STAT3011

GROUP 1 PRESENTATION

PENG ZHICHAO
LI JINZHAO
WONG KI YAN
YUEN CHUN WING
CHEUNG SIU FUNG
WONG CHUN FAN

ROOT FINDING

Bisection method

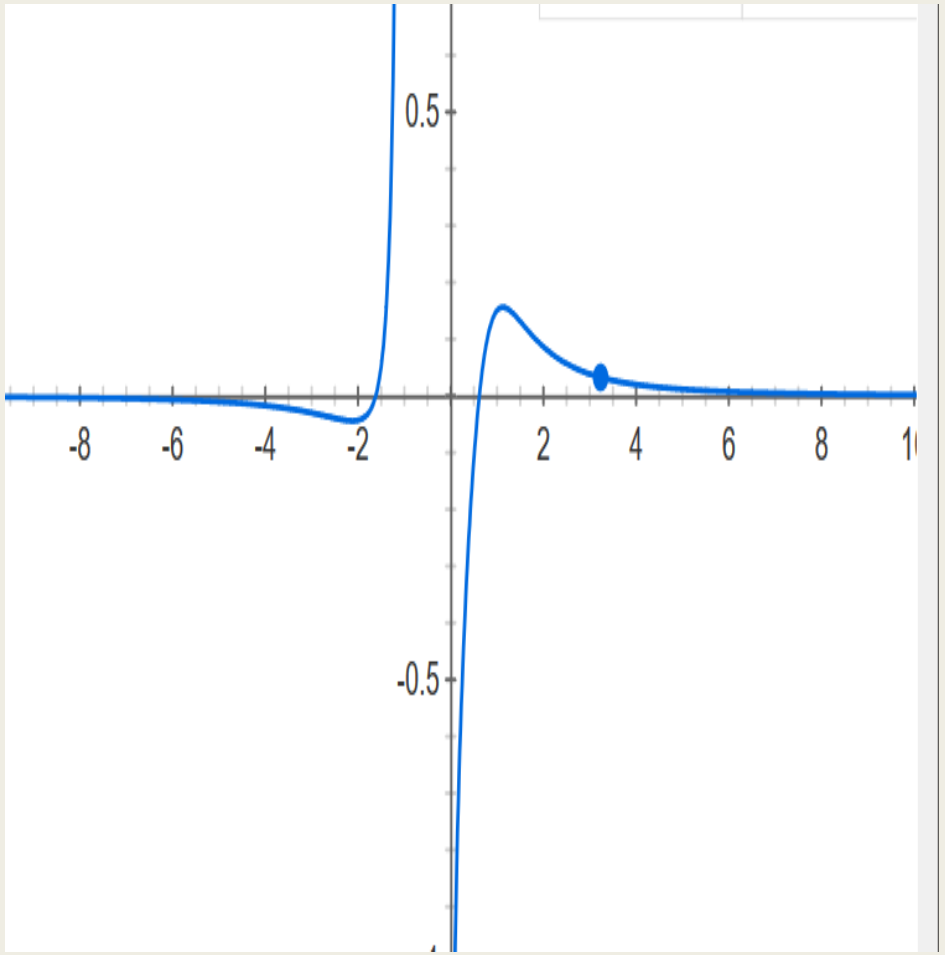
Bisection method

- Recall that f(x) is maximum if f’(x) = 0 and f’’(x) < 0
- We can apply bisection method to find the root(s) of the equation

Theory

Plot of f(x)

- We use the plot make an educated guess for a and b
- We choose a = 0.5 and b = 1.5



Result

Newton’s method

- Alternate method to find roots of f’(x) = 0
- More advanced but effective method

Theory

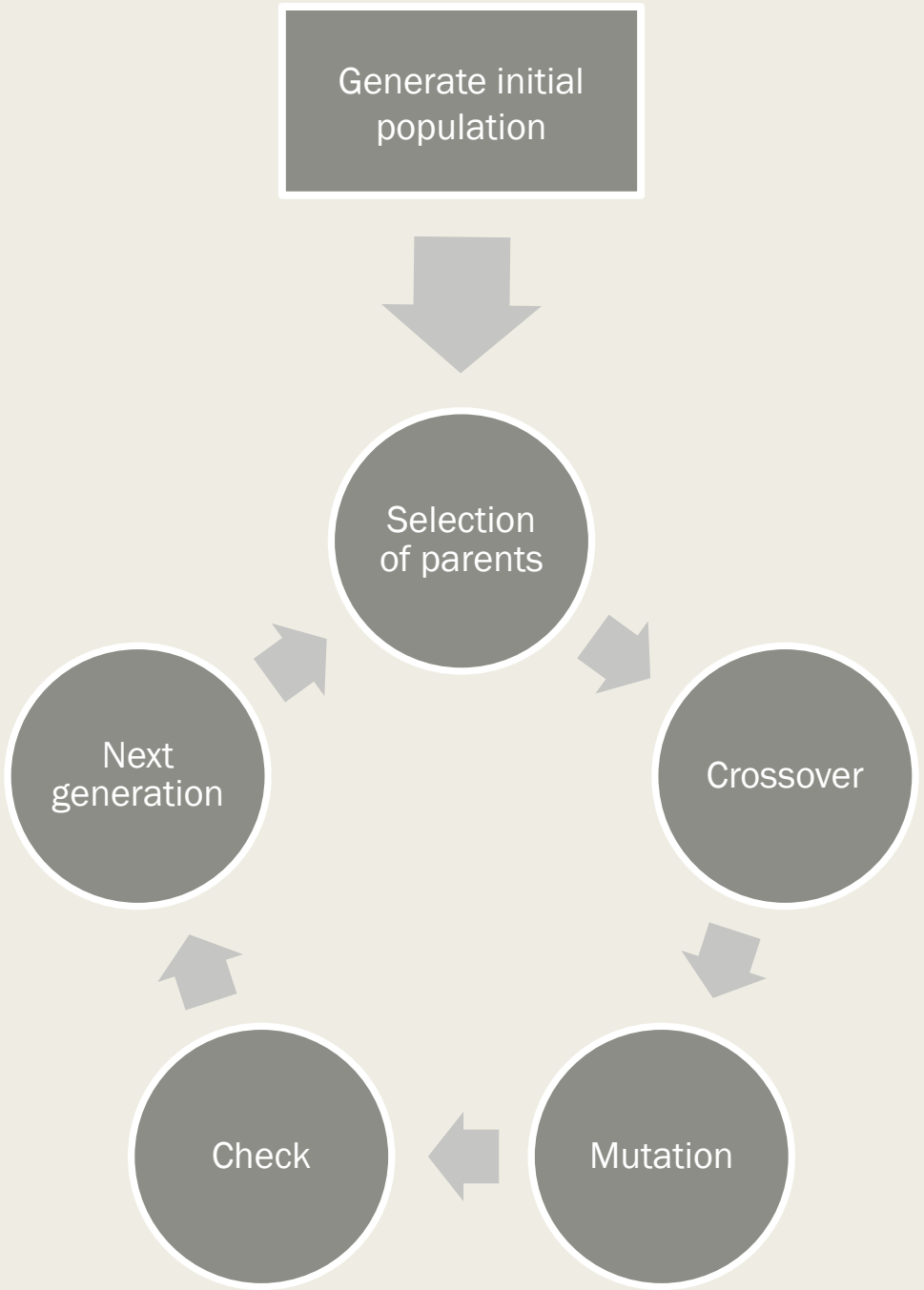
Result

Comparison

Comparison

- The problem with newton’s method is that finding f’’(x) may be complicated and time-consuming
- Bisection method converges relatively slow
- Bisection method is easy to implement

GENETIC ALGORITHM


Generate initial population
Selection of parents
Crossover
Mutation
Check
Next generation

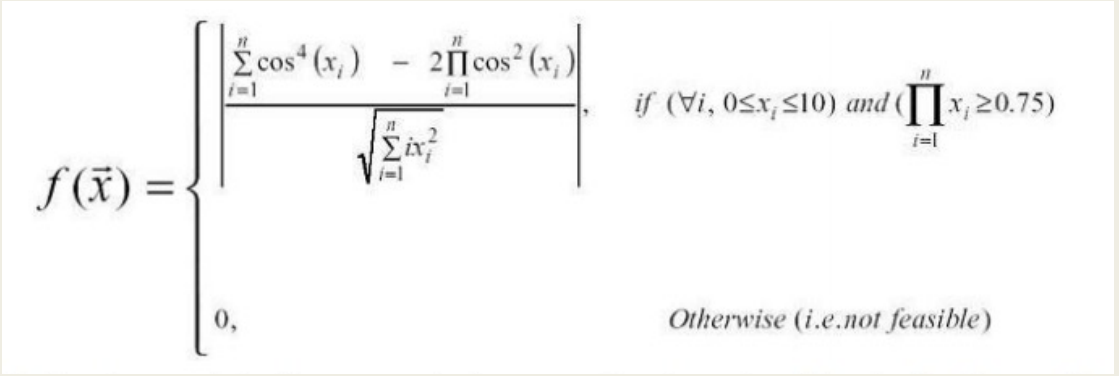
Generate initial population

- **decimal** : generate a series of 14 binary numbers
- **gfunction** : convert the binary numbers into decimal numbers with 4 significant figures which range between 0 to 10
- **ppcode** : generate a complete set of X1 to X10 as the initial population, i.e. product of X1 to X10 is larger or equals to 0.75

- Population size: n = 21

Selection of parents

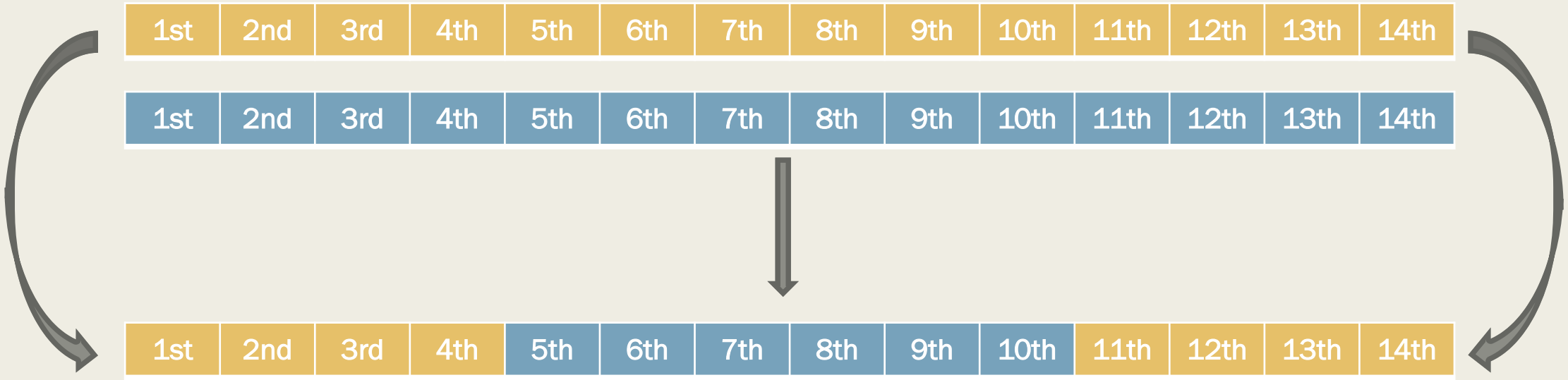
- **fitness** : calculate the value by substituting X1 to X10 into formula f(X)



- **rangroup** : randomly divide the population into 7 groups and select the maximum within the subgroup
- **rangroup6** : repeat the rangroup function for 6 times in order to get 21 pairs of parents

Crossover

- **Nextgen** : keep the binary digits of chromosome A (yellow) and replace the central part (the 5^{th} to 10^{th} digits) by chromosome B (blue)
- **Nextgen3** : select the parents by probability that higher the fitness, higher chance to be selected.

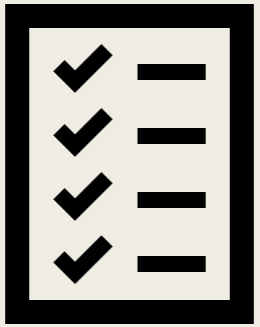


Mutation

- **Mutation** : change on the binary digits, either from 0 to 1, or from 1 to 0, this leads to the change of the value of X1 to X10
- mutation rate = 0.02

Check

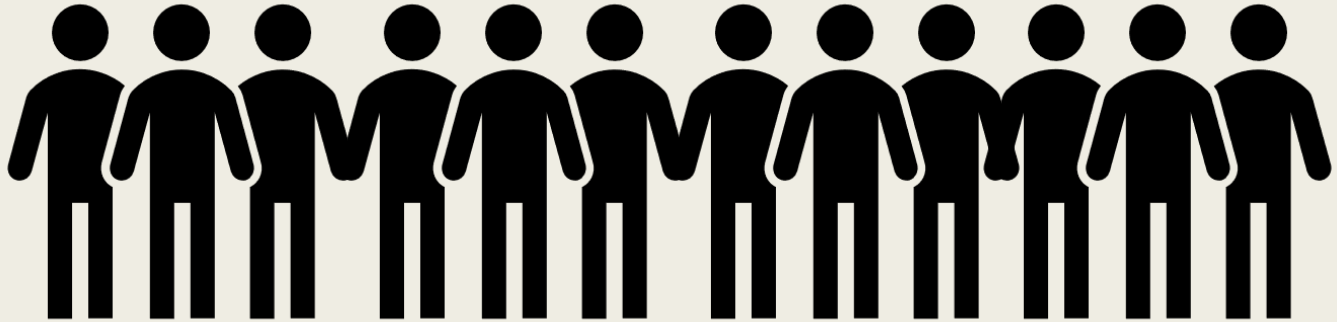
- **check** : ensure there is no missing value within the next generation
- **finstep** : check product of X1 to X10 of the next generation is larger than or equals to 0.75 or not; if yes, the next generation is generated, if not, a new individual will be produced by function **Nextgen3**



Next generation

- **GA :** generate n generations by repeating the abovementioned steps to obtain the maximum

- **Result :** return the maximum value of f(x) and the corresponding X1 to X10 after having number of iterations

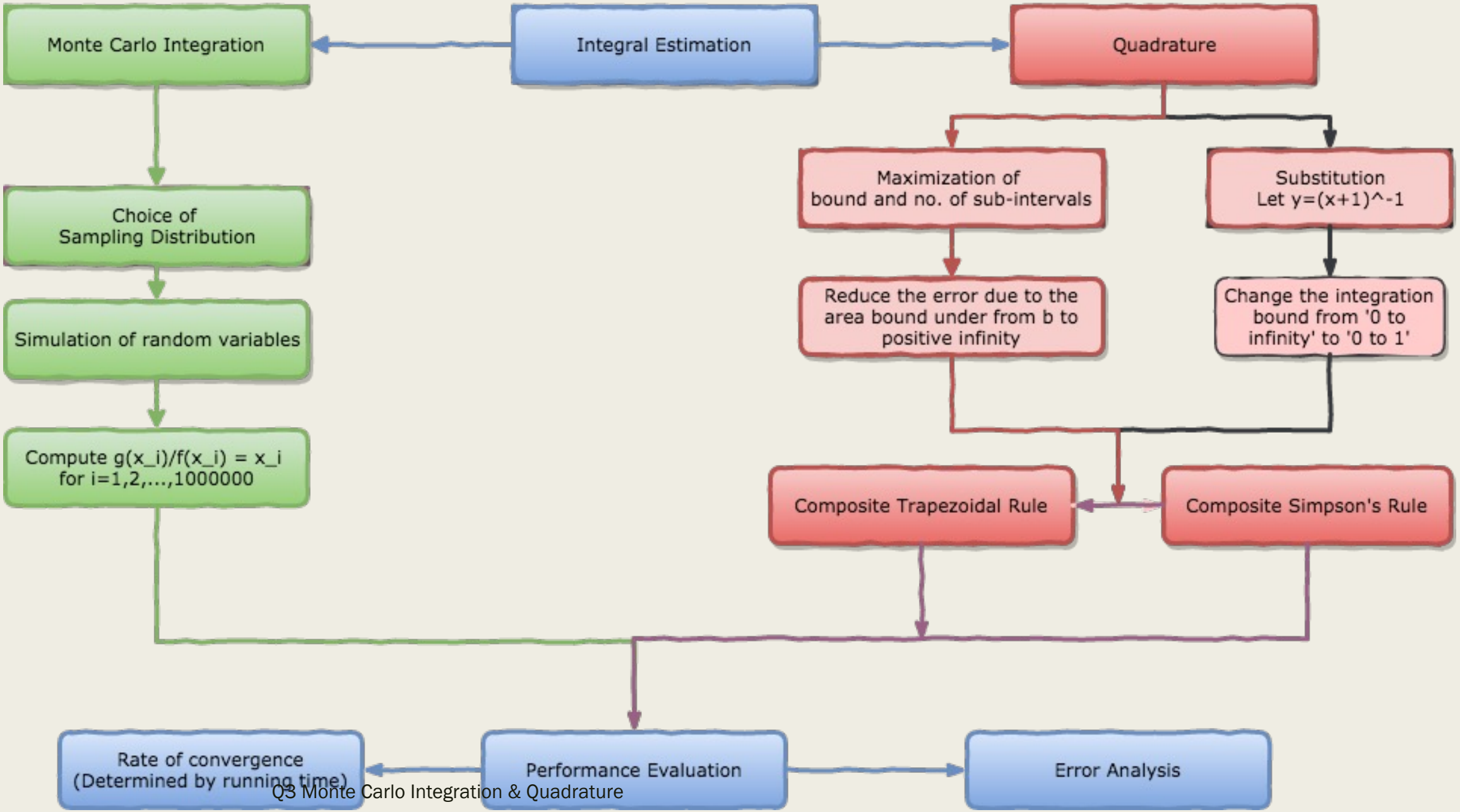


Result

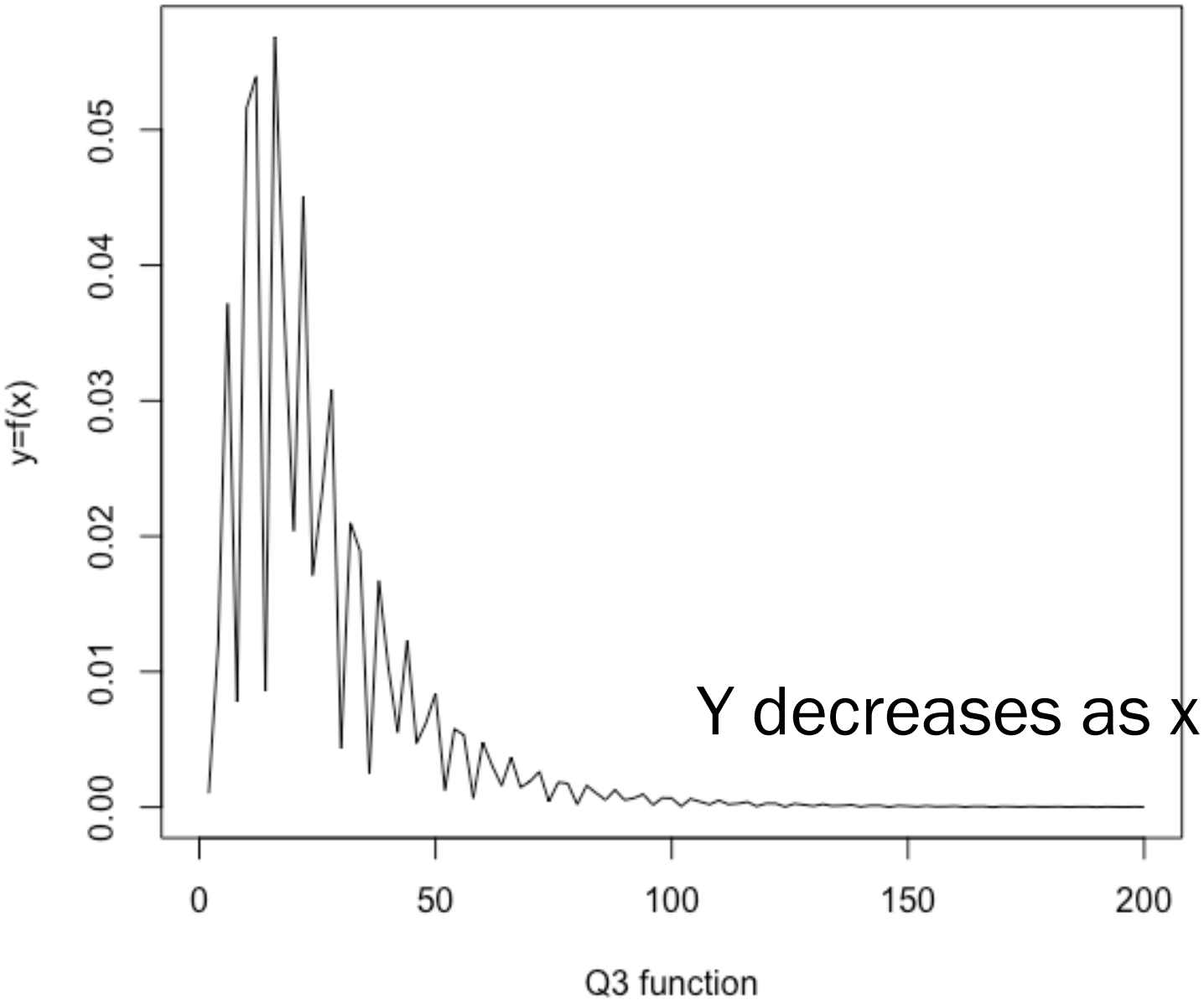
f(x) = 0.7441926	
3.1636	0.3131
3.1227	0.3577
3.0025	0.4010
2.9128	0.3516
1.4649	0.3772

INTEGRAL ESTIMATION

Flow Chart of Question 3



Graph Illustration of the function



Y decreases as x tends to infinity

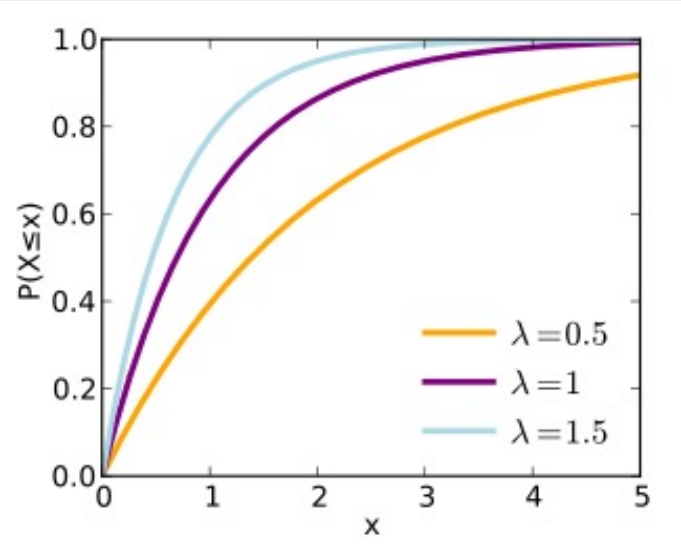
Monte Carlo Integration

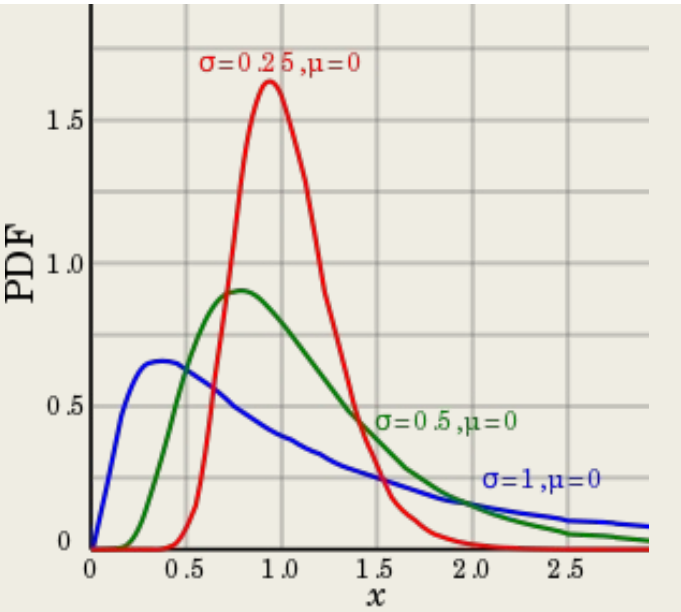
Big Idea Behind is the Law of Large Numbers which states that

Monte Carlo Integration

Procedure of Monte Carlo Integration

Error Analysis of Monte Carlo Integration


1.0
0.8
0.6
0.4
0.2
0.0
P(X≤x)
0
1
2
3
4
5
x
λ=0.5
λ=1
λ=1.5


σ=0.25,μ=0
1.5
1.0
0.5
0
PDF
σ=0.5,μ=0
σ=1,μ=0
0
0.5
1.0
1.5
2.0
2.5
x

Transformation of the formula

R code – Exponential Distribution (1)

```
mcie <- function(n,f){          #Set up a function of simulation
  U <- runif(n)    # Simulate n random numbers
  X <- -log((1-U))  #Simulate n values from Exp(1)
  Int <- c(mean(f(X)), var(f(X))/n) #Compute g(x_i)/f(x_i)=x_1 for i = 1 to n and the error estimation
  Int} #Calculate the sample mean to estimate the population mean
set.seed(77960)
mcie(5000000,f3a)
```

R code – Lognormal Distribution (3,1)

```
mciln <- function(n,f){          #Set up a function of simulation
  X <- rlnorm(n, meanlog=3, sdlog=1)  #Simulate n values from Lognormal(0,1)
  Int <- c(mean(f(X)), var(f(X))/n) #Compute g(x_i)/f(x_i)=x_1 for i = 1 to n and the error estimation
  Int} #Calculate the sample mean to estimate the population mean
set.seed(77960)
mciln(5000000,f3b)
```

R code

```
#Calculate the running time
runtime=function(fn){
  start_time=Sys.time()
  fn
  end_time=Sys.time()
  return(start_time-end_time)
}
runtime(mcie(100000,f3))
runtime(mciln(100000,f3))
```

Results

- > set.seed(77960)
- > mciln(5000000,f3b)
- [1] 1.128507e+00 1.079433e-07

- > set.seed(77960)
- > mcie(5000000,f3a)
- [1] 0.9318097 0.3014599

- > runtime(mcie(100000,f3))
- Time difference of -0.118232 secs
- > runtime(mciln(100000,f3))
- Time difference of -0.03117204 secs

Results

	X~Exp(1)	X~Lognormal(3,1)
Integral	0.9318097	1.128507
Error Estimate	0.3014	1.079433*10^(-7)
Running Time(seconds)	0.118232	0.03117204

Interesting Finding - Monte Carlo Integration

Both running time and error sampling from exponential distribution is much higher than lognormal distribution.

Error estimation decrease with N -> the Error is inversely proportional to the sample size.

How Did Composite Trapezoidal Rule Work?

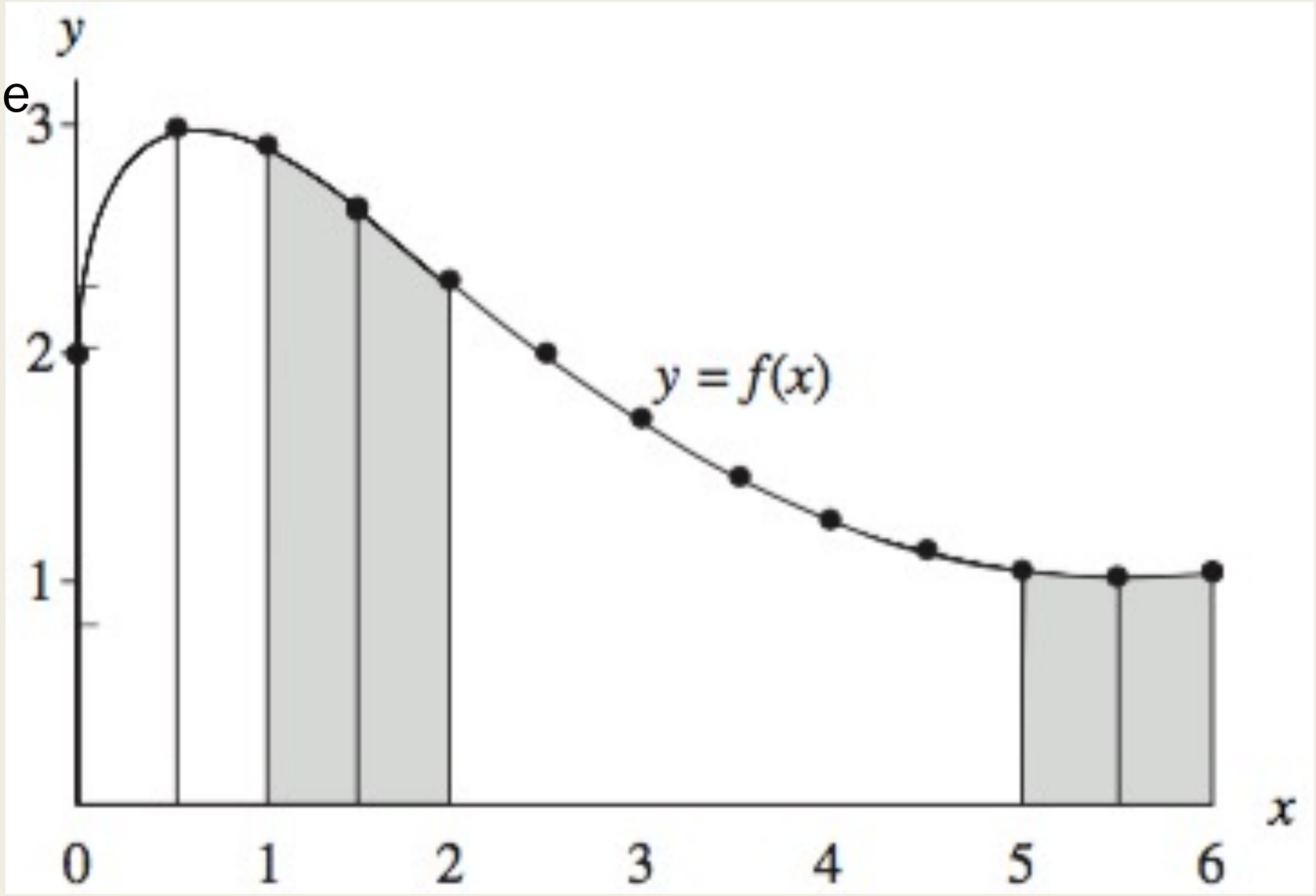
A Simple Example

b=6

a=0

h=0.5

n=12



How Did Composite Trapezoidal Rule Work?

How Did Composite Simpson’s Rule Work?

- Trapezoidal Rule: Approximate the integral by subdividing the area bound by a series of trapezoids.
- Simpson’s Rule attempted to improve the accuracy by using quadratic polynomials, which means that using the parabolic arcs instead of straight line segments.

- Here, we use the same notation for the number of subintervals, length of each subinterval, lower bound and upper bound.

Clear Difference between the rules

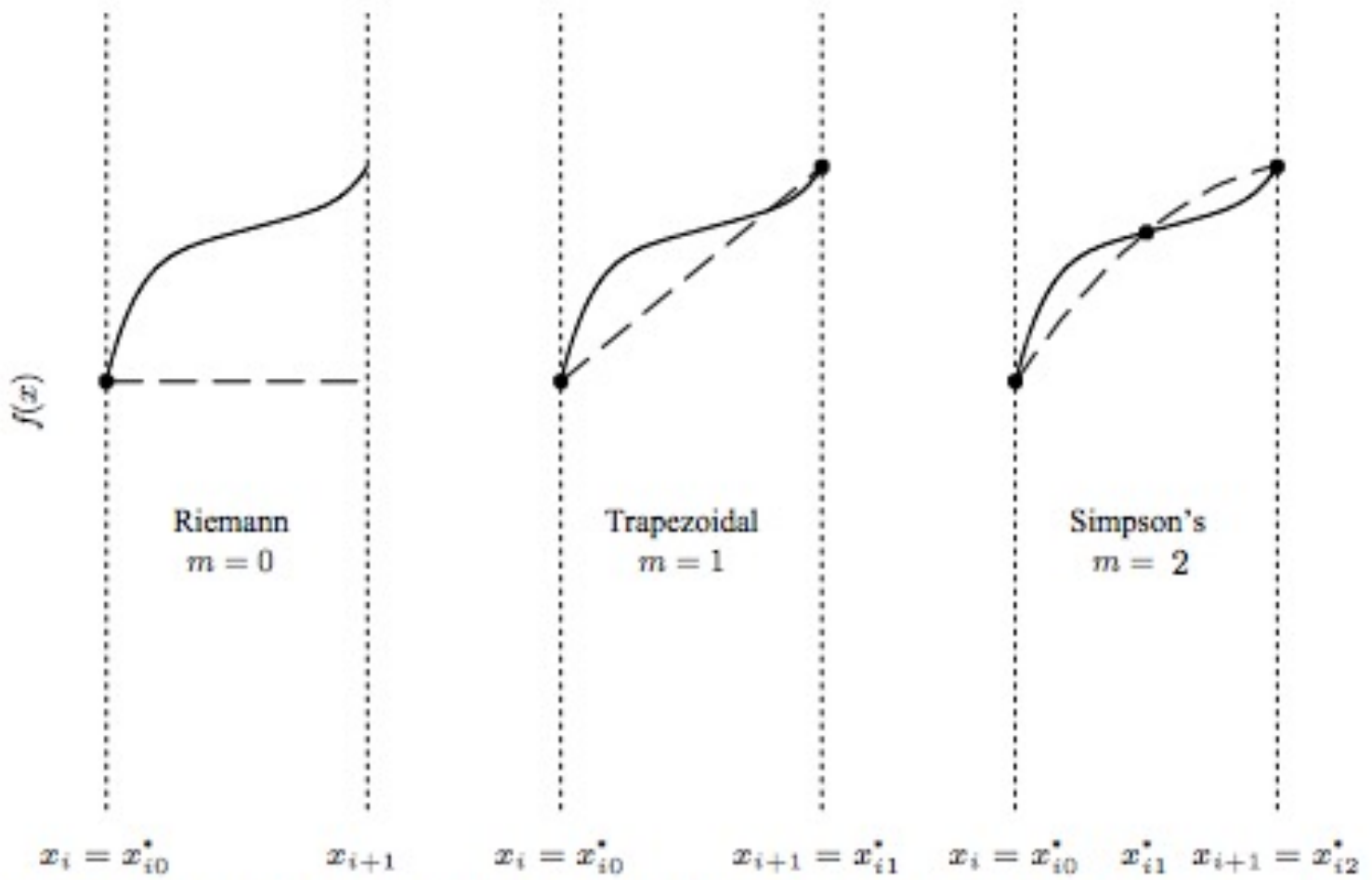


FIGURE 5.2 Approximation (dashed) to f (solid) provided on the subinterval $[x_i, x_{i+1}]$, for the Riemann, trapezoidal, and Simpson's rules.

Generalized formula of Composite Simpson’s Rule

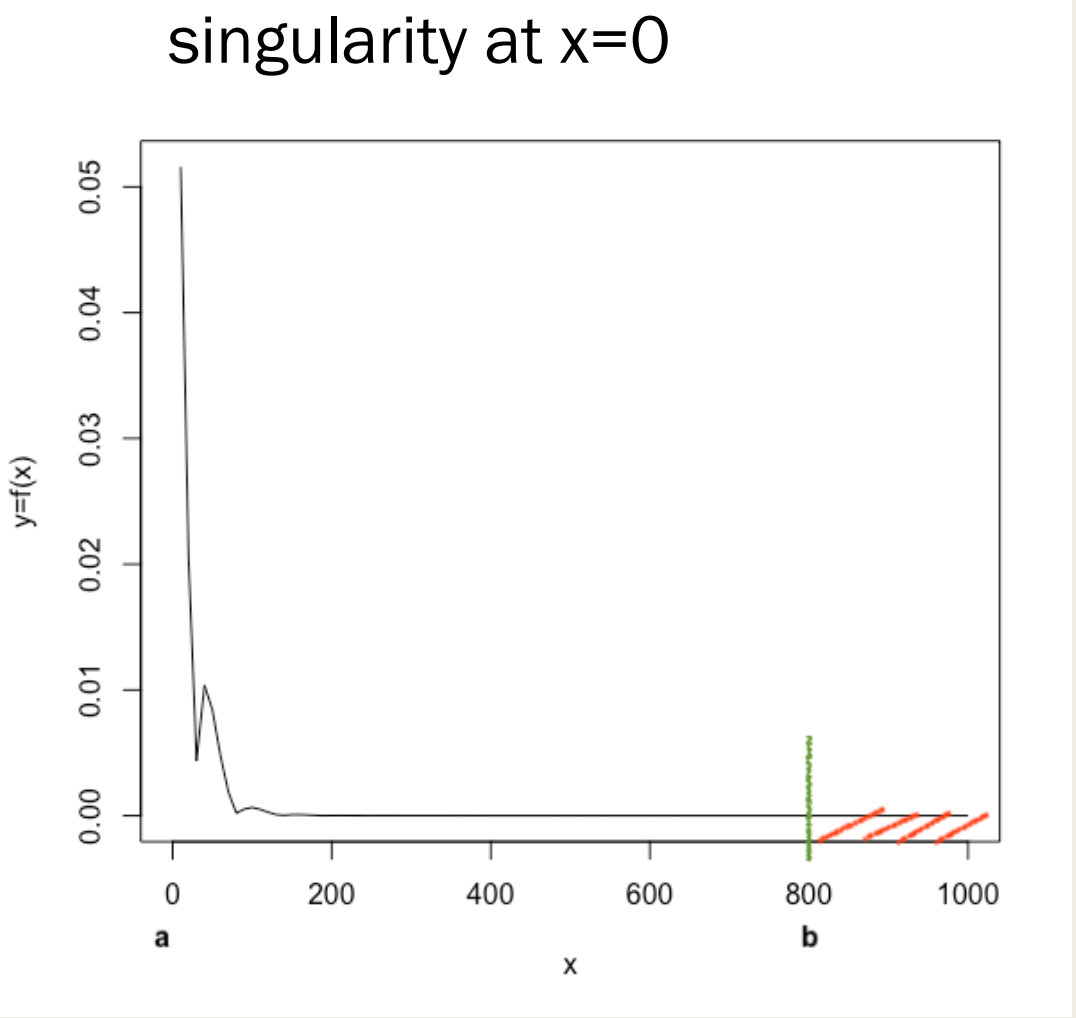
Q2 Genetic Algorithm

Potential Problem on Q3

R will get in the loop and fail to response if we set the upper bound to positive infinity.

The estimation may work worse on infinity bound since part of the area bound is not calculated as the figure illustrated.

The integral has singularity at x=0



Our Approaches

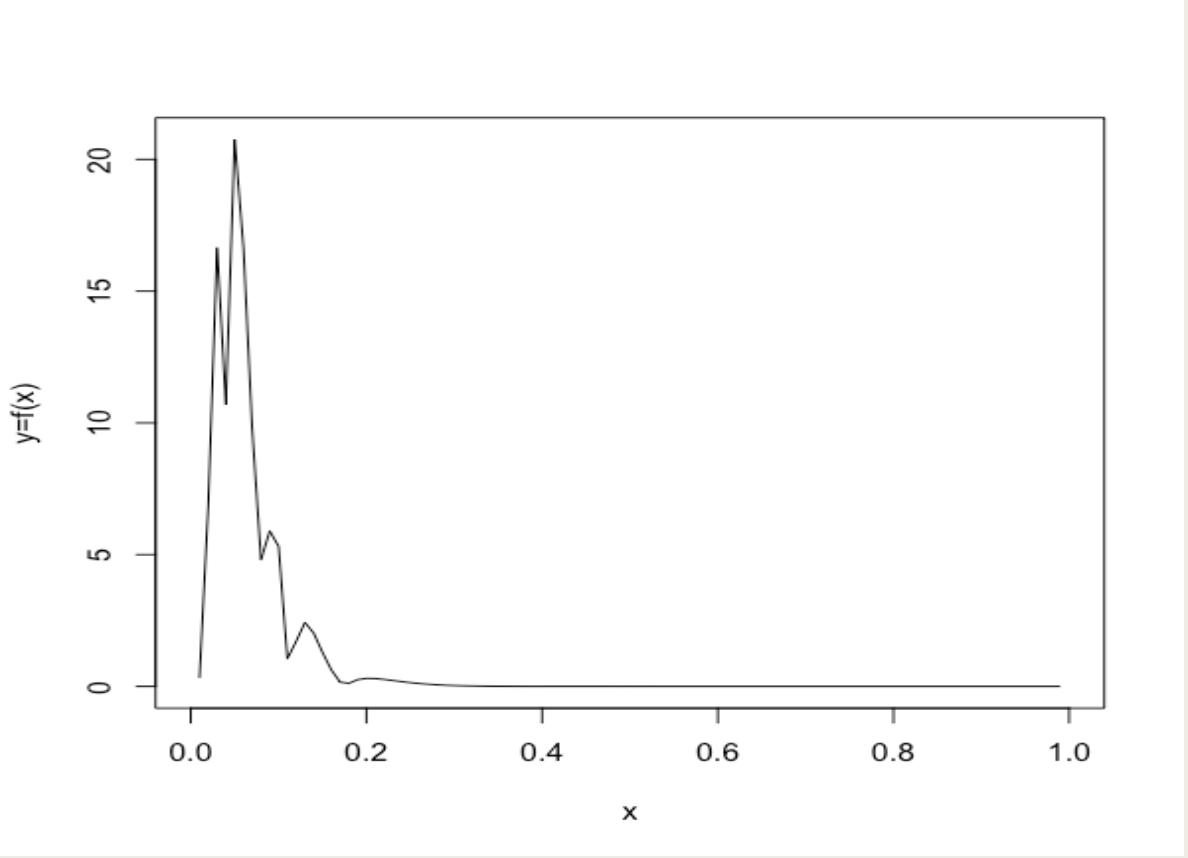
- 1. Compromise part of the accuracy, maximize the bound according to the performance of the computer.
  The error can be negligible due to the tendency to 0 as x increases.
- Here we decide to calculate the integral up to 10000000 (i.e. b=1e7), subdividing the whole interval into 100000 pieces (i.e. n=1e5)
- 2. Transform the curve in order to change to a calculable interval by substitution.
- Here we choose to substitute u equaling to 1/(x+1).

Our Approaches

- 2. Transform the curve in order to change to a calculable interval by substitution.
  Here we choose to substitute u equaling to 1/(x+1).

- 3. Add an extremely small adjustment value to the initial value x.
  A better approach mentioned in the reference book is transformation.

Details of Substitution



R Codes for the integral estimation - Substitution

```
#By substitution
#Define the function
fy=function(y){
  result=(1/(y*(1-y)))*abs(cos((1-y)/y))*exp(-(log((1-y)/y)-3)^2)
  return(result)
}
curve(fy,0,1, ylab = "y=f(x)") #sketch the curve in 0 to 1
```

R Codes for the integral estimation - Substitution

```
#1.Composite trapezoidal rule
CTrape<-function(funct,a,b,n){ #Define the parameters #n is the no. of subinterval
  h <- (b - a) / n #Calculate the increment
  c <- 1:(n - 1) #Create a vector for the order of the increment
  xj <- a + c * h #Transform the order vector to a vector of f(x_j)
  Integral <- (1/2*h) * (2 * sum(funct(xj)))
  #Composite Trapezoidal Rule= h/2+[f(a)+Summation_c=1_n-1(f(x_j)+f(b))]
```

R Codes for the integral estimation - Substitution

```
#Adjustment on the a for getting error on a = 0
value<-function (funct, x, adj=h/1e6) { #Set up an adjustment value
#Compromise on a small error to avoid the infinity value on 0
    if (is.nan(funct(x))) { #if condition when f(x)=NaN
      if (x==0) return (funct(adj+x)) #add adjustment value to x_1 when x=0
      else return (funct(adj*(-1)+x)) #prevent any NaN cases of x_j!=0
    }
    else {return(funct(x))}
  }
```

R Codes for the integral estimation - Substitution

```
 Integral=Integral+(h/2*(value(funct,a)+value(funct,b))) #Calculate the final Integral
  (Result= Integral)
}
set.seed(77960)
CTrape(fy,0,1,2000)
```

R Codes for the integral estimation - Substitution

```
#2. Composite Simpson's rule
CSimp<-function(funct,a,b,n) { #Define the parameters #n is the no. of subinterval
  h <- (b - a) / n #Calculate the increment
  c<-0:(n-1)
  d <- 1:(n - 1) #Create a vector for the order of the increment
  xj1 <- a+(c+1/2)*h  #Transform the order vector to a vector of f(x_j)
  xj2<- a+(d*h)
  Integral <- (1/6*h) * (4 * sum(funct(xj1))+2*sum(funct(xj2)))
#Calculate parts of the formula of Simpson's rule
```

R Codes for the integral estimation - Substitution

```
#Adjustment on the a for getting error on a = 0
value <- function (funct, x, adj=h/1000000) {
#Set up an adjustment value  #Compromise on a small error to avoid the infinity value on 0
    if (is.nan(funct(x))) {
      if (x==0) return (funct(adj+x))
      else return (funct(adj*(-1)+x))
    }
    else {return(funct(x))}
  }
```

R Codes for the integral estimation - Substitution

```
 Integrand=Integral+(h/6*(value(funct,a)+value(funct,b)))

  #Composite Simpson's Rule= h/6*[f(a)+4*Summation_i=0_n-
1(f(a+(i+1/2)*h))+2*Summation_i=1_n-1(f(a+i*h)+f(b)]

  (Result= Integral) #return the result

}

set.seed(77960)

CSimp(fy,0,1,2000)
```

Matlab code for the error estimation - Substitution

```
syms x %create symbolic variable x
tic %start stop watch timer
equation=(abs(cos((1-x)/x))/x/(1-x))*exp(-(log((1/x)/x)-3)^2);
d4equation=diff(equation,4); %obtain the fourth derivative of the equation
a=0;b=0; %initialize the variables
for n=0.0005:0.25:1 %define the interval and the increment
    a=a+subs(d4equation,x,n); %for loop for the sum of the derivatives at points
    b=b+1; %order increases by 1 after sum of a
end
average1=a/b; %calculate the mean of the derivatives
cserr=average1.*(1/2880).*(1e7-1/1e7).^5./1e5.^4 %Error estimation of CsimpsonRule
toc %stop the stop watch timer and show the elapsed time
```

Matlab code for the error estimation - Substitution

```
tic %start stop watch timer
equation=(abs(cos((1-x)/x))/x/(1-x))*exp(-(log((1/x)/x)-3)^2);
d4equation2=diff(equation2,2); %obtain the second derivative of the equation
c=0;d=0; %initialize the variables
for n=0.0005:0.25:1 %define the interval and the increment
    c=c+subs(d4equation2,x,n); %for loop for the sum of the derivatives at points
    d=d+1; %order increases by 1 after sum of c
end
average2=c/d; %calculate the mean of the derivatives
cterr=average2.*(-1/12).*(1e7-1/1e7).^3./1e5.^2 %Error estimation of
CTrapezoidalRule
```

Matlab code for the error estimation - Substitution

tic %start the stop watch timer

cserr-cterr %Calculate the difference between the two rules

sign(cserr-cterr) %show which rule has higher error

toc %obtain the elapsed time

R Codes for the integral estimation - Substitution

```
#Calculate the running time for the integral estimation by two rules
runtime=function(fn){
  start_time=Sys.time()
  fn
  end_time=Sys.time()
  return(-1*(end_time-start_time))
}
runtime(CTrape(fy,0.0005,1,2000))
runtime(CSimp(fy,0.0005,1,2000))
```

Results - Substitution

	CTrape	CSimp
Integral	1.123706	1.134448
Sign of the difference of the errors[Csimp-Ctrape]	Positive (i.e. $Error_{Simposon's} > Error_{Trapezoidal}$)	
Running Time(seconds) **[Integral Estimation]**	0.0001599789	0.0002520084
Running Time(seconds) **[Error Estimation]**	3.226210	7.16087
Running Time(seconds) [Sign of difference]	4.107797	

R Codes for the integral estimation - Maximization

```
#By maximization Define the function and draw the curve

f3 <- function(x) {return((abs(cos(x))/x)*exp(-(log(x)-3)^2)) } #function of question 3

f3d2<- function(x) {return(x^5*exp(-1*(log(x)^2)-9)*(2*sin(x)*dist.delta(cos(x))-
abs(cos(x)))

                    +2*x^3*exp(-1*(log(x)^2)-9)*abs(cos(x))*(2*(log(x)^2)-
9*log(x)+x*(2*log(x)-5)*tan(x)+9))} #second derivative of the function

curve(f3d2, 0,1000 , ylab = "y=f(x)") #sketch the curve

curve(f3, 0, 50 , ylab = "y=f(x)") #sketch the curve
```

R Codes for the integral estimation - Maximization

```
value <- function (funct, x, adj=h/1000000) { #Set up an adjustment value
#Compromise on a small error to avoid the infinity value on 0
    if (is.nan(funct(x))) { #if condition when f(x)=NaN
      if (x==0) return (funct(adj+x)) #add adjustment value to x_1 when x=0
      else return (func(adj*(-1)+x)) #prevent any NaN cases of x_j!=0
    }
    else {return(funct(x))}
  }
```

R Codes for the integral estimation - Maximization

```
Integrand=Integrand+(h/2*(value(funct,a)+value(funct,b))) #Calculate the final integrand


  (Result= Integrand)
}
set.seed(77960)
CTrape(f3, 0, 100000, 10000000)
```

R Codes for the integral estimation - Maximization

```
#2. Composite Simpson's rule

CSimp<-function(funct,a,b,n) { #Define the parameters #n is the no. of subinterval

CSimp<-function(funct,a,b,n) { #Define the parameters #n is the no. of subinterval

  h <- (b - a) / n #Calculate the increment

  c<-0:(n-1)

  d <- 1:(n - 1) #Create a vector for the order of the increment

  xj1 <- a+(c+1/2)*h  #Transform the order vector to a vector of f(x_j)

  xj2<- a+(d*h)

  Intergrand <- (1/6*h) * (4 * sum(funct(xj1))+2*sum(funct(xj2)))
```

R Codes for the integral estimation - Maximization

```
#Calculate parts of the formula of Simpson's rule

value <- function (funct, x, adj=h/1000000) { #Set up an adjustment valueCompromise on a small error to avoid the infinity value on 0

    if (is.nan(funct(x))) {

      if (x==0) return (funct(adj+x))

      else return (func(adj*(-1)+x))

    }

    else {return(funct(x))}

  }
```

R Codes for the integral estimation - Maximization

```
  Integrand=Integrand+(h/6*(value(funct,a)+value(funct,b)))

  #Composite Simpson's Rule= h/6*[f(a)+4*Summation_i=0_n-
1(f(a+(i+1/2)*h))+2*Summation_i=1_n-1(f(a+i*h)+f(b)]

  return(Integrand)

}

set.seed(77960)

CSimp(f3, 0, 100000, 1000000)
```

Matlab code for the error estimation - Maximization

```
syms x %create symbolic variable x

tic %start stop watch timer

equation=(abs(cos(x))/x)*exp(-(log(x)-3)^2); %define the equation

d4equation=diff(equation,4); %obtain the fourth derivative of the
equation

a=0;b=0; %initialize the variables

for n=1/1e7:5000:1e7 %define the interval and the increment

    a=a+subs(d4equation,x,n); %for loop for the sum of the derivatives at
points

    b=b+1; %order increases by 1 after sum of a

end
```

Matlab code for the error estimation - Maximization

```
average1=a/b; %calculate the mean of the derivatives

cserr=average1.*(1/2880).*(1e7-1/1e7).^5./1e5.^4 %Error estimation of CsimpsonRule

toc %stop the stop watch timer and show the elapsed time


tic %start stop watch timer

equation2=(abs(cos(x))/x)*exp(-(log(x)-3)^2); %define the equation

d4equation2=diff(equation2,2); %obtain the second derivative of the equation

c=0;d=0; %initialize the variables

for n=1/1e7:5000:1e7 %define the interval and the increment

    c=c+subs(d4equation2,x,n); %for loop for the sum of the derivatives at points
```

Matlab code for the error estimation - Maximization

```
average2=c/d; %calculate the mean of the derivatives

cterr=average2.*(-1/12).*(1e7-1/1e7).^3./1e5.^2 %Error estimation of CTrapezoidalRule

toc %stop the stop watch timer


tic %start the stop watch timer

cserr-cterr %Calculate the difference between the two rules

sign(cserr-cterr) %show which rule has higher error

toc %obtain the elapsed time
```

Matlab code for the error estimation - Maximization

```
syms x %create symbolic variable x
tic %start stop watch timer
equation=(abs(cos((1-x)/x))/x/(1-x))*exp(-(log((1/x)/x)-3)^2);
d4equation=diff(equation,4); %obtain the fourth derivative of the equation
a=0;b=0; %initialize the variables
for n=1/1e7:1e5:1e7 %define the interval and the increment
    a=a+subs(d4equation,x,n); %for loop for the sum of the derivatives at points
    b=b+1; %order increases by 1 after sum of a
end
average1=a/b; %calculate the mean of the derivatives
cserr=average1.*(1/2880).*(1e7-1/1e7).^5./1e5.^4 %Error estimation of CsimpsonRule
toc %stop the stopwatch timer and show the elapsed time
```

R Codes for the integral estimation - Maximization

```
#Calculate the running time
runtime=function(fn){
  start_time=Sys.time()
  fn
  end_time=Sys.time()
  return(-1*(start_time-end_time))
}
runtime(CTrape(f3, 0, 100000, 10000000))
runtime(CSimp(f3, 0, 100000, 10000000))
```

Results - Maximization

	CTrape	CSimp
Integral	1.128537	1.128537
Sign of the difference of the errors[Csimp-Ctrape]	Positive (i.e. $Error_{Simposon's} > Error_{Trapezoidal}$)	
Running Time(seconds) **[Integral]**	0.8330648	1.493879
Running Time(seconds) [Error estimation]	163.353260	283.826745
Running Time(seconds) [Sign of difference]	75.48112	

Interesting Finding - Quadrature

Rate of Convergence: Substitution >>> Maximization of bound
Error on the approaches: Substitution > Maximization

Running Time & Error: Composite Simpson's > Composite Trapezoidal Rule

Error estimation decrease with h -> Error is proportional to h^2

The references suggests the improvement of trapezoidal rule – Simpson's rule

MARKOV CHAIN MONTE CARLO SAMPLING

QUESTION 4

Calculate the correlation between x&y of the following distribution:

METHOD 1: METROPOLIS-HASTINGS ALGORITHM

Idea:

- Random Walk, the next point is a small steps from the current point, by a proposal distribution.
- Accept/Move to the proposed new point if it has higher function value, otherwise stays.
- Points stay long in where function value is high, and the density approximates to target distribution.
- It's similar to rejection method but in higher dimension.

METHOD 2: GIBBS SAMPLING

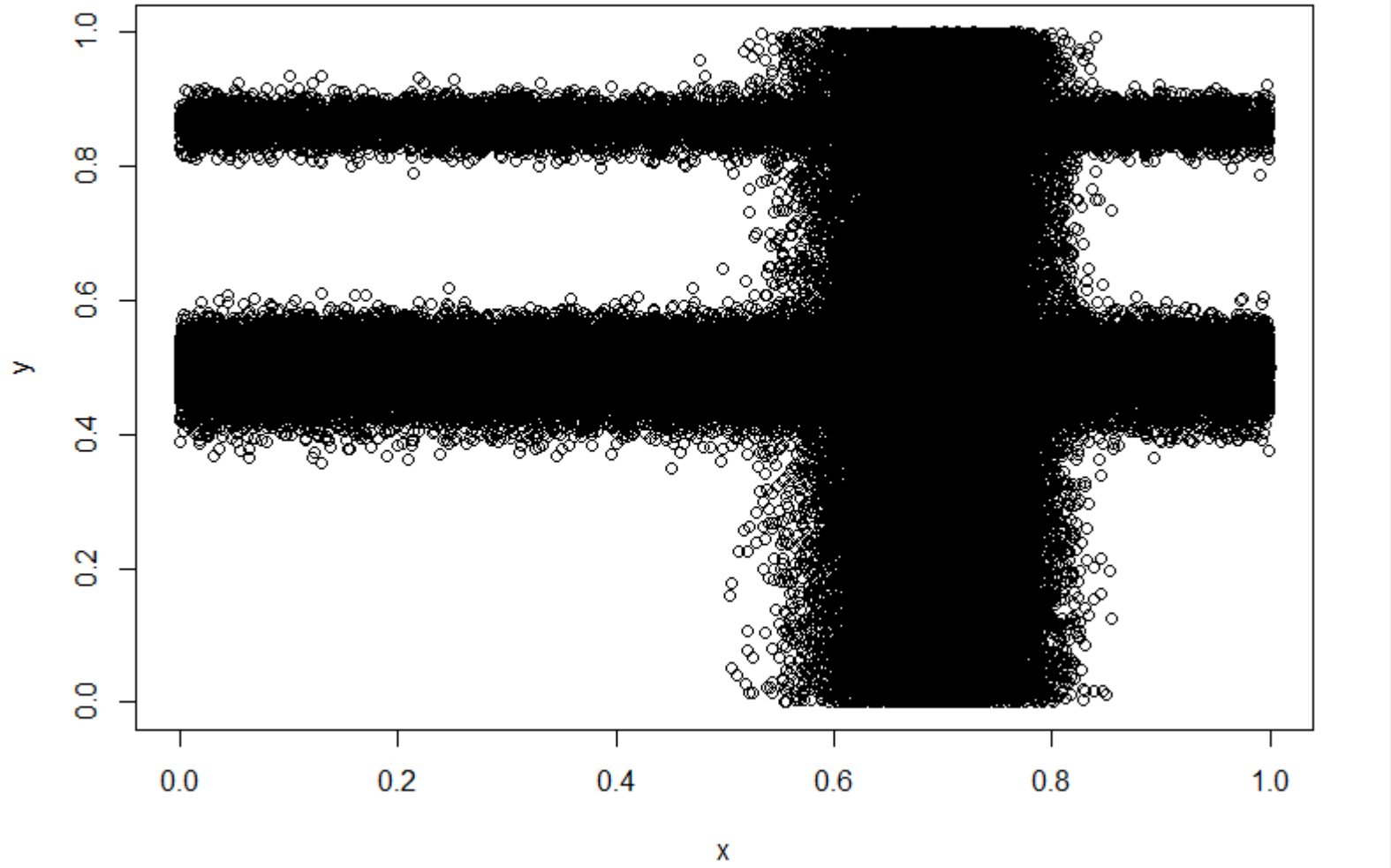
Method 2a: Gibbs Sampling with rejection method

Method 2b: Gibbs Sampling with one dimensional Metropolis Algorithm

- Proposal distribution: U[0,1]

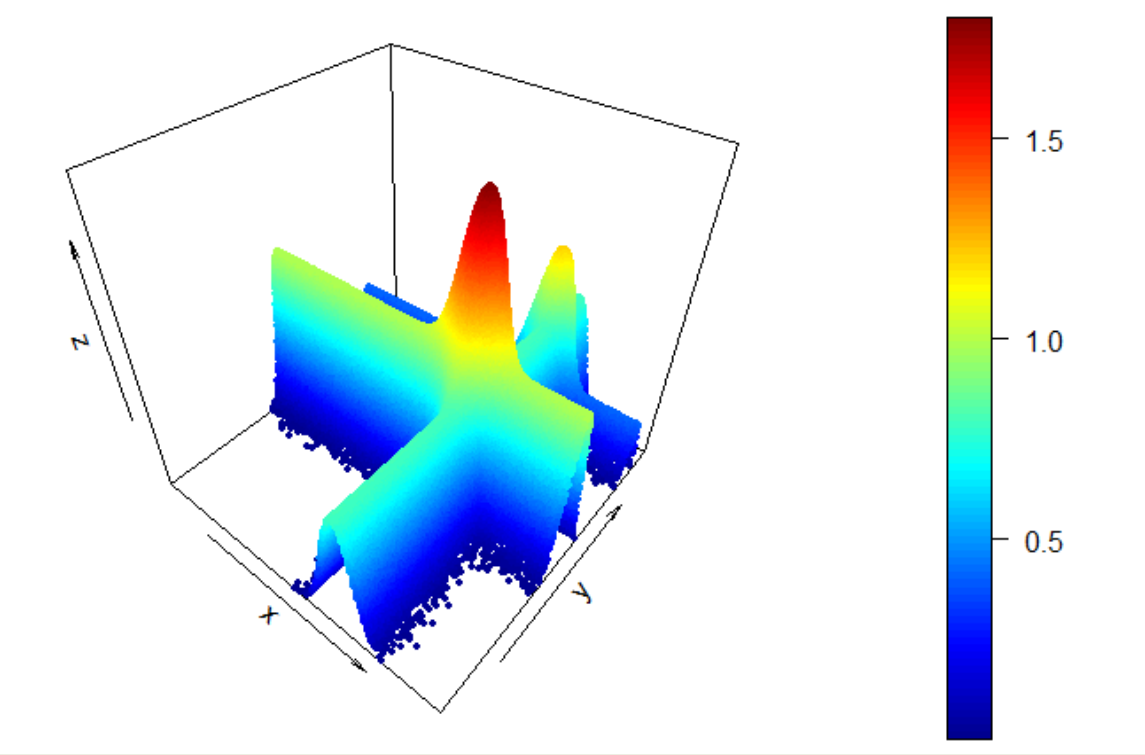
MCMC Process Simulation

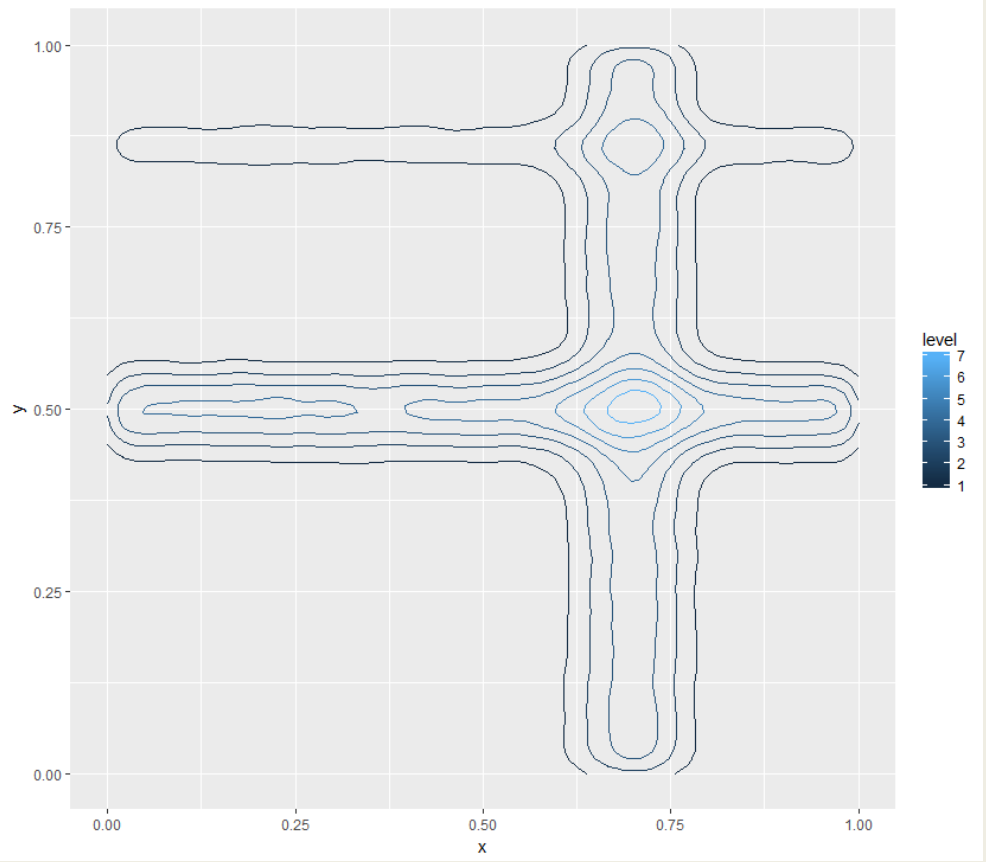
- The points in the beginning move rigorously and it may not in the high probability region, so we usually burn the points in the beginning. Here we burn 1,000 points since it appears that points are in high probability region when n=1,000.



PLOTS OF SAMPLED FUNCTION

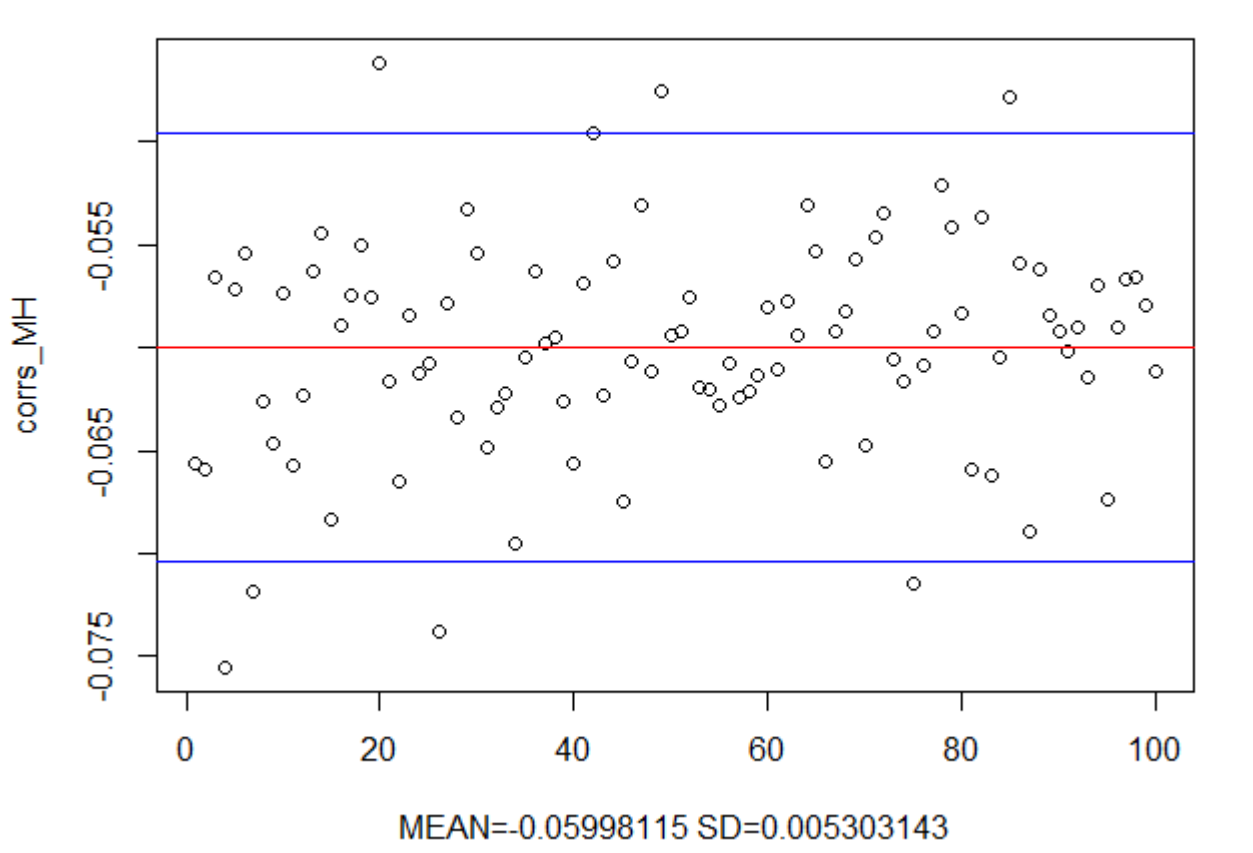
- In each chain, we sample 100,000 (1e5) pairs of points
- We burn 1,000(1e3) points, therefore 99,000 points remain
- Color of the 2d density plot become lighter, indicating that points are converge to the high value region

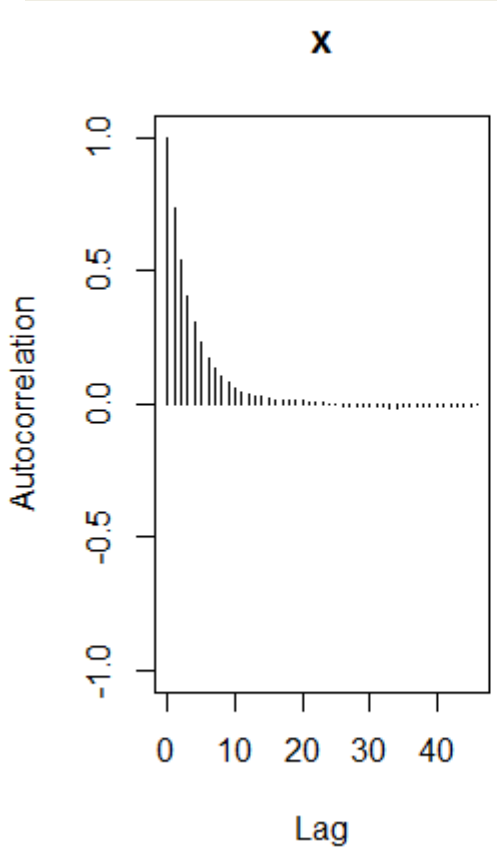


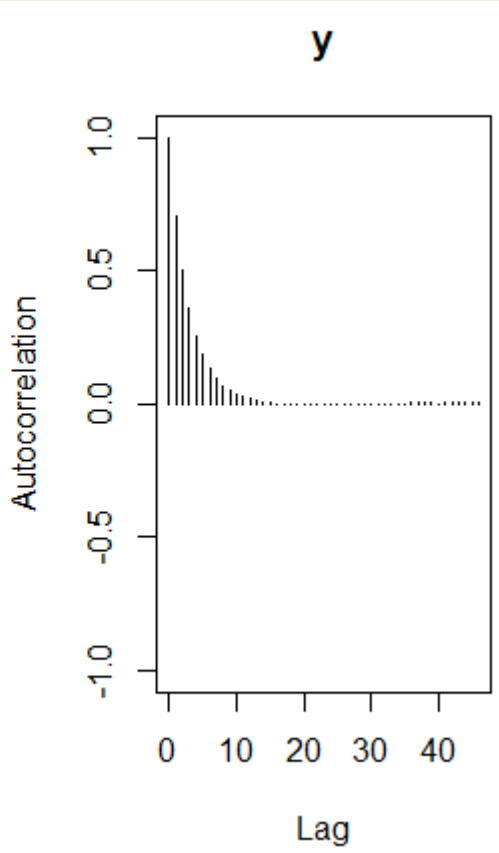


RESULT OF METHOD1: METROPOLIS-HASTINGS ALGORITHM

- Mean of correlation of 100 chain = -0.05990, SD = 0.005303, mean of acceptance rate =0.279
- ACF Plot: ACF decay as time lag k increase, implies stationary, satisfy the condition of MCMC
- Runtime for 1 chain: 53.08231 secs, Runtime for 100 chains: 88.9071 mins (5334.426 secs)

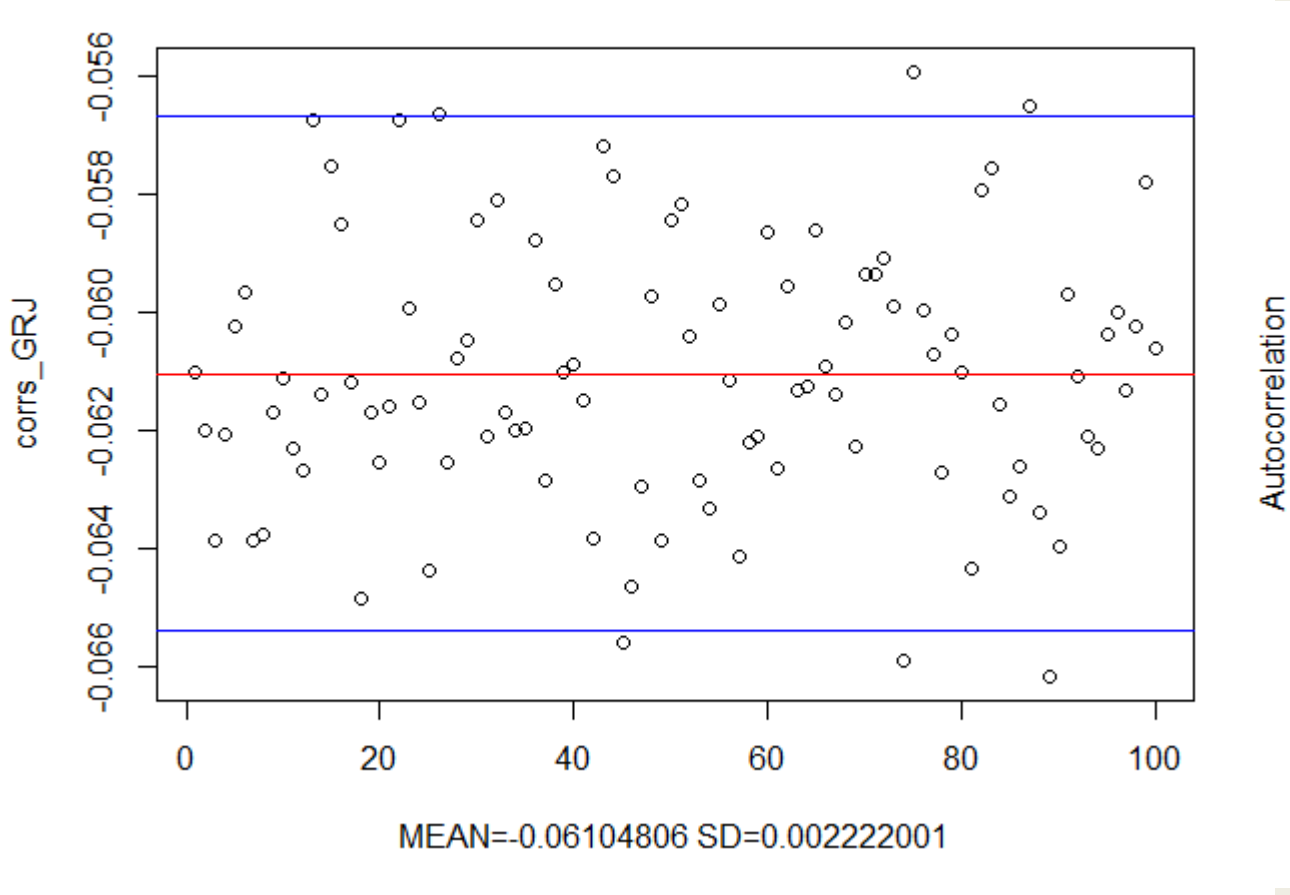


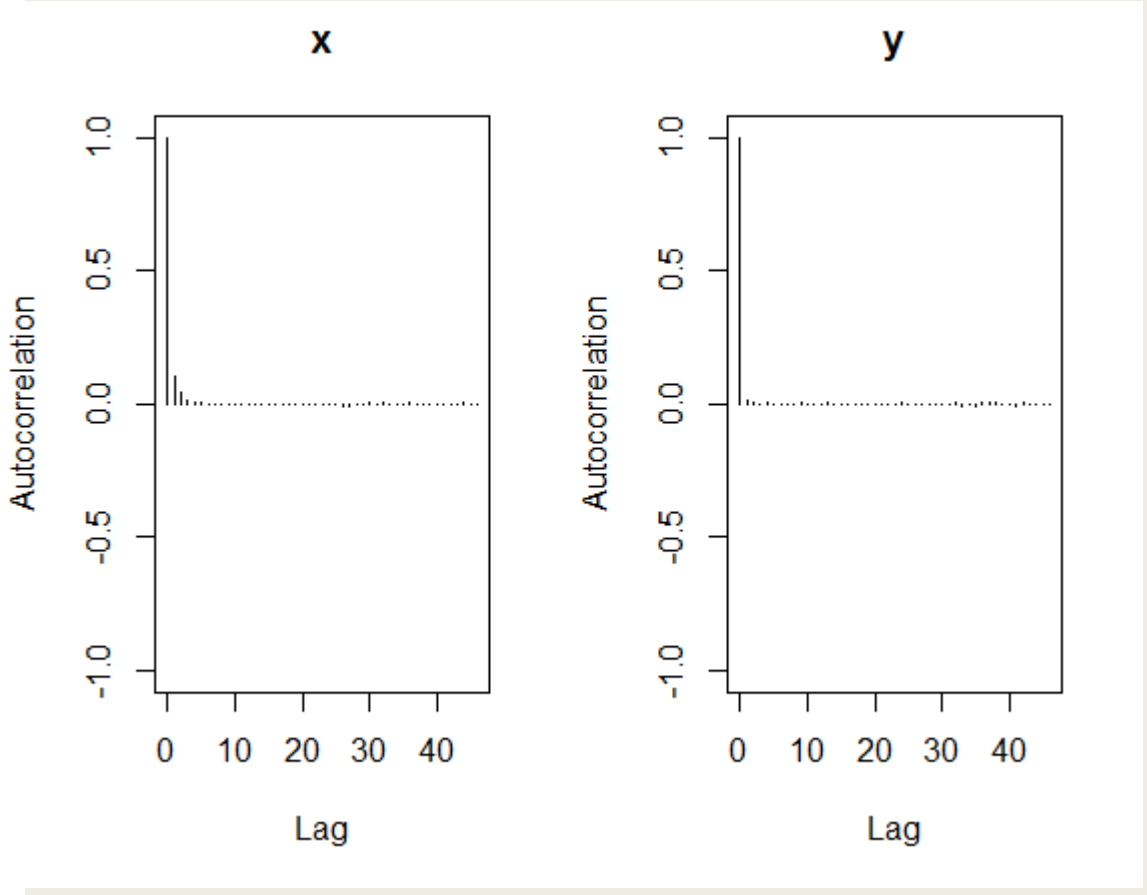




RESULT OF METHOD2A: GIBBS SAMPLING WITH REJECTION METHOD

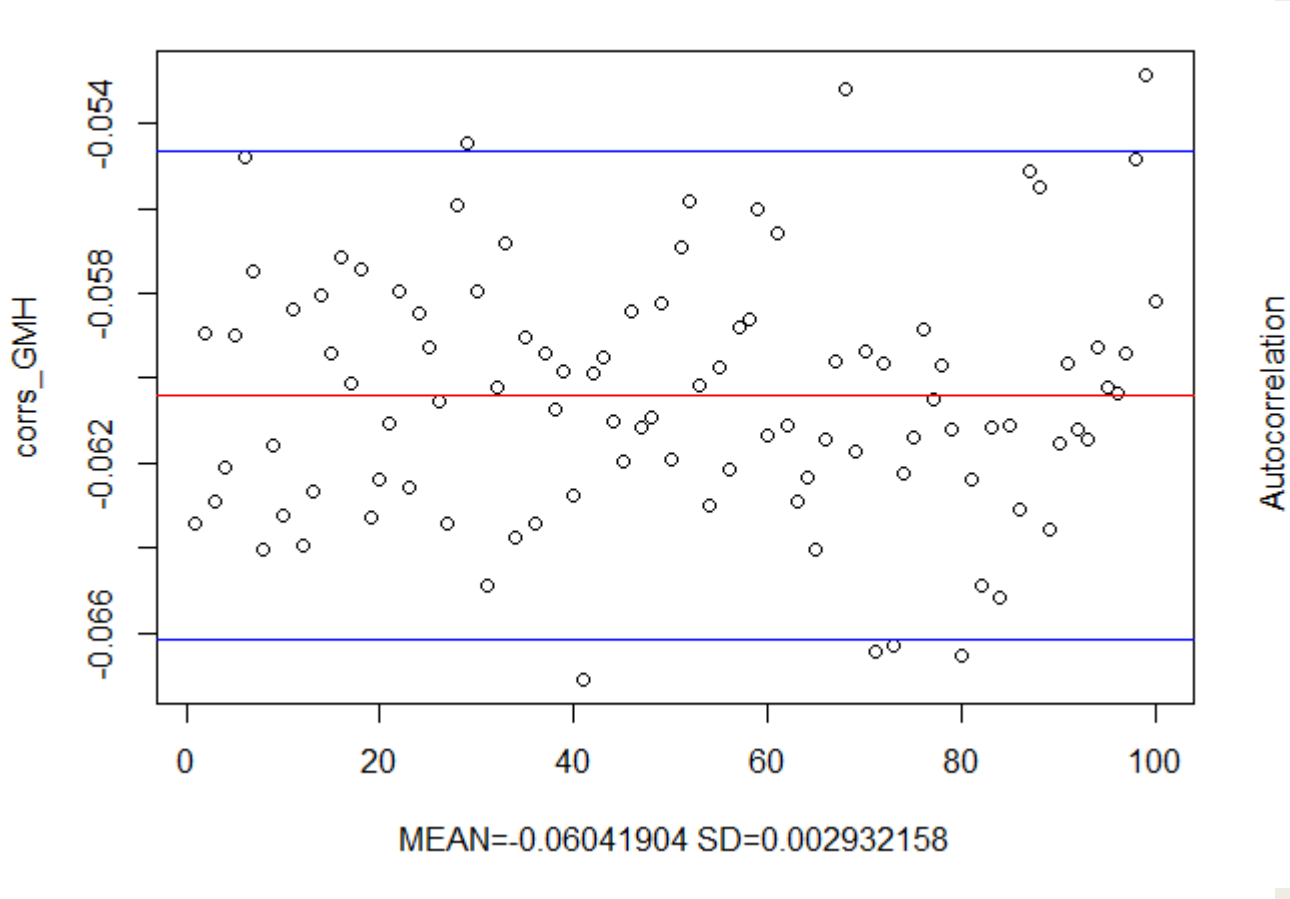
- Mean of correlation of 100 chain = -0.06104806, SD = 0.002222001
- ACF Plot: ACF decay as time lag k increase, implies stationary, satisfy the condition of MCMC
- Runtime for 1 chain: 30.66692 secs, Runtime for 100 chains: 50.4084 mins (3024.50373 secs)

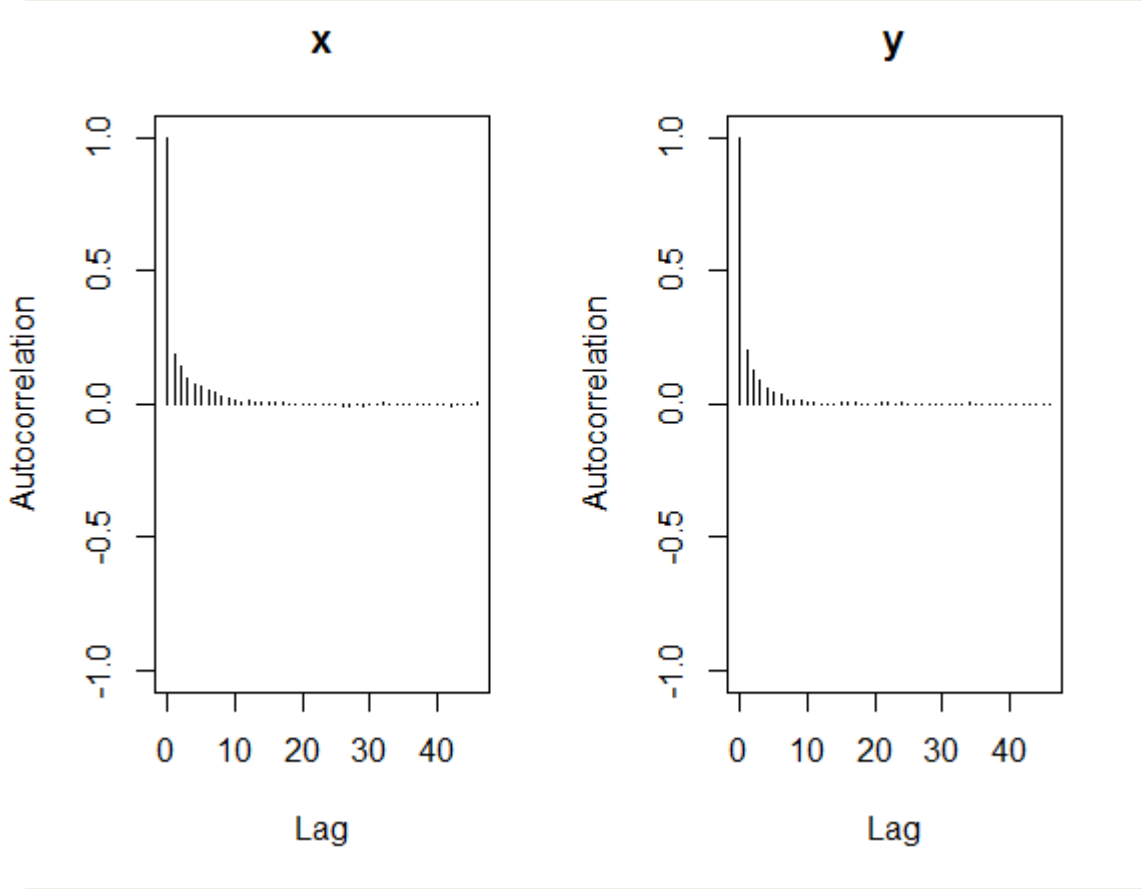




RESULT OF METHOD2B: GIBBS SAMPLING WITH 1D METROPOLIS

- Mean of correlation of 100 chain = -0.06041904, SD = 0.002932158
- ACF Plot: ACF decay as time lag k increase, implies stationary, satisfy the condition of MCMC
- Runtime for 1 chain: 2.72927 secs, Runtime for 100 chains: 4.556789 mins (273.40733 secs)





COMPARISON OF RESULTS

Method		Mean	SD	95% C.I.	Running Time	Average Acceptance Rate
Metropolis-Hastings		-0.05990	0.005303	(-0.07037, -0.04959)	53.08231 secs	0.279
Gibbs Sampling	With Rejection Method	-0.06105	0.002222	(-0.06540, -0.05669)	30.66692 secs	1
	With 1D Metropolis	-0.06042	0.002932	(-0.06617, -0.05467)	2.72927 secs	

- Gibbs Sampling is faster than Metropolis-Hasting Algorithm and the fastest one is the method combining above two.
- Gibbs Sampling with Rejection Method yield the smallest confidence interval, its result concentrates more than the others, meaning that it gives more accurate answer.
- Gibbs Sampling with 1D Metropolis is the most efficient one, which produced similar result as other with much less amount of time.

Q&A

THANK YOU!