Final 2

Erick Martinez - 216087033

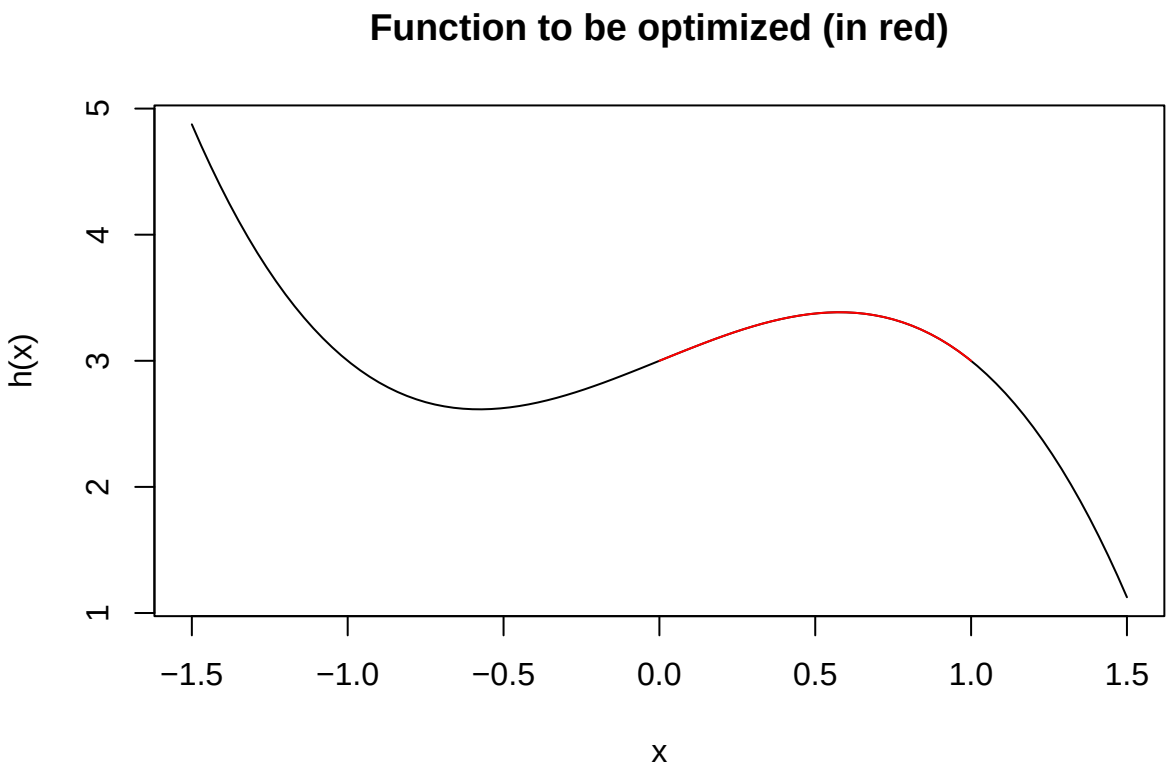
2023-04-16

I am conducting a simulation study for different optimization methods to compare their efficiency and efficacy. To do this we first a unimodal function, and then I will be comparing the results by optimizing a multimodal function. I will be comparing benefits and drawback for each method.

Unimodal Function

We choose the function $h(x) = x^5 - x^{23} + 2$ on the interval $[0, 1]$

```
# Function
h = function(x) {-x^3 + x + 3}
h_prime = function(x) {-3*x^2 + 1}

# Plot
curve(h, from=-1.5, to=1.5, main="Function to be optimized (in red)")
curve(h, from=0, to=1, col="red", xlim=c(-1.5, 1.5), add=TRUE)
```



Newton-Raphson method

By assigning $g = h'$ and $g' = h''$

```
nr_opt = function(n, x0, tol){
  ## Function
  g = function(x) {-3*x^2 + 1}
  g_prime = function(x) {-6*x}
  ## Newton-Raphson algorithm
  i = 0
  while (i < n) {
    x = x0 - (g(x0) / g_prime(x0))
    if (abs(x-x0) < tol){
      return(x)
      break
    }

  x0 = x
  i = i+1
  }
return(x)
}

# Results
x_opt = nr_opt(100, 0.5, 1e-8)
```

```
x_opt ; h(x_opt)
```

```
## [1] 0.5773503
```

```
## [1] 3.3849
```

```
x_opt = nr_opt(100, 1, 1e-10)
x_opt ; h(x_opt)
```

```
## [1] 0.5773503
```

```
## [1] 3.3849
```

```
# ---
optimize(h, interval=c(0, 1), maximum=TRUE)
```

```
## $maximum
## [1] 0.5773569
##
## $objective
## [1] 3.3849
```

Basic Monte Carlo optimization (using a uniform distribution)

```
# Parameters and function to be optimized
h = function(x) {-x^3 + x + 3}

# Basic MC optimization function
basic_opt = function(n){
  ## RV generator
  U = runif(n, min=0, max=1)

  ## Basic stochastic optimization
  hh = h(U)

  loc = which.max(hh)

  return(U[loc])
}

# Results
x = basic_opt(100)
x ; h(x)
```

```
## [1] 0.5786461
```

```
## [1] 3.384897
```

```
x = basic_opt(1000)
x ; h(x)
```

```
## [1] 0.5780531
```

```
## [1] 3.384899
```

```
x = basic_opt(10000)
x ; h(x)
```

```
## [1] 0.5773447
```

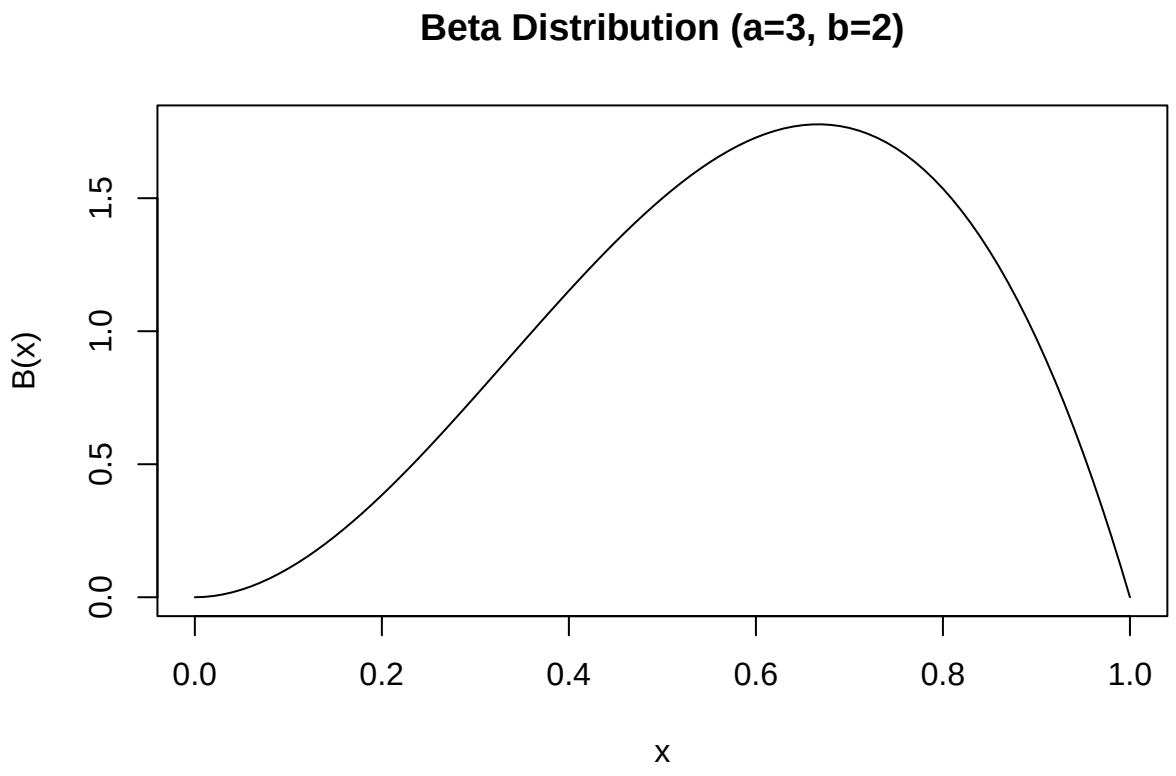
```
## [1] 3.3849
```

Monte Carlo optimization (something better than uniform distribution)

Let's choose different densities of g and compare: **a)** $g(x) = beta$

We can use beta since our problem is bounded for the interval [0,1]

```
# Plot
curve(dbeta(x, shape1=3, shape2=2), from=0, to=1, ylab="B(x)", main="Beta Distribution (a=3, b=2)")
```



```r
# Function
h = function(x) {-x^3 + x + 3}

beta_opt = function(n){
  ## RV generator
  X = rbeta(n, shape1=3, shape2=2)

  ## Stochastic optimization
  hh = h(X)

  loc = which.max(hh)

  return(X[loc])
}

# Results
x = beta_opt(100)
x ; h(x)
```

```
## [1] 0.5815666
```

```
## [1] 3.384869
```

```r
x = beta_opt(1000)
x ; h(x)
```

```
## [1] 0.5773696
```

```
## [1] 3.3849
```

```r
x = beta_opt(10000)
x ; h(x)
```

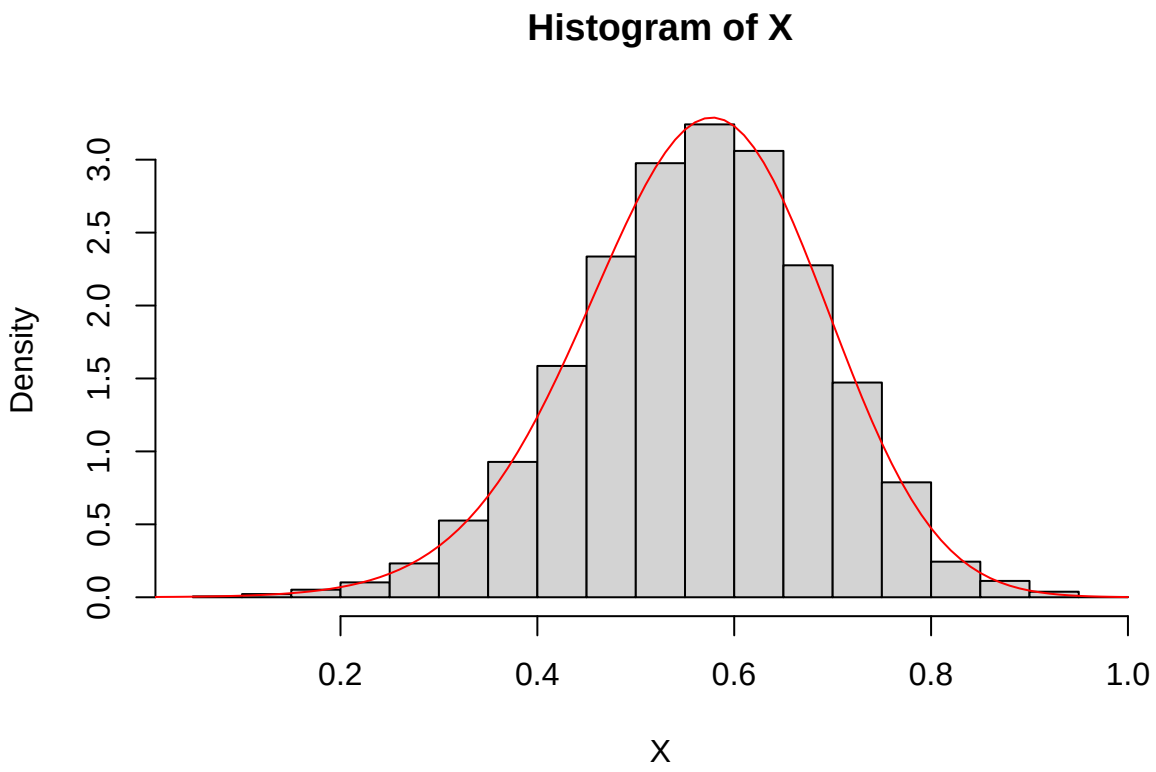
```
## [1] 0.577382
```

```
## [1] 3.3849
```

b) $g \propto e^h$ using $g(\theta) = \frac{e^{h(\theta)}}{\int_0^1 e^{h(\theta)} d\theta}$

```r
# Define the temperature
T = 0.05
# Integrate e^h
int = integrate(function(x) exp(h(x)/T), lower=0, upper=1)

# Store g and the non-uniform density f
g = function(x) {exp(h(x)/T) / int$value}

f = function(x) dbeta(x, shape1=3, shape2=2)

# Find the upper bound for the accept reject algorithm
```

```r
upper_bound = optimize(function(x) g(x)/f(x), interval=c(0,1), maximum=TRUE)
M = upper_bound$objective

# Build accept-reject algorithm
ac_rej = function(n){
  X = rep(0, n)
  i = 1

  while(i <= n){
    U = runif(1, min=0, max=1)
    Y = rbeta(1, shape1=3, shape2=2)

    if (M*U <= g(Y)/f(Y)) {
      X[i] = Y
      i = i + 1
    }
  }
return(X)
}

# Density histogram
X = ac_rej(10000)
hist(X, freq=FALSE)
curve(g, from=0, to=1, col="red", add=TRUE)
```



```
# Build the optimization function
ar_opt = function(n) {
  ## RV generator
  X = ac_rej(n)

  ## Stochastic optimization
  hh = h(X)

  loc = which.max(hh)

  return(X[loc])
}

# Results
x = ar_opt(100)
x; h(x)
```

```
## [1] 0.5760122
```

```
## [1] 3.384897
```

```
x = ar_opt(1000)
x; h(x)
```

```
## [1] 0.5773041
```

```
## [1] 3.3849
```

```
x = ar_opt(10000)
x; h(x)
```

```
## [1] 0.5773595
```

```
## [1] 3.3849
```

Simulated annealing

```
simulated_annealing_opt = function(n, tol){
  ## Define the annealing parameters
  min_iter = 10
  max_iter = n
  T = 1 / (1 + 1:max_iter)^2
  scale = 5*sqrt(T)

  # Initial values
  X = runif(1, min=0, max=1)
  h_val = h(X)
  h_cur = h_val
```

```
  iter = 1
  diff = 1

  ## Annealing loop
  while(diff > tol) {
    prop = X[iter] + runif(1, min=-1, max=1) * scale[iter]

    ## Condition for accepting or rejecting the proposal value
    if ((prop > 1) || (prop < 0) || (log(runif(1))*T[iter] > h(prop)-h_cur))  prop = X[iter]
    X = c(X, prop)
      h_cur = h(prop)
      h_val = c(h_val, h_cur)

      ## Stopping condition
      if ((iter > min_iter) && (length(unique(X[(iter/2):iter])) > 1))  diff = max(h_val) - max(h_val[1

      iter = iter + 1

      ## Second stopping condition (max iterations)
      if ((iter > max_iter))  break
  }

x_max = tail(X, n=1)
h_x = tail(h_val, n=1)
return(list(x=x_max, hx=h_x))
}

# Results
results = simulated_annealing_opt(100, 1e-10)
results$x ; results$hx
```

```
## [1] 0.570776
```

```
## [1] 3.384826
```

```
results = simulated_annealing_opt(1000, 1e-10)
results$x ; results$hx
```

```
## [1] 0.4472661
```

```
## [1] 3.357792
```

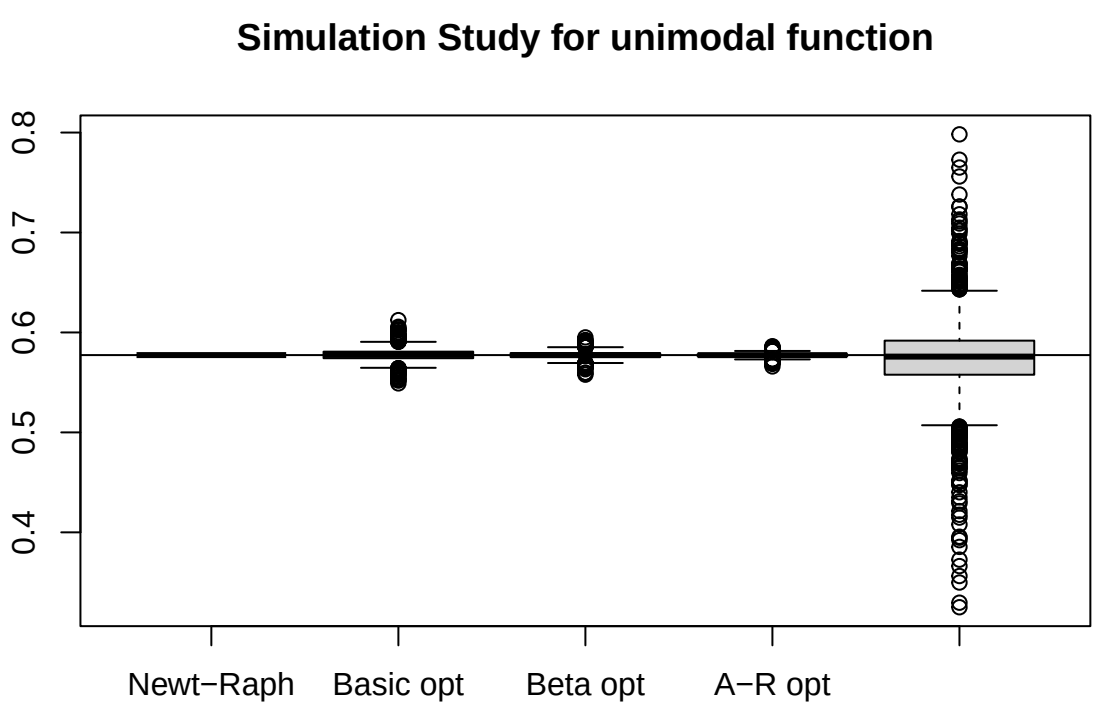
```
results = simulated_annealing_opt(10000, 1e-12)
results$x ; results$hx
```

```
## [1] 0.5675164
```

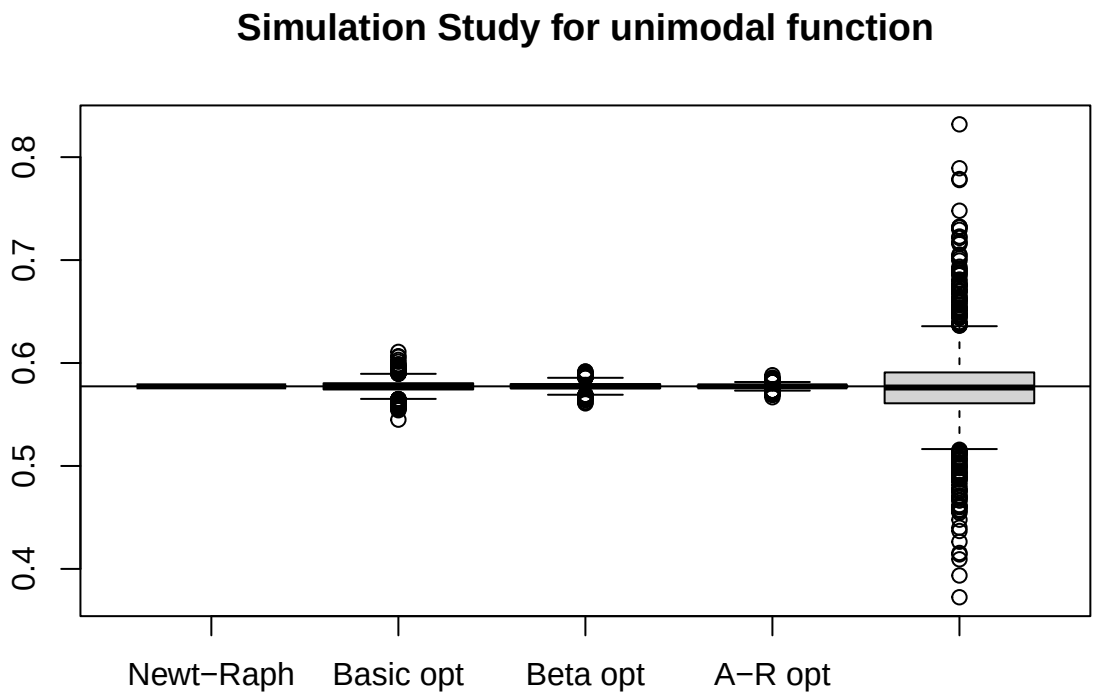
```
## [1] 3.384734
```

Simulation Study

```r
# Create a function to perform an estimator
estimator = function(n, tol, B=1000) {
  ## Newton-Raphson
  res1 = rep(0,B)

  ## Basic Optimization
  res2 = rep(0,B)

  ## Beta optimization
  res3 = rep(0,B)

  ## Accept-Reject g density optimization
  res4 = rep(0,B)

  ## Simulated annealing optimization
  res5 = rep(0,B)

  ## Iterate the estimator
  for (i in 1:B){
    ### Store each iteration
    m1 = nr_opt(n, 0.5, tol)
    m2 = basic_opt(n)
    m3 = beta_opt(n)
    m4 = ar_opt(n)
    m5 = simulated_annealing_opt(n, tol)

    res1[i] = m1

    res2[i] = m2

    res3[i] = m3

    res4[i] = m4

    res5[i] = m5$x

  }

boxplot(res1, res2, res3, res4, res5, names=c("Newt-Raph", "Basic opt", "Beta opt", "A-R opt", "Sim. an
abline(h=sqrt(1/3))
}

# Perform the simulation
estimator(100, 1e-10)
```




```
estimator(100, 1e-12)
```

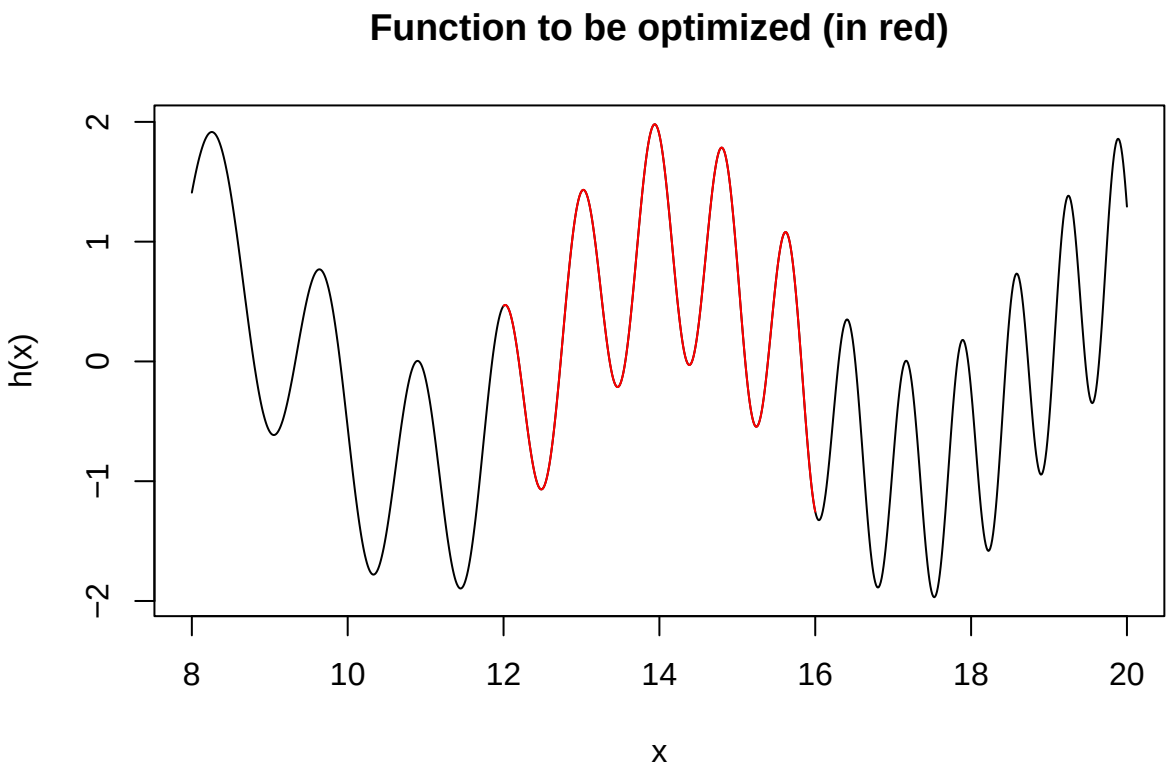


When I use the temperature decrease $T = 1/log(1 + 1 : max_i ter)$ with the scale defined by $\sqrt{T}$, the box plot for the simulated annealing has extremely large first and third quartiles and the value is off. With the temperature decrease and scaling chosen is still off but it converges and reduces the IQR.

Multimodal Function

E choose the function $h(x) = sin(x) + sin(\frac{x^2}{4} - 3)$ on the interval $[12, 16]$

```
# Function
h = function(x) {sin(x) + sin(x^2/4 -3)}
h_prime = function(x) {cos(x) + cos(x^2/4 -3)*x/2}

# Plot
curve(h, from=8, to=20,n=1000, main="Function to be optimized (in red)")
curve(h, from=12, to=16,n=1000, col="red", add=TRUE)
```



Newton-Raphson method

By assigning $g = h'$ and $g' = h''$

```
nr_opt = function(n, x0, tol){
  ## Function
  g = function(x) {cos(x) + cos(x^2/4 -3)*x/2}
  g_prime = function(x) {-sin(x) + cos(x^2/4 -3)/2 - x^2*sin(x^2/4 -3)/4}
  ## Newton-Raphson algorithm
  i = 0
  while (i < n) {
    x = x0 - (g(x0) / g_prime(x0))
    if (abs(x-x0) < tol){
      return(x)
      break
    }

  x0 = x
  i = i+1
  }
return(x)
}

# Results
x_opt = nr_opt(100, 12, 1e-8)
x_opt ; h(x_opt)
```

```
## [1] 12.02156
```

```
## [1] 0.4715719
```

```
x_opt = nr_opt(100, 1, 1e-10)
x_opt ; h(x_opt)
```

```
## [1] 1.064513
```

```
## [1] 0.4623326
```

```
# ---
optimize(h, interval=c(12, 16), maximum=TRUE)
```

```
## $maximum
## [1] 14.79907
##
## $objective
## [1] 1.785368
```

It can be seen that for this case the Newton-Raphson method is sensitive to the initial condition x_0 since it stops once a maximum is found, regardless on whether its a local maximum or a global maximum!

Basic Monte Carlo optimization (using a uniform distribution)

```
# Parameters and function to be optimized
h = function(x) {sin(x) + sin(x^2/4 -3)}

# Basic MC optimization function
basic_opt = function(n){
  ## RV generator
  U = runif(n, min=12, max=16)

  ## Basic stochastic optimization
  hh = h(U)

  loc = which.max(hh)

  return(U[loc])
}

# Results
x = basic_opt(100)
x = h(x)
```

Monte Carlo optimization (something better than uniform distribution)

Let's choose different densities of g and compare: **a)** $g(x) = Truncated\ normal\ distribution\ \epsilon\ [12, 16]$

With the truncated N we can set the probabilites outside of the interval to be 0

```r
h = function(x) {sin(x) + sin(x^2/4 -3)}

norm_opt = function(n){
  ## RV generator
  X = rtruncnorm(n, a=12, b=16, mean=14, sd=1)

  ## Stochastic optimization
  hh = h(X)

  loc = which.max(hh)

  return(X[loc])
}

# Results
x = norm_opt(1000)
x
```

```
## [1] 13.94202
```

```r
h(x)
```

```
## [1] 1.980142
```

b) $g \propto e^h$ using $g(\theta) = \frac{e^{h(\theta)}}{\int_0^1 e^{h(\theta)} d\theta}$

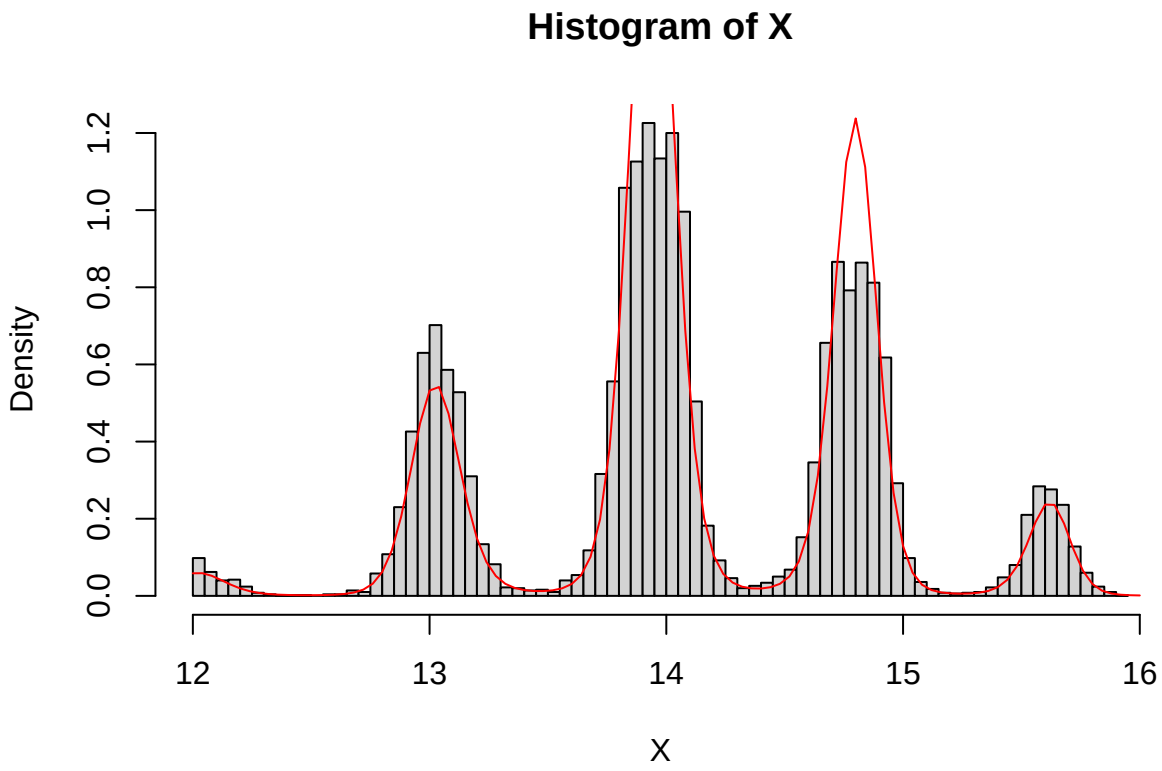
```r
h = function(x) {sin(x) + sin(x^2/4 -3)}

# Define the temperature
T = 0.433
# Integrate e^h
int = integrate(function(x) exp(h(x)/T), lower=12, upper=16)

# Store g and the non-uniform density f
g = function(x) {exp(h(x)/T) / int$value}

f = function(x) dtruncnorm(x,a=12, b=16, mean=14, sd=1)

# Find the upper bound for the accept reject algorithm
upper_bound = optimize(function(x) g(x)/f(x), interval=c(12, 16), maximum=TRUE)
M = upper_bound$objective
M
```

```
## [1] 2.17707
```

```r
acceptance_rate = 1/M
acceptance_rate
```

```
## [1] 0.4593329
```

```r
# Build accept-reject algorithm
ac_rej = function(n){
  X = rep(0, n)
  i = 1

  while(i <= n){
    U = runif(1, min=0, max=1)
    Y = rtruncnorm(1, a=12, b=16, mean=14, sd=1)
    if (M*U <= g(Y)/f(Y)) {
      X[i] = Y
      i = i + 1
    }
  }

return(X)
}

# Density histogram
X = ac_rej(10000)
hist(X, breaks=100, freq=FALSE)
curve(g, from=12, to=16, col="red", add=TRUE)
```




```r
# Build the optimization function
ar_opt = function(n) {
  ## RV generator
```

```
  X = ac_rej(n)

  ## Stochastic optimization
  hh = h(X)

  loc = which.max(hh)

  return(X[loc])
}

# Results
x = ar_opt(100)
x;h(x)
```

```
## [1] 13.93734
```

```
## [1] 1.980058
```

```
x = ar_opt(1000)
x; h(x)
```

```
## [1] 13.94005
```

```
## [1] 1.980239
```

```
x = ar_opt(10000)
x; h(x)
```

```
## [1] 13.94007
```

```
## [1] 1.980239
```

For this problem the temperature choice greatly affects the acceptance rate. As a consequence this can cause computational problems, so I am choosing T to maximize the acceptance rate instead.

The accept reject algorithm is having problems!! not a single M*U is never less that the function g/f

Simulated annealing

```
h = function(x) {sin(x) + sin(x^2/4 -3)}
simulated_annealing_opt = function(n, tol){
  ## Define the annealing parameters
  min_iter = 10
  max_iter = n
  T = 1 / (1 + 1:max_iter)^2
  scale = 5*sqrt(T)

  # Initial values
  X = runif(1, min=12, max=16)
```

```
  h_val = h(X)
  h_cur = h_val

  iter = 1
  diff = 1

  ## Annealing loop
  while(diff > tol) {
    prop = X[iter] + runif(1, min=-1, max=1) * scale[iter]

    ## Condition for accepting or rejecting the proposal value
    if ((prop > 1) || (prop < 0) || (log(runif(1))*T[iter] > h(prop)-h_cur))  prop = X[iter]
    X = c(X, prop)
      h_cur = h(prop)
      h_val = c(h_val, h_cur)

      ## Stopping condition
      if ((iter > min_iter) && (length(unique(X[(iter/2):iter])) > 1))  diff = max(h_val) - max(h_val[1

      iter = iter + 1

      ## Second stopping condition (max iterations)
      if ((iter > max_iter))  break
  }

x_max = tail(X, n=1)
h_x = tail(h_val, n=1)
return(list(x=x_max, hx=h_x))
}

# Results
results = simulated_annealing_opt(1000, 1e-8)

results$x ; results$hx
```

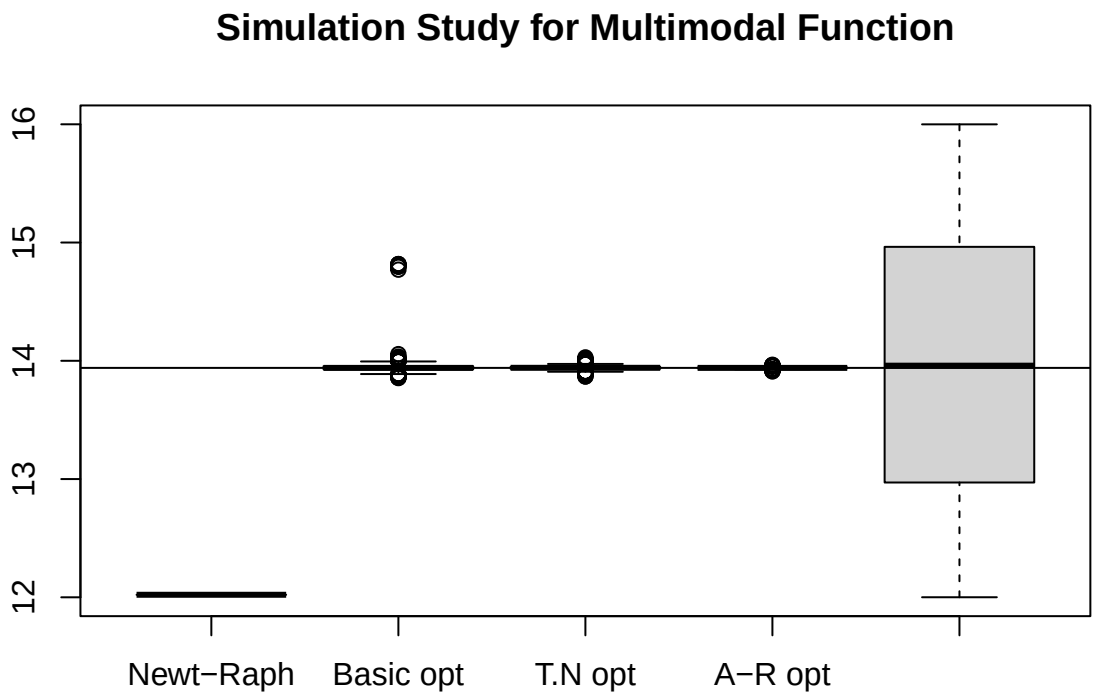
```
## [1] 12.73668
```

```
## [1] 0.02658499
```

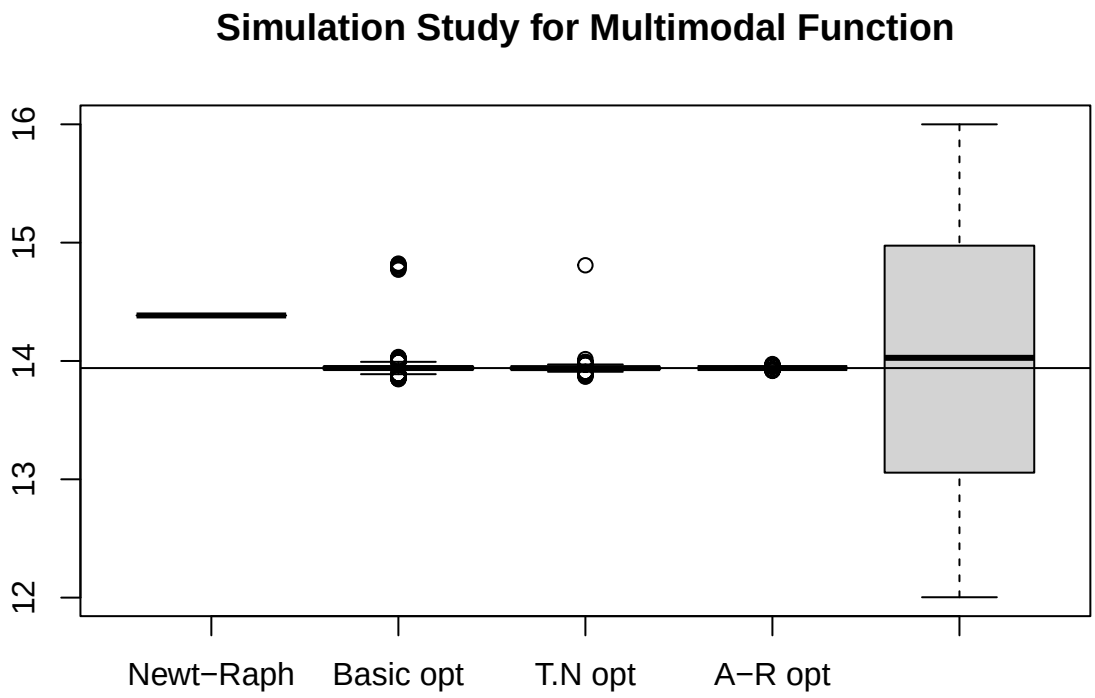
Simulation Study

```
# Create a function to perform an estimator
estimator = function(n,x0, tol, B=1000) {
  ## Newton-Raphson
  res1 = rep(0,B)

  ## Basic Optimization
  res2 = rep(0,B)

  ## Beta optimization
  res3 = rep(0,B)
```

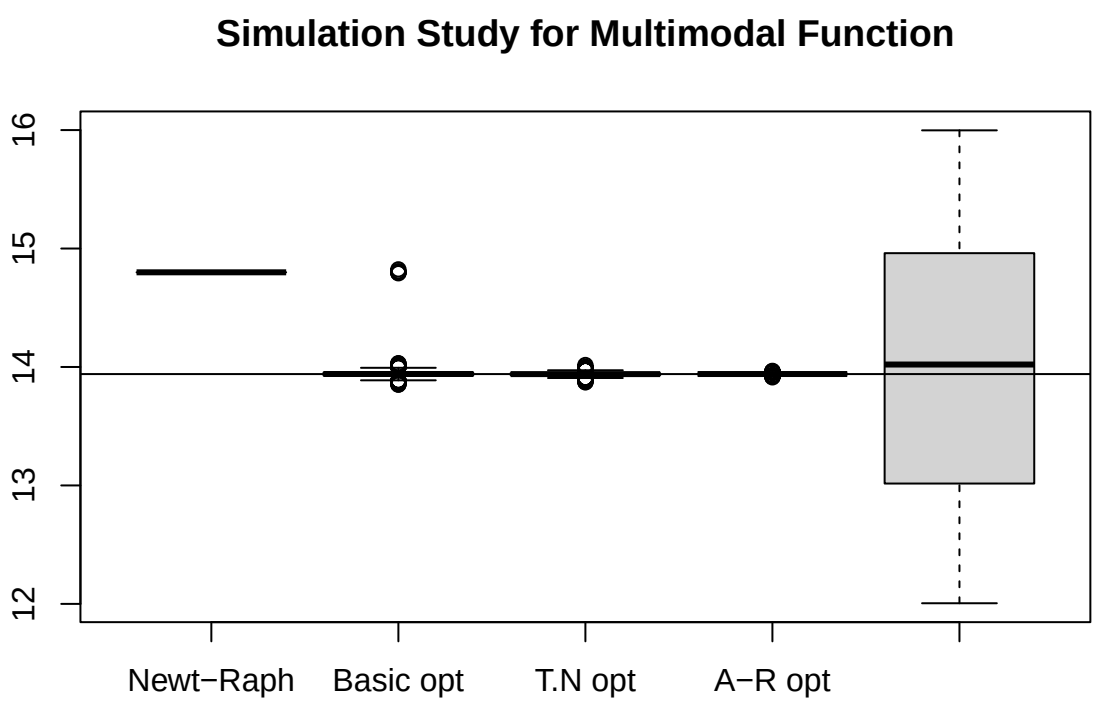
```
  ## Accept-Reject g density optimization
  res4 = rep(0,B)

  ## Simulated annealing optimization
  res5 = rep(0,B)

  ## Iterate the estimator
  for (i in 1:B){
    ### Store each iteration
    m1 = nr_opt(n, x0, tol)
    m2 = basic_opt(n)
    m3 = norm_opt(n)
    m4 = ar_opt(n)
    m5 = simulated_annealing_opt(n, tol)

    res1[i] = m1

    res2[i] = m2

    res3[i] = m3

    res4[i] = m4

    res5[i] = m5$x

  }

boxplot(res1, res2, res3, res4, res5, names=c("Newt-Raph", "Basic opt", "T.N opt", "A-R opt", "Sim. anne
abline(h=13.94)
}

# Perform the simulation
## For x0=12
estimator(n=100, x0=12 ,tol=1e-8)
```




```
## For x0=13.2
estimator(n=100, x0=13.2 ,tol=1e-8)
```




```
## For x0=14.2
estimator(n=100, x0=14.2 ,tol=1e-12)
```


Simulation Study for Multimodal Function
16
15
14
13
12
Newt−Raph
Basic opt
T.N opt
A−R opt