned probabilities. That is, if the observed value is between 0 and 0.45, we say the asset increased by 100, if it is between .45 and .7, we say it decreased by 200. If it is greater than .7 we say it stayed the same.

Histogram and Empirical Distribution Function:





The bar lengths represent the probability that a price fluctuation at that number will occur. And the CDF confirms that the asset will either increase by 100, decrease by 200 or stay the same with probability 1.

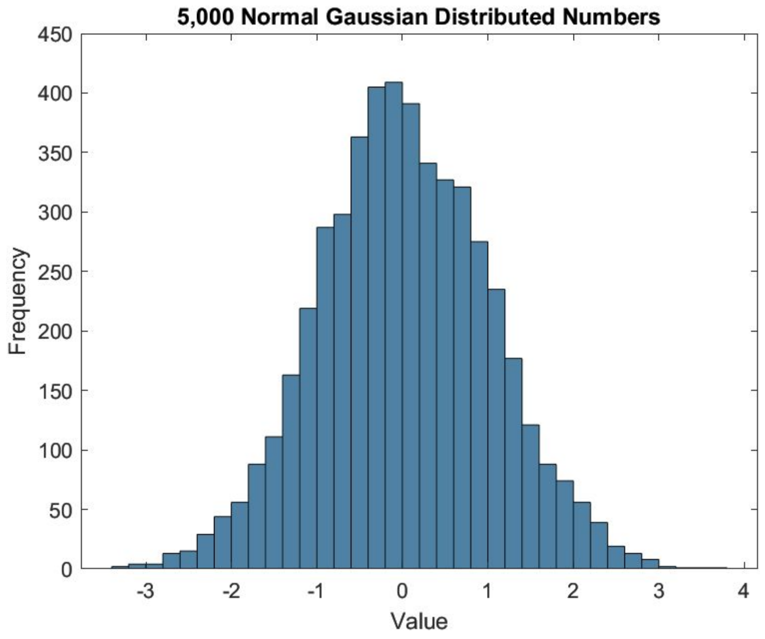
QUESTION 3

In this question, we use matlab’s binornd(n,p) function to generate the random numbers, with n = 70 and p = 0.7. Now, we have to calculate the probability that the random variable is less than 50.
We do this by sorting the array of 5000 observations, and then find the index that contains the greatest value that is less than 50. This value, call it ‘idx,’ is the number of instances the random variable is less than 50. We divide this by 5000 to get an empirical probability, which is close to the theoretical probability.

idx	Empirical probability	Theoretical probability	Error
2767	0.5534	0.545	1.52%

QUESTION 4

Here, we are asked to generate 5000 standard normal distributed numbers by using MATLAB's randn function. We do this by using the arguments: randn(1,5000). The histogram shown below:



Results and error:

	Observed	**Expected**	**Percent error**
Variance	0.96006	1	3.994%
Mean	0.0119	0	N/A

QUESTION 5

a)

We generate values N, each time increasing by a factor of 10, where N is the minimum number of the n observations of random variables such that their sum is greater than 1.

Iterations	**E[N]**
10	2.6
10^2	2.65
10^3	2.727
10^4	2.7226

10^5	2.72
10^6	2.71903

b) Our goal is to determine the expected number of Uniform Random Variables on [0,1] needed to sum up to 1:

$$E[N=\min\{n: \sum U_i>1\}]$$

Via the table it is clear the values are converging to natural constant *e*. This value is roughly 2.71828, and as we can see it is nearing this value. We can provide an elementary proof of this by breaking the question further.

We can find the Expected Value of numbers needed to surpass 1. This can be shown as

$$E[N] = 1 + \sum_{k>=1} P(N > k)$$

Where N > k if the summation of the first terms did not exceed 1. We can see with k = 1, 2, 3 that P(N > k) = 1/k!. This makes sense since when k = 1 we know the probability P(N > 1) is 1, and further with P(N > 2) this would be 1/2!. This carries us to the truth that:

$$E[N] = 1 + \sum_{k>=1} 1/k!$$
$$E[N] = 1 + e - 1$$
$$E[N] = e$$

QUESTION 6

a) Our goal is to estimate $P(X > 5)$ using the Monte Carlo method.
By definition of expected values,

$$E_x(1_{x>5}) = \int_{-\infty}^{\infty} 1_{x>5}(x) * f(x)dx$$

Where

$$1_{x>5}(x) \text{ -- Identity function when } x > 5$$
$$f(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$$

The Monte Carlo method relies on generating n random variables y_i from $Y \sim N(0,1)$ following

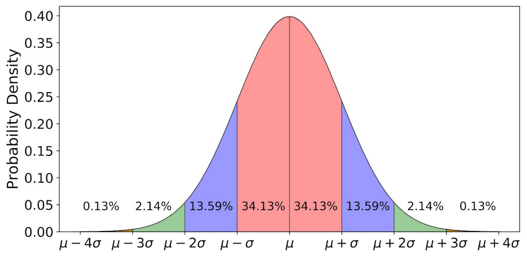
$$z_i = 1_{y>5}(y_i)$$
$$\widehat{p} = \frac{1}{n}\sum_{i=1}^{n} z_i$$

Implementing this is rather simple; we are merely counting how many times a value greater than 5 occurs in proportion to the total samples.

RESULT:

n	10	100	1,000	10,000	100,000	1,000,000
Estimator	0	0	0	0	0	0

As expected, even with 10^6 iterations, the outcoming probability is 0 because a value greater than 5 is almost improbable in a standard normal distribution. In the image, 5 does not even show in the distribution because it has a negligible area. Hence, a different approach should be used.



b) Importance Sampling

Importance sampling relies on making another random variable that generates values concentrated in the region of importance. In this case, it is values over 5. So, we introduce a new random variable $Y \sim N(5.5,\ 0.5)$ and transforming the integral for expected values.

$$E_Y = \int_{-\infty}^{\infty} 1_{y>5}(y) * \frac{f(y)}{g(y)} * g(y)dy$$

Where

$$g(y) = \frac{1}{\sqrt{\pi}} e^{-\frac{(y-5.5)^2}{2*0.5}}$$
$$f(y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}}$$

By definition of expected values, we need to find

$$E_Y\left(1_{y>5}(y)\frac{f(y)}{g(y)}\right)$$

The Monte-Carlo method on this model means that we generate a random y_i from $Y \sim N(5.5,\ 0.5)$ and finding

$$z_i = 1_{y>5}(y_i)\frac{f(y_i)}{g(y_i)}$$

$$\widehat{p} = \frac{1}{n} \sum_{i=1}^{n} z_i$$

RESULTS: This almost negligible value is expected for $P(X > 5)$

n	10	100	1,000	10,000	100,000	1,000,000
Estimator	1.52 e-07	2.62 e-07	3.14 e-07	3.61 e-07	3.54 e-07	3.57 e-07

We can conclude that $P(X > 5) \approx 3.57 * 10^{-7}$

c) As expected, Importance Sampling added an extra level of accuracy because the samples were more concentrated in the region of concern, $X > 5$. The Importance Sampling method was repeated with varying n and consistently produced non-zero values, meaning it is consistently more accurate.

<u>QUESTION 7</u>

a) We are asked to approximate the integral with a simple monte carlo integration method. We will evaluate the integrand at random points (A,B) where both A and B are uniformly distributed on [0,1]. After doing this 1000 times, we average the results. This gives us a sample approximation. We then collect 1000 sample approximations and take their average. Essentially, we are averaging the results from 1000 simulations of 1000 numbers each.

Results:

Estimate	4.8941
Variance	0.0340
95% Confidence Interval	[4.5324, 5.2557]

b) Now we use antithetic variates to reduce the variance, and tighten our confidence interval. In this method, we create pairs of random variables (x,y) where a,b $\in$ [0,1], and take their average, call this U. If Cov(x,y) < 0, then Var(U) < (½) $\boldsymbol{\sigma}^2$, <u>reducing overall variance.</u> Instead of making another uniform distribution, we pair each x with (1-x). (x,(1-x)) is clearly negatively correlated, and it ensures that each *pair* is independent. For consistency, we do this with data from part a).

Results:

	Value	**Improvement from part a)**

Estimate	4.8975	N/A, expected values are equal
Variance	0.0116	65.88% smaller
95% Confidence Interval	[4.6862, 5.1087]	41.59% tighter

The variance decreased significantly. This is because the negative correlation between our pairs was incredibly strong.

c) With control variates, we have to find a random variable C that is positively correlated to our data. We create a new variable Y = X - (C-E(C)), which will have the same mean, but a smaller variance if C is highly positively correlated with X. Let U,V ~Uniform(0,1), we use control variates $Y_1 = U + V$ and $Y_2 = (U+V)^2$

Results:

Y_1	**Value**	**Improvement from part a)**
Estimate	4.8942	N/A, expected values are equal
Variance	0.0306	10% smaller
95% Confidence Interval	[4.5515, 5.2369]	5.24% tighter

Y_2	**Value**	**Improvement from part a)**
Estimate	4.8943	N/A, expected values are equal
Variance	0.0262	22.94% smaller
95% Confidence Interval	[4.5770, 5.2116]	12.26% tighter

d) The most successful variance reduction technique is clearly the antithetic method. Creating input pairs with perfectly negative correlations allowed for maximum variance reduction. The control variate methods were also successful, but not to the same extent. Y_2 was better because the integrand also squares its two inputs, so it has a stronger positive correlation than Y_1 , which doesn't square the inputs.

QUESTION 8

a)

The fundamental equation used for the Monte-Carlo simulation to find call price is as follows:

$$Y_i = e^{-rT}[S_0 * exp\{(r - \frac{1}{2}\sigma^2) * T + \sigma\sqrt{T}Z_i)\} - K]$$

Where Z_i is random variable with standard normal distribution ($\sigma\sqrt{T}Z_i$ replaces the Wiener process W_T). This is derived from the Geometric Brownian Motion

$$S_t = S_0 * exp(r - \frac{1}{2}\sigma^2) * T + \sigma\sqrt{T}Z_i)$$

and the payoff function of a call option:

$$Profit = max(S_t - K, 0)$$

Then the average over all i is taken to estimate the call price (Lu).

$$\widehat{C} = \frac{1}{n}\sum_{i=1}^{n} Y_i$$

Refer to part c for the results. Additionally refer to function *getMCPrice* in the Appendix for the Python code.

b) The Black-Scholes Formula is an exact calculation of the call price (Lu):

$$C(S,t) \ = \ S_0 N(d_1) - Ke^{-rT}N(d_2)$$

Where

$N(*) = P(X < \ *)$ -- the probability of * in the cumulative distribution function of a standard normal distribution

$$d_1 = \frac{ln(\frac{S_0}{K}) + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}} \ , \ d_2 = d_1 - \sigma\sqrt{T}$$

With the given conditions, the exact European call price is $11.67. Refer to function *getBlackScholesPrice* in the appendix.

c) Comparison of Results

Exact Value from Black-Scholes: $11.67

n	Monte-Carlo Estimation	Error Relative to B-S
100	$12.90	10.54%
1,000	$10.64	8.83%

10,000	\$11.37	2.57%
100,000	\$11.79	1.03%
1,000,000	\$11.66	0.09%

The Monte Carlo method has been repeated with different values of n, and all of them are compared to the exact Black-Scholes price. It is clear that the Monte Carlo estimation converges to the exact value with increasing n. This result makes sense because more random points in the sample will produce a more accurate estimation,

QUESTION 9

a)

Since we are not given a value for volatility--and the Black-Scholes Formula requires said variable--additional research was done. The AMZN stock has an implied volatility (calls) of 0.2499. Time can also be calculated by assuming that the time from today to January 2020 is 2 months, or $2/12 = 0.16666$ years. Using the same Python code from equation 8, the call price from B-S becomes **\$58.81.**

The Put-Call Parity Relation is a method to derive Put price from Call price, and vice versa. We use this equation to find AMZN's corresponding Put price from the known Call price.

$$P(t) \ = \ C(t) - S(t) + K * B(t,T)$$

Where

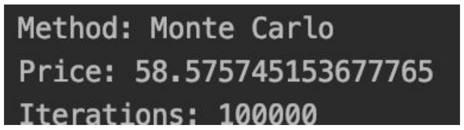
$B(t,T)$ is the present value factor for K. If the bond interest rate r is constant and very small,

$$B(t,T) = exp(-r(T-t))$$

In this case, $t = 0$, meaning the time frame has been calibrated to the present day. The *PCParity* function in Python (see Appendix) was added based on this equation, and the corresponding Put option is **\$88.17**.

b) We reuse the existing code from Question 8, this time only running it with one value of n = 100,000

Monte Carlo Method (n = 100,000)	Black-Scholes (Exact)
\$58.57	\$58.81


```
Method: Monte Carlo
Price: 58.575745153677765
Iterations: 100000
```

c) <u>Antithetic Variables Method: Variance Reduction</u>

One way to reduce variance is to use the antithetic variables method. The underlying principle is that the Monte Carlo method works best if the random variables follow a normal distribution as precisely as possible. Namely, we can enforce the symmetry of a standard normal distribution that has a median of 0. We generate a $-Z$ for every Z we produce, essentially creating negative duplicates (or the antithesis) of the same number.

$$Y_i = e^{-rT}[S_0 * exp\{(r - \tfrac{1}{2}\sigma^2) * T + \sigma\sqrt{T}Z_i)\} - K]$$
$$\overline{Y_i} = e^{-rT}[S_0 * exp\{(r - \tfrac{1}{2}\sigma^2) * T + \sigma\sqrt{T}(-Z_i))\} - K]$$

Since we want the same number of iterations for all methods, we will be taking n/2 random numbers because the other half will come from the $-Z$ conjugate.

$$\widehat{C} = \tfrac{1}{n}(\sum_{i=1}^{n/2} Y_i + \sum_{i=1}^{n/2} \overline{Y_i})$$

The main Monte Carlo equation/method, however, stays the same (Ødegaard). The Python code is under the function *getMCPrice_Antithetic*; the algorithm estimates the price to be **$58.67**

```
Antithetic Variables: 58.674493833216076
```

d) <u>Control Variates Method: Variance Reduction</u>

The Control Variates Method compares the error of the Monte Carlo algorithm with another known asset, and offsets our desired calculation by that amount of error. We will look at the same AMZN stock, but in this case calculate the exact put option using the Black-Scholes formula as a reference (i.e. control). Then we use the regular "vanilla" Monte Carlo method to estimate both the put and call options of the stock. The estimator using control variates will be as follows.

$$\widehat{c_{cv}} = \widehat{c} + (p_{bs} - \widehat{p})$$

Where

- p_{bs} = Put price from Black-Scholes
- $\widehat{p}$ = Put price estimator from Vanilla Monte Carlo method
- $\widehat{c}$ = Call price estimator from Vanilla Monte Carlo method
- $\widehat{c_{cv}}$ = Call price estimator using Control Variates

This is based on the underlying assumption that the errors of both call and put options are equal to each other.

$$\widehat{c_{cv}} - \widehat{c} = p_{bs} - \widehat{p}$$

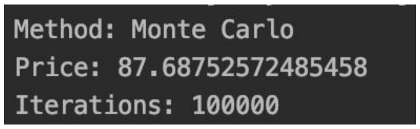
Hence, the control variates call estimator offsets the error by the difference of the put option estimator (Ødegaard).

1. Finding p_{bs} follows the same method as part a). The function *getBlackScholesPrice* has been modified to find both call and put options. It calculates the exact call option price, but if a put option is desired, it will call the PCParity function to calculate the corresponding put option price. Running the Python code produces an output of **$87.82.**

2. Estimating the put option price $\widehat{p}$ using the Monte-Carlo method requires another principal equation similar to the call option estimator.

$$Y_i = e^{-rT}[K - S_0 * exp\{(r - \tfrac{1}{2}\sigma^2) * T + \sigma\sqrt{T}Z_i)\}]$$

   This has been added to the getMCPrice function under the 'put' condition. Running the code with 100,000 iterations produces a result of **$87.69.**



```
Method: Monte Carlo
Price: 87.68752572485458
Iterations: 100000
```

3. We estimate $\widehat{c}$ using the usual Monte-Carlo method, which we have done in part b with a result of **$58.57.**
4. Finding $\widehat{c_{cv}}$

$$\therefore \widehat{c_{cv}} = 58.57 + (87.82 - 87.69) = 58.7$$

*Refer to the function *getMCPrice_CV* in the Appendix which implements this method.

<u>COMPARISON OF RESULTS</u>

Black-Scholes Value: $58.81

Monte Carlo Method (n=100,000)	Value	Error Relative to B-S
Vanilla Monte Carlo	$58.57	0.41%
Antithetic Variables	$58.67	0.24%
Control Variates	$58.7	0.19%

The results show that both the Antithetic Variable and Control Variates alteration outputs a better estimator of the call price than the Vanilla Monte Carlo method. This is expected because the former two reduces the variance of the random points generated from the normal distribution, effectively shrinking the confidence interval.

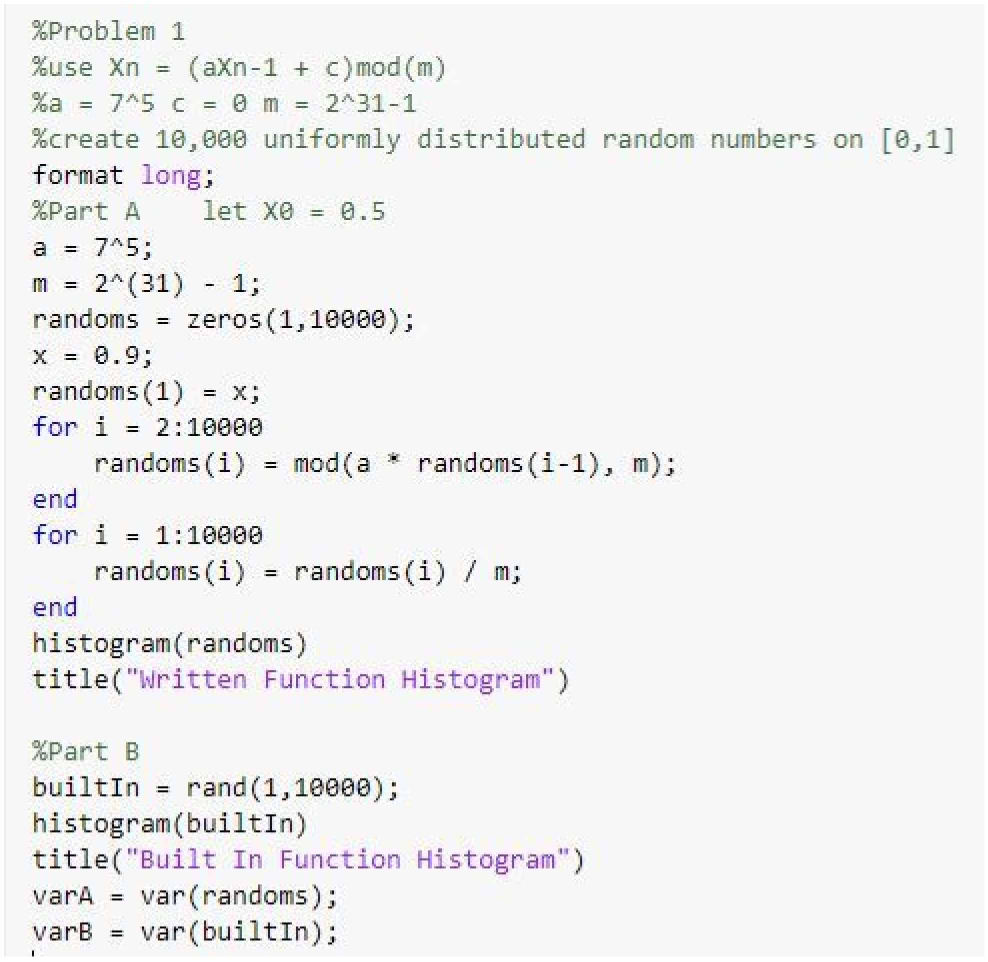
However, Monte Carlo methods are random in nature, so comparing the results of one iteration is not sufficient because they may not be reliable indicators of . Two ways to quantify the effectiveness/accuracy of all methods is to repeat them 1,000 times and compare the results' variances and average relative error.

Monte Carlo Method (n=100,000)	Average relative error after 1,000 iterations	Variance after 1,000 iterations
Vanilla Monte Carlo	4.59%	0.104
Antithetic Variables	3.51%	0.069
Control Variates	4.03%	0.092

As expected, the relative error and variance for the variance-reducing methods are lower than the Vanilla Monte Carlo method. This means they will consistently produce better results. Notice that Control Variates has a lower accuracy and higher variability than the Antithetic Variables method. In reality, the CV method gives freedom on what stock can be used as the control. Using AMZN's Put price as the control may not have been the most ideal, but this was the only option given limited information.

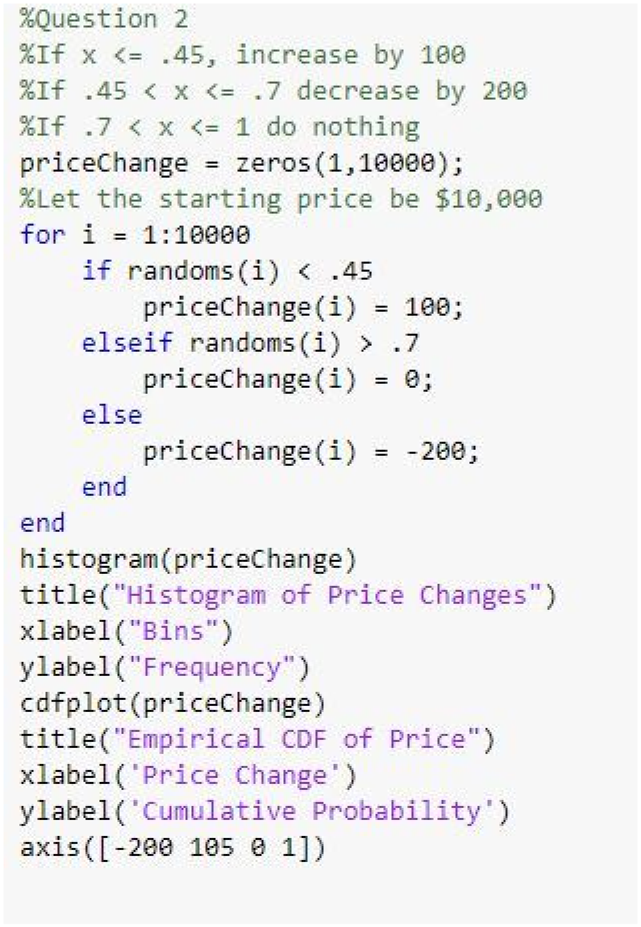
APPENDIX (Code)

QUESTION 1:



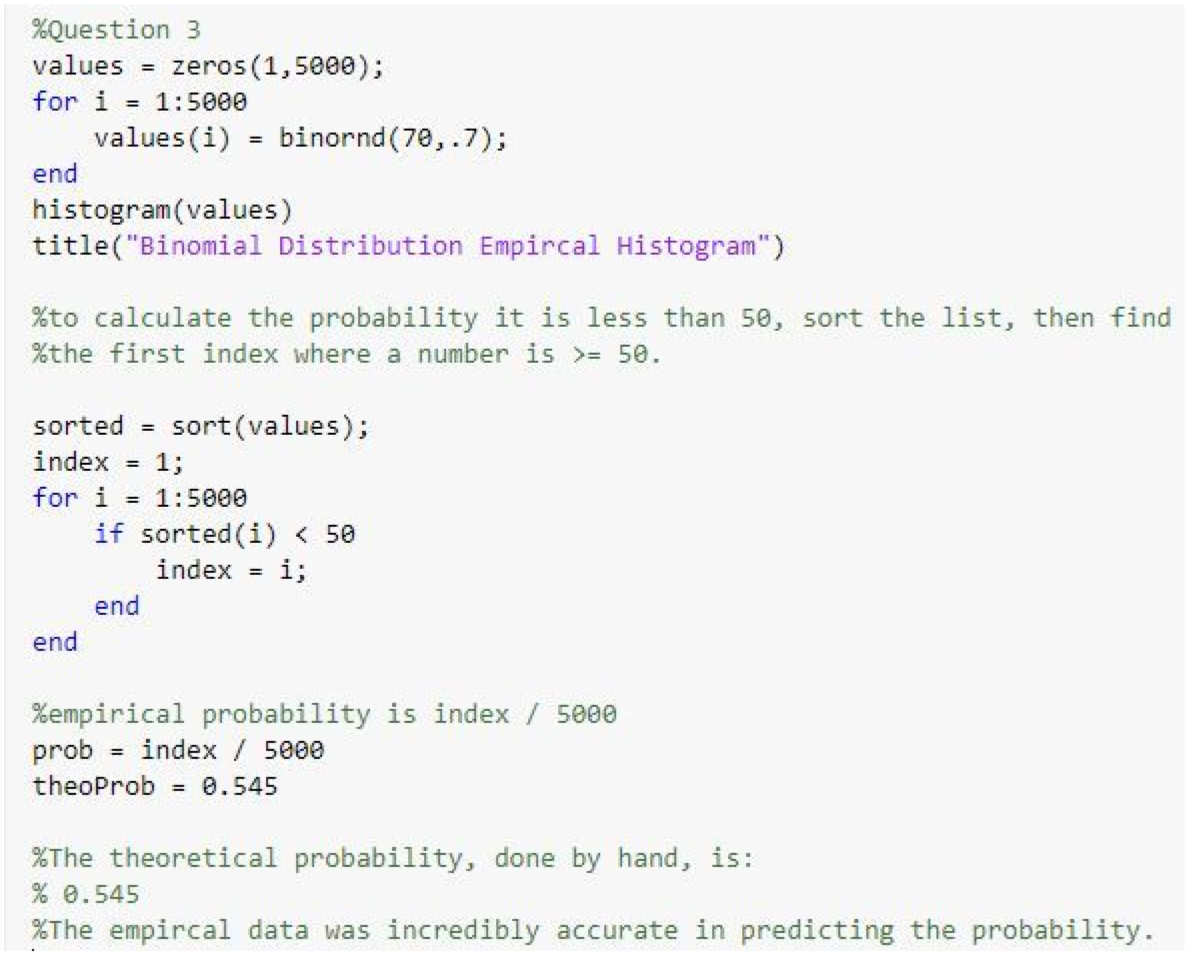
```
%Problem 1
%use Xn = (aXn-1 + c)mod(m)
%a = 7^5 c = 0 m = 2^31-1
%create 10,000 uniformly distributed random numbers on [0,1]
format long;
%Part A    let X0 = 0.5
a = 7^5;
m = 2^(31) - 1;
randoms = zeros(1,10000);
x = 0.9;
randoms(1) = x;
for i = 2:10000
    randoms(i) = mod(a * randoms(i-1), m);
end
for i = 1:10000
    randoms(i) = randoms(i) / m;
end
histogram(randoms)
title("Written Function Histogram")

%Part B
builtIn = rand(1,10000);
histogram(builtIn)
title("Built In Function Histogram")
varA = var(randoms);
varB = var(builtIn);
```

QUESTION 2:



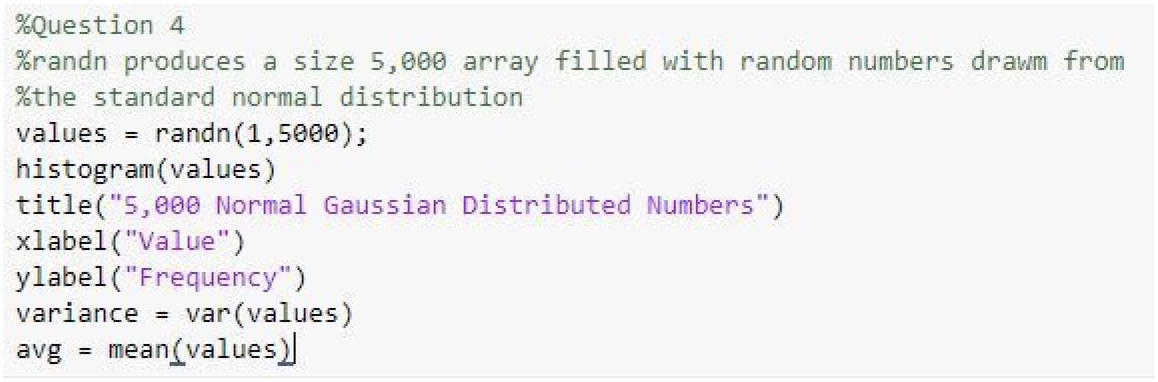
```
%Question 2
%If x <= .45, increase by 100
%If .45 < x <= .7 decrease by 200
%If .7 < x <= 1 do nothing
priceChange = zeros(1,10000);
%Let the starting price be $10,000
for i = 1:10000
    if randoms(i) < .45
        priceChange(i) = 100;
    elseif randoms(i) > .7
        priceChange(i) = 0;
    else
        priceChange(i) = -200;
    end
end
histogram(priceChange)
title("Histogram of Price Changes")
xlabel("Bins")
ylabel("Frequency")
cdfplot(priceChange)
title("Empirical CDF of Price")
xlabel('Price Change')
ylabel('Cumulative Probability')
axis([-200 105 0 1])
```

<u>QUESTION 3:</u>



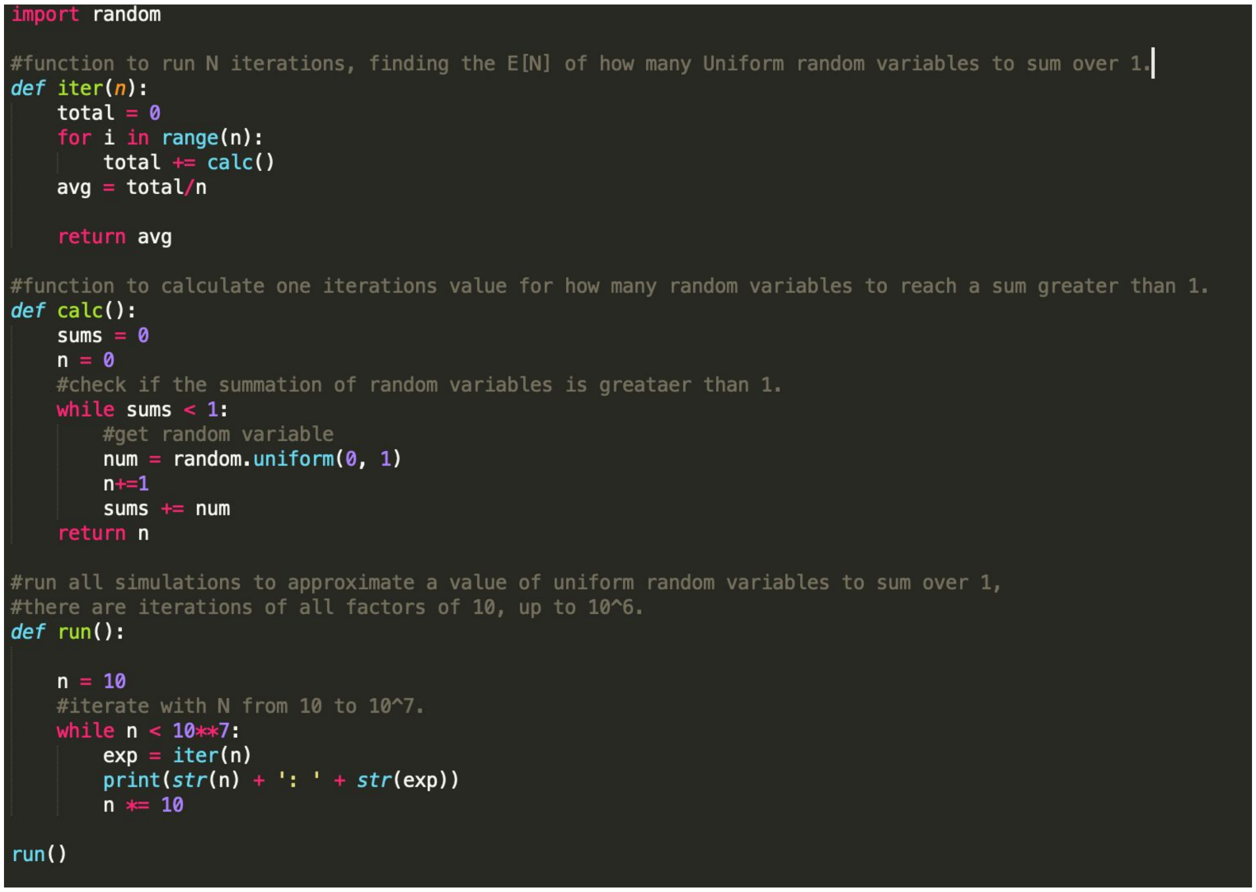
```
%Question 3
values = zeros(1,5000);
for i = 1:5000
    values(i) = binornd(70,.7);
end
histogram(values)
title("Binomial Distribution Empircal Histogram")

%to calculate the probability it is less than 50, sort the list, then find
%the first index where a number is >= 50.

sorted = sort(values);
index = 1;
for i = 1:5000
    if sorted(i) < 50
        index = i;
    end
end

%empirical probability is index / 5000
prob = index / 5000
theoProb = 0.545

%The theoretical probability, done by hand, is:
% 0.545
%The empircal data was incredibly accurate in predicting the probability.
```

QUESTION 4:



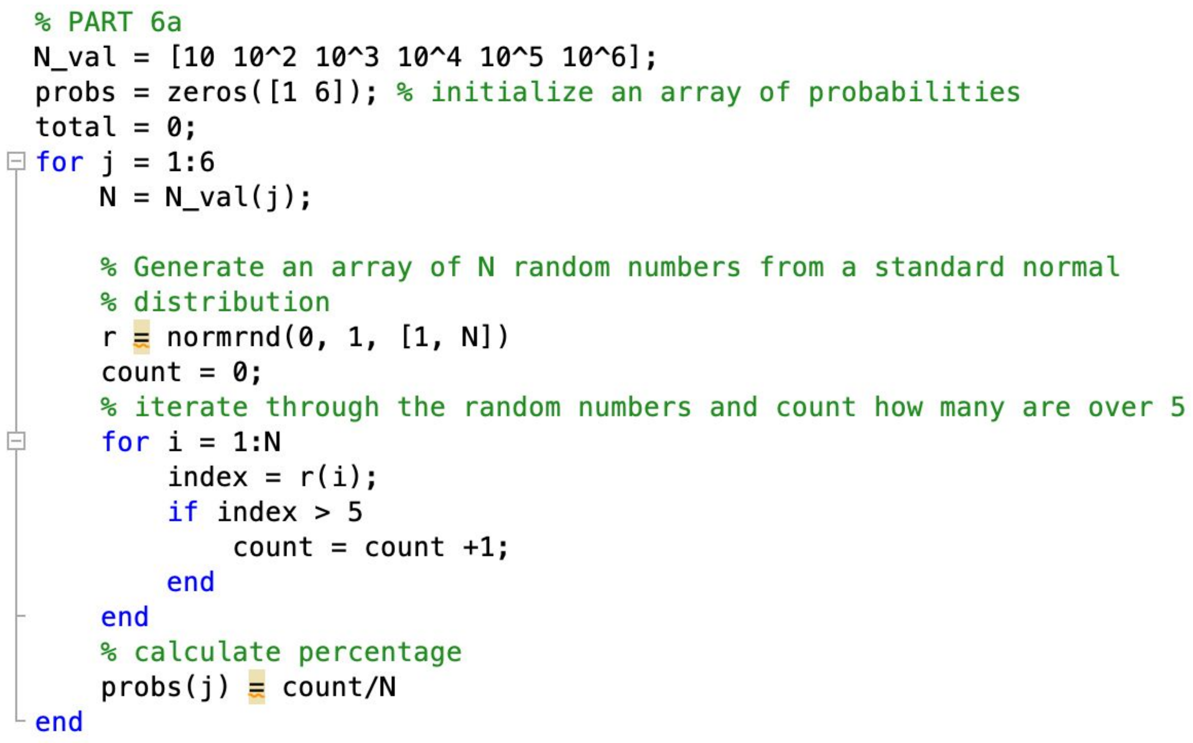
```
%Question 4
%randn produces a size 5,000 array filled with random numbers drawm from
%the standard normal distribution
values = randn(1,5000);
histogram(values)
title("5,000 Normal Gaussian Distributed Numbers")
xlabel("Value")
ylabel("Frequency")
variance = var(values)
avg = mean(values)
```

QUESTION 5:



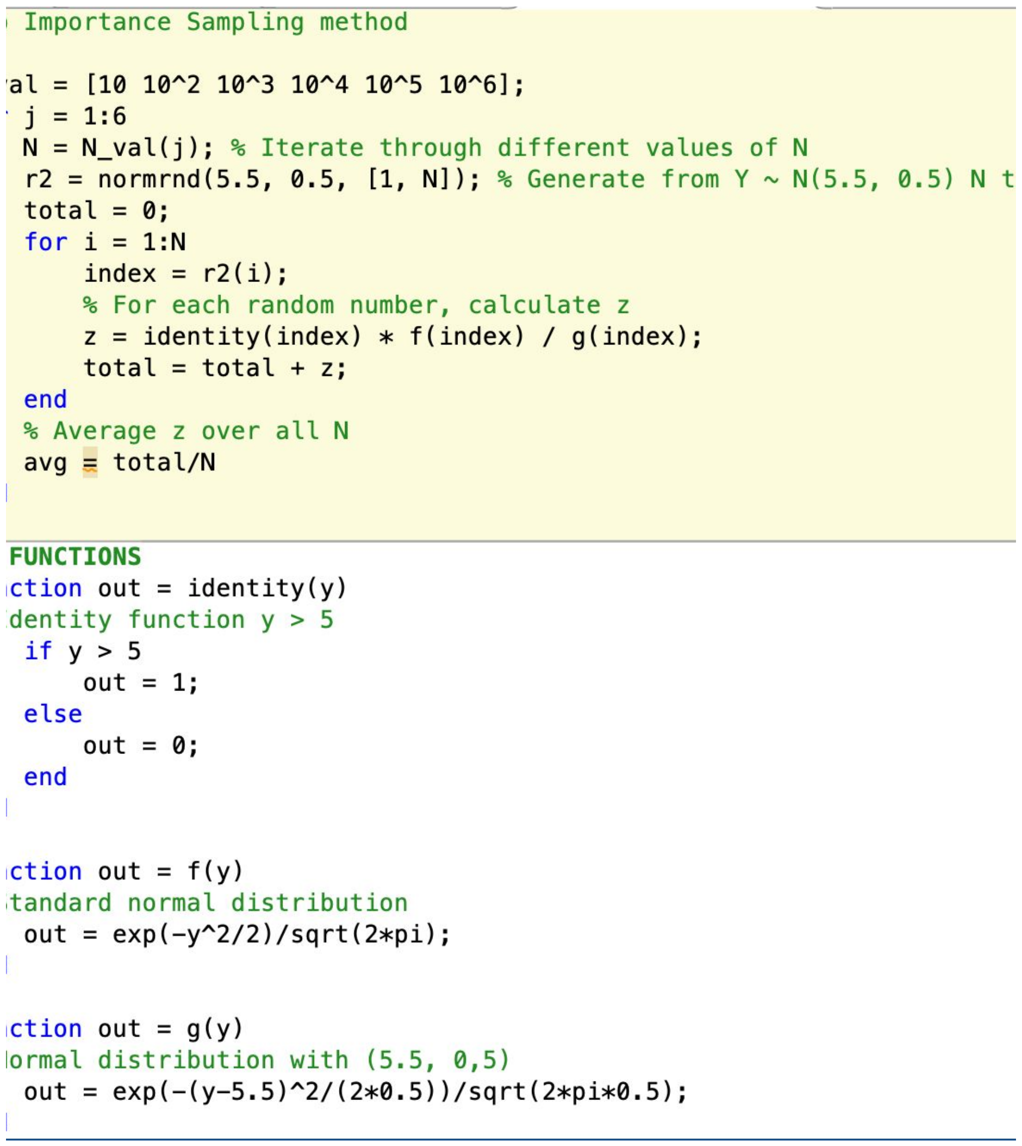
```python
import random

#function to run N iterations, finding the E[N] of how many Uniform random variables to sum over 1.
def iter(n):
    total = 0
    for i in range(n):
        total += calc()
    avg = total/n

    return avg

#function to calculate one iterations value for how many random variables to reach a sum greater than 1.
def calc():
    sums = 0
    n = 0
    #check if the summation of random variables is greataer than 1.
    while sums < 1:
        #get random variable
        num = random.uniform(0, 1)
        n+=1
        sums += num
    return n

#run all simulations to approximate a value of uniform random variables to sum over 1,
#there are iterations of all factors of 10, up to 10^6.
def run():

    n = 10
    #iterate with N from 10 to 10^7.
    while n < 10**7:
        exp = iter(n)
        print(str(n) + ': ' + str(exp))
        n *= 10

run()
```

QUESTION 6:



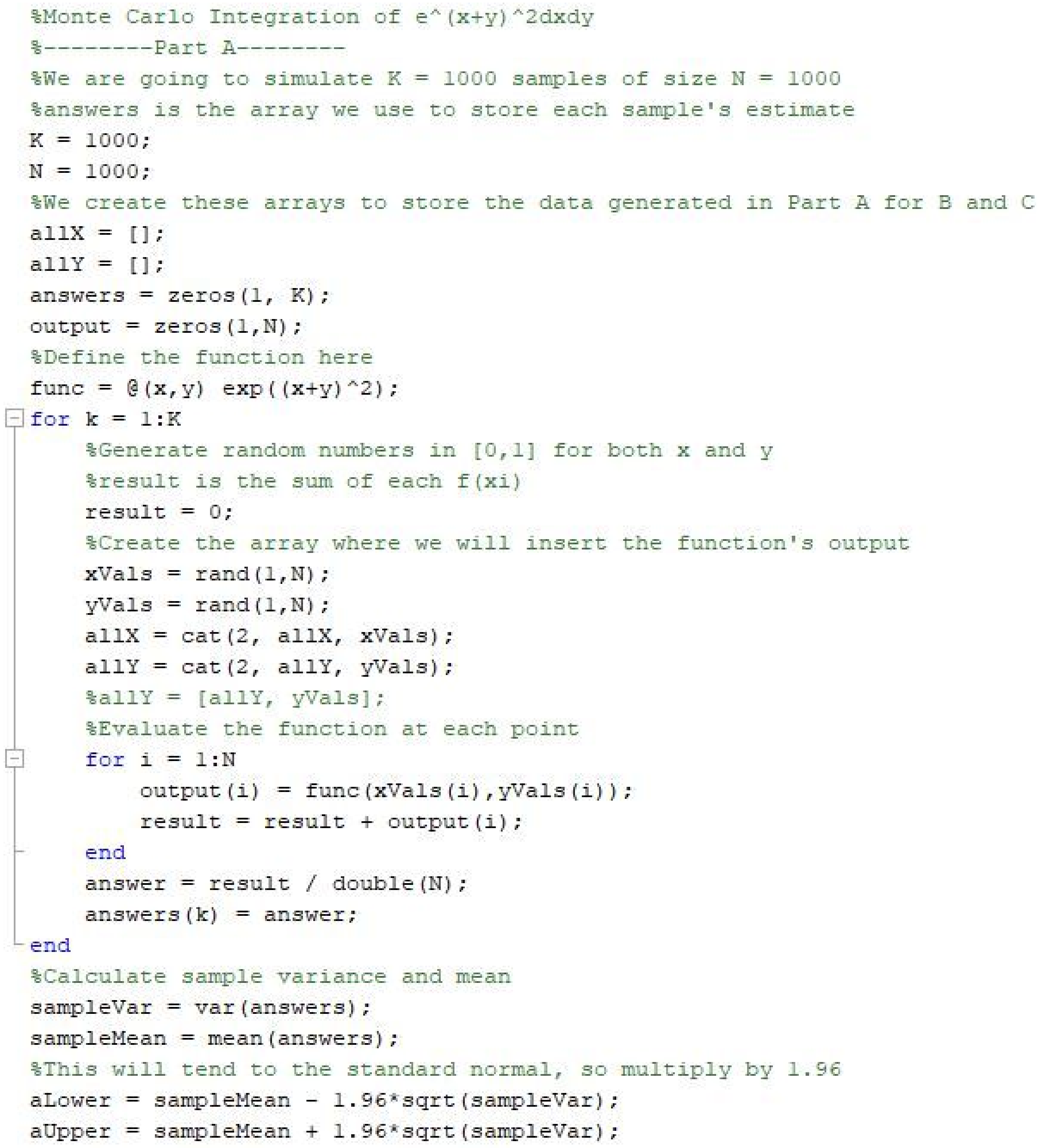
```matlab
% PART 6a
N_val = [10 10^2 10^3 10^4 10^5 10^6];
probs = zeros([1 6]); % initialize an array of probabilities
total = 0;
for j = 1:6
    N = N_val(j);

    % Generate an array of N random numbers from a standard normal
    % distribution
    r = normrnd(0, 1, [1, N])
    count = 0;
    % iterate through the random numbers and count how many are over 5
    for i = 1:N
        index = r(i);
        if index > 5
            count = count +1;
        end
    end
    % calculate percentage
    probs(j) = count/N
end
```

Part b

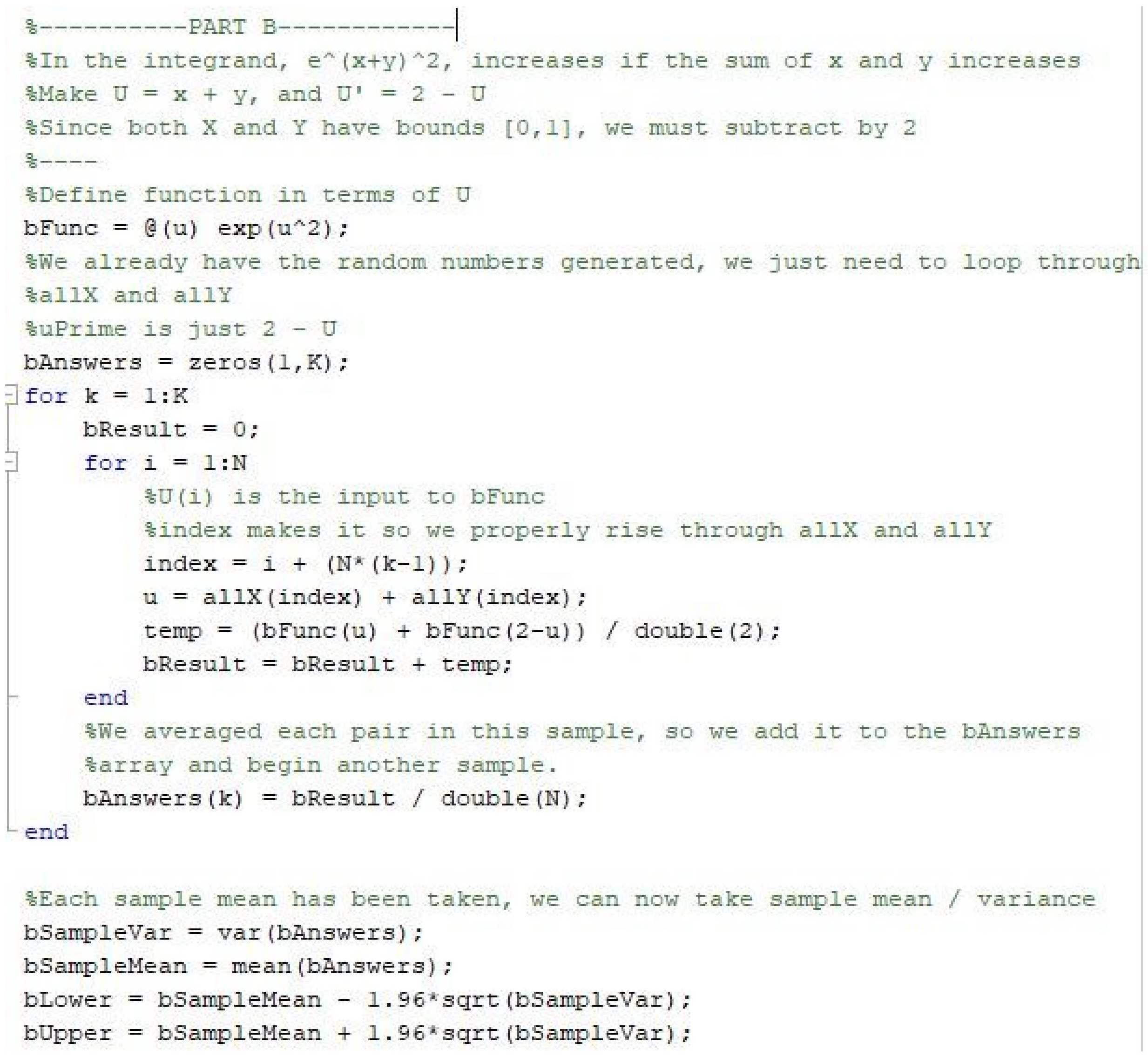


```
 Importance Sampling method

al = [10 10^2 10^3 10^4 10^5 10^6];
 j = 1:6
 N = N_val(j); % Iterate through different values of N
 r2 = normrnd(5.5, 0.5, [1, N]); % Generate from Y ~ N(5.5, 0.5) N t
 total = 0;
 for i = 1:N
     index = r2(i);
     % For each random number, calculate z
     z = identity(index) * f(index) / g(index);
     total = total + z;
 end
 % Average z over all N
 avg = total/N


FUNCTIONS
ction out = identity(y)
dentity function y > 5
 if y > 5
     out = 1;
 else
     out = 0;
 end


ction out = f(y)
tandard normal distribution
 out = exp(-y^2/2)/sqrt(2*pi);


ction out = g(y)
ormal distribution with (5.5, 0,5)
 out = exp(-(y-5.5)^2/(2*0.5))/sqrt(2*pi*0.5);
```

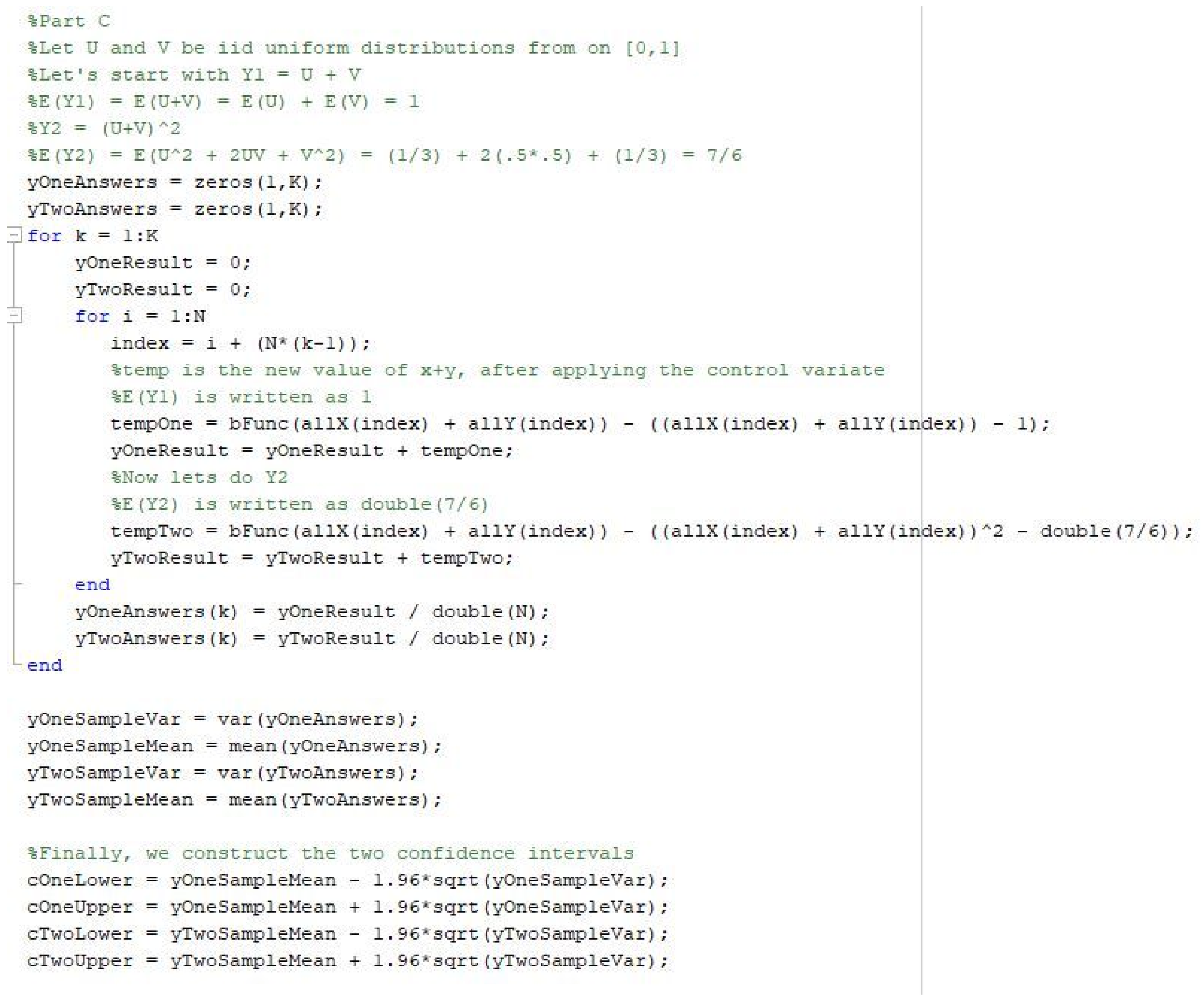
QUESTION 7:



```
%Monte Carlo Integration of e^(x+y)^2dxdy
%--------Part A--------
%We are going to simulate K = 1000 samples of size N = 1000
%answers is the array we use to store each sample's estimate
K = 1000;
N = 1000;
%We create these arrays to store the data generated in Part A for B and C
allX = [];
allY = [];
answers = zeros(1, K);
output = zeros(1,N);
%Define the function here
func = @(x,y) exp((x+y)^2);
for k = 1:K
    %Generate random numbers in [0,1] for both x and y
    %result is the sum of each f(xi)
    result = 0;
    %Create the array where we will insert the function's output
    xVals = rand(1,N);
    yVals = rand(1,N);
    allX = cat(2, allX, xVals);
    allY = cat(2, allY, yVals);
    %allY = [allY, yVals];
    %Evaluate the function at each point
    for i = 1:N
        output(i) = func(xVals(i),yVals(i));
        result = result + output(i);
    end
    answer = result / double(N);
    answers(k) = answer;
end
%Calculate sample variance and mean
sampleVar = var(answers);
sampleMean = mean(answers);
%This will tend to the standard normal, so multiply by 1.96
aLower = sampleMean - 1.96*sqrt(sampleVar);
aUpper = sampleMean + 1.96*sqrt(sampleVar);
```



```
%----------PART B------------
%In the integrand, e^(x+y)^2, increases if the sum of x and y increases
%Make U = x + y, and U' = 2 - U
%Since both X and Y have bounds [0,1], we must subtract by 2
%----
%Define function in terms of U
bFunc = @(u) exp(u^2);
%We already have the random numbers generated, we just need to loop through
%allX and allY
%uPrime is just 2 - U
bAnswers = zeros(1,K);
for k = 1:K
    bResult = 0;
    for i = 1:N
        %U(i) is the input to bFunc
        %index makes it so we properly rise through allX and allY
        index = i + (N*(k-1));
        u = allX(index) + allY(index);
        temp = (bFunc(u) + bFunc(2-u)) / double(2);
        bResult = bResult + temp;
    end
    %We averaged each pair in this sample, so we add it to the bAnswers
    %array and begin another sample.
    bAnswers(k) = bResult / double(N);
end

%Each sample mean has been taken, we can now take sample mean / variance
bSampleVar = var(bAnswers);
bSampleMean = mean(bAnswers);
bLower = bSampleMean - 1.96*sqrt(bSampleVar);
bUpper = bSampleMean + 1.96*sqrt(bSampleVar);
```

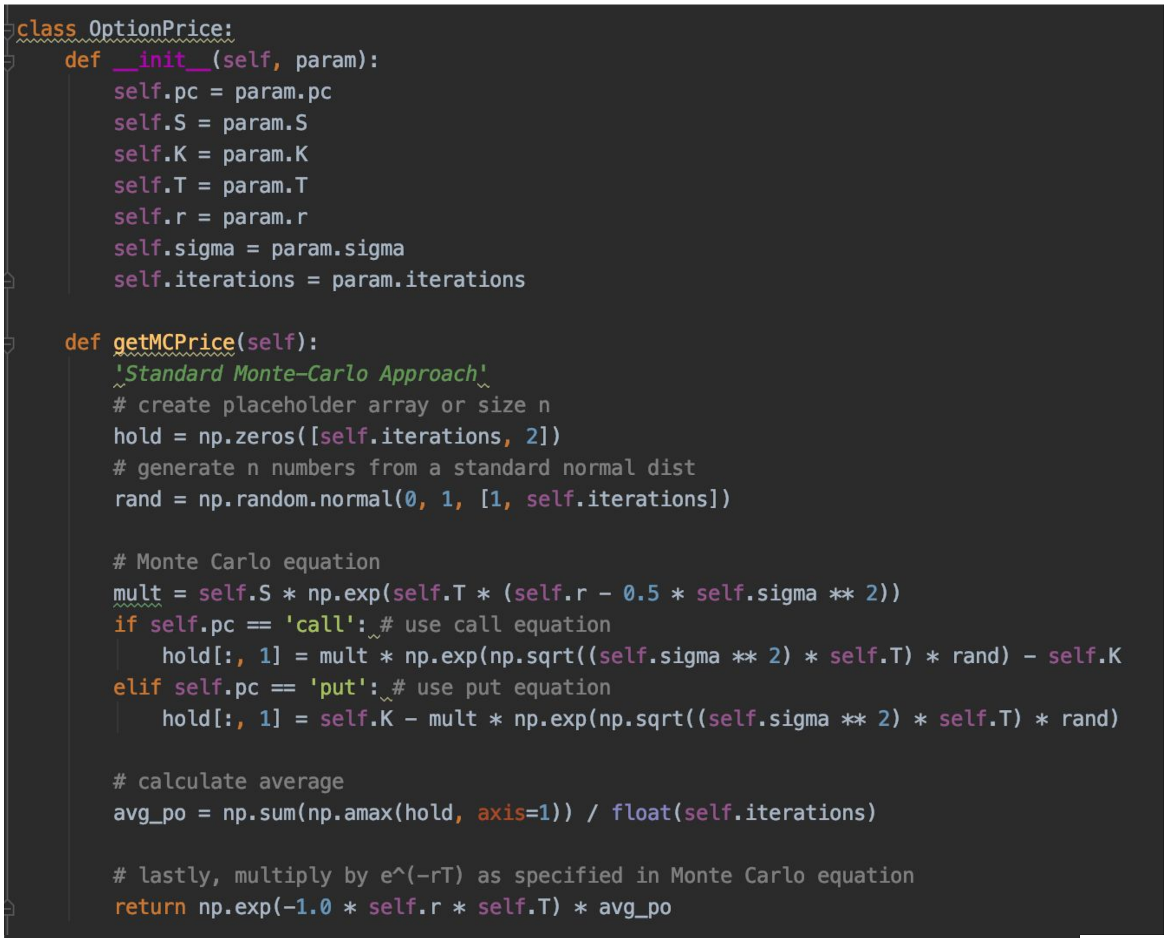


```
%Part C
%Let U and V be iid uniform distributions from on [0,1]
%Let's start with Y1 = U + V
%E(Y1) = E(U+V) = E(U) + E(V) = 1
%Y2 = (U+V)^2
%E(Y2) = E(U^2 + 2UV + V^2) = (1/3) + 2(.5*.5) + (1/3) = 7/6
yOneAnswers = zeros(1,K);
yTwoAnswers = zeros(1,K);
for k = 1:K
    yOneResult = 0;
    yTwoResult = 0;
    for i = 1:N
       index = i + (N*(k-1));
       %temp is the new value of x+y, after applying the control variate
       %E(Y1) is written as 1
       tempOne = bFunc(allX(index) + allY(index)) - ((allX(index) + allY(index)) - 1);
       yOneResult = yOneResult + tempOne;
       %Now lets do Y2
       %E(Y2) is written as double(7/6)
       tempTwo = bFunc(allX(index) + allY(index)) - ((allX(index) + allY(index))^2 - double(7/6));
       yTwoResult = yTwoResult + tempTwo;
    end
    yOneAnswers(k) = yOneResult / double(N);
    yTwoAnswers(k) = yTwoResult / double(N);
end

yOneSampleVar = var(yOneAnswers);
yOneSampleMean = mean(yOneAnswers);
yTwoSampleVar = var(yTwoAnswers);
yTwoSampleMean = mean(yTwoAnswers);

%Finally, we construct the two confidence intervals
cOneLower = yOneSampleMean - 1.96*sqrt(yOneSampleVar);
cOneUpper = yOneSampleMean + 1.96*sqrt(yOneSampleVar);
cTwoLower = yTwoSampleMean - 1.96*sqrt(yTwoSampleVar);
cTwoUpper = yTwoSampleMean + 1.96*sqrt(yTwoSampleVar);
```

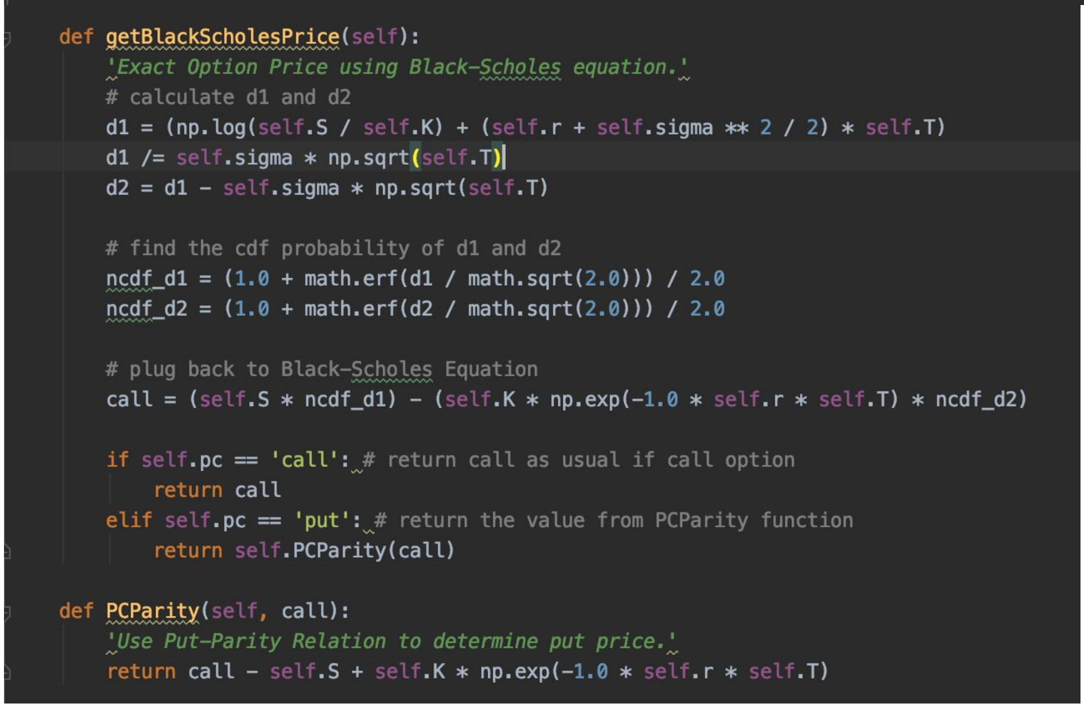
QUESTION 8 & 9

The following code has been produced in Python because an Object-Oriented language is beneficial for reproducibility. Additionally, this code includes the added functions/conditions to accommodate for the methods in question 9. We have made a class called *Option* to execute all our desired operations.

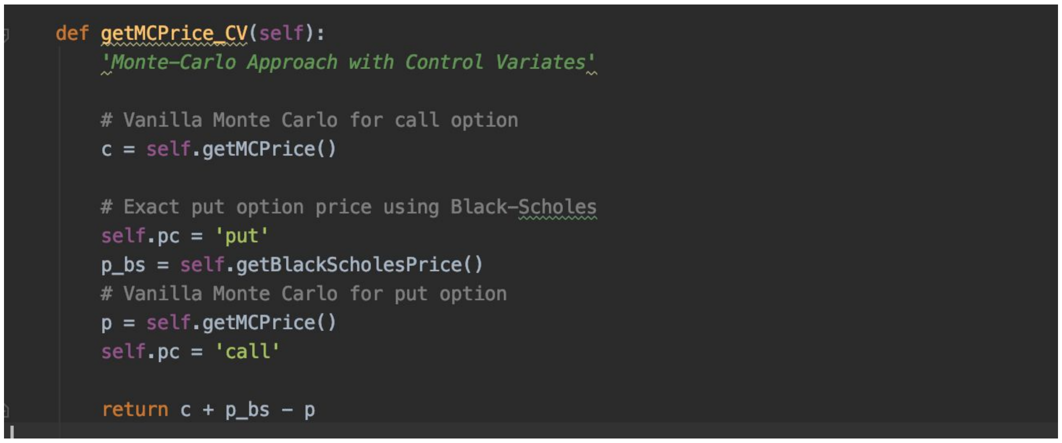
We have taken the code from this source, but have modified it significantly to accommodate to the questions of this report:
https://pawsdevelopment.wordpress.com/2016/11/22/monte-carlo-european-vanilla-option-pricing-with-python/



```python
class OptionPrice:
    def __init__(self, param):
        self.pc = param.pc
        self.S = param.S
        self.K = param.K
        self.T = param.T
        self.r = param.r
        self.sigma = param.sigma
        self.iterations = param.iterations

    def getMCPrice(self):
        'Standard Monte-Carlo Approach'
        # create placeholder array or size n
        hold = np.zeros([self.iterations, 2])
        # generate n numbers from a standard normal dist
        rand = np.random.normal(0, 1, [1, self.iterations])

        # Monte Carlo equation
        mult = self.S * np.exp(self.T * (self.r - 0.5 * self.sigma ** 2))
        if self.pc == 'call': # use call equation
            hold[:, 1] = mult * np.exp(np.sqrt((self.sigma ** 2) * self.T) * rand) - self.K
        elif self.pc == 'put': # use put equation
            hold[:, 1] = self.K - mult * np.exp(np.sqrt((self.sigma ** 2) * self.T) * rand)

        # calculate average
        avg_po = np.sum(np.amax(hold, axis=1)) / float(self.iterations)

        # lastly, multiply by e^(-rT) as specified in Monte Carlo equation
        return np.exp(-1.0 * self.r * self.T) * avg_po
```

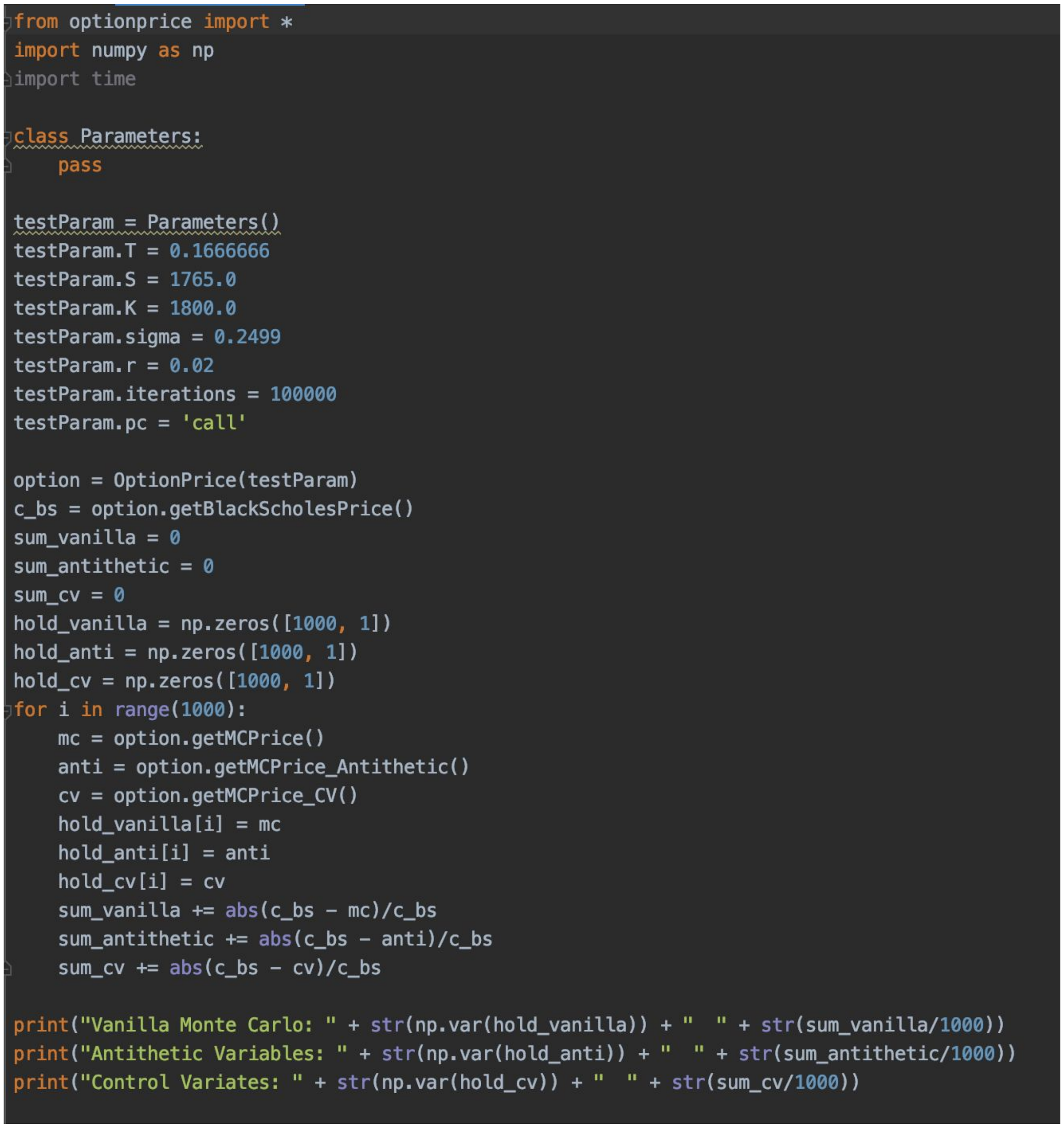


```python
    def getBlackScholesPrice(self):
        'Exact Option Price using Black-Scholes equation.'
        # calculate d1 and d2
        d1 = (np.log(self.S / self.K) + (self.r + self.sigma ** 2 / 2) * self.T)
        d1 /= self.sigma * np.sqrt(self.T)
        d2 = d1 - self.sigma * np.sqrt(self.T)

        # find the cdf probability of d1 and d2
        ncdf_d1 = (1.0 + math.erf(d1 / math.sqrt(2.0))) / 2.0
        ncdf_d2 = (1.0 + math.erf(d2 / math.sqrt(2.0))) / 2.0

        # plug back to Black-Scholes Equation
        call = (self.S * ncdf_d1) - (self.K * np.exp(-1.0 * self.r * self.T) * ncdf_d2)

        if self.pc == 'call': # return call as usual if call option
            return call
        elif self.pc == 'put': # return the value from PCParity function
            return self.PCParity(call)

    def PCParity(self, call):
        'Use Put-Parity Relation to determine put price.'
        return call - self.S + self.K * np.exp(-1.0 * self.r * self.T)
```



```python
    def getMCPrice_CV(self):
        'Monte-Carlo Approach with Control Variates'

        # Vanilla Monte Carlo for call option
        c = self.getMCPrice()

        # Exact put option price using Black-Scholes
        self.pc = 'put'
        p_bs = self.getBlackScholesPrice()
        # Vanilla Monte Carlo for put option
        p = self.getMCPrice()
        self.pc = 'call'

        return c + p_bs - p
```

Implementation of the Class



```python
from optionprice import *
import numpy as np
import time

class Parameters:
    pass

testParam = Parameters()
testParam.T = 0.1666666
testParam.S = 1765.0
testParam.K = 1800.0
testParam.sigma = 0.2499
testParam.r = 0.02
testParam.iterations = 100000
testParam.pc = 'call'

option = OptionPrice(testParam)
c_bs = option.getBlackScholesPrice()
sum_vanilla = 0
sum_antithetic = 0
sum_cv = 0
hold_vanilla = np.zeros([1000, 1])
hold_anti = np.zeros([1000, 1])
hold_cv = np.zeros([1000, 1])
for i in range(1000):
    mc = option.getMCPrice()
    anti = option.getMCPrice_Antithetic()
    cv = option.getMCPrice_CV()
    hold_vanilla[i] = mc
    hold_anti[i] = anti
    hold_cv[i] = cv
    sum_vanilla += abs(c_bs - mc)/c_bs
    sum_antithetic += abs(c_bs - anti)/c_bs
    sum_cv += abs(c_bs - cv)/c_bs

print("Vanilla Monte Carlo: " + str(np.var(hold_vanilla)) + "  " + str(sum_vanilla/1000))
print("Antithetic Variables: " + str(np.var(hold_anti)) + "  " + str(sum_antithetic/1000))
print("Control Variates: " + str(np.var(hold_cv)) + "  " + str(sum_cv/1000))
```

Works Cited

Lu, Bingqian. Monte Carlo Simulations and Option Pricing. 2011, Monte Carlo Simulations and Option Pricing, www.personal.psu.edu/alm24/undergrad/bingqianMonteCarlo.pdf.

Ødegaard, Bernt. “Option Pricing by Simulation.” Stavanger, University of Stavanger. http://finance.bi.no/~bernt/gcc_prog/recipes/recipes/node12.html.