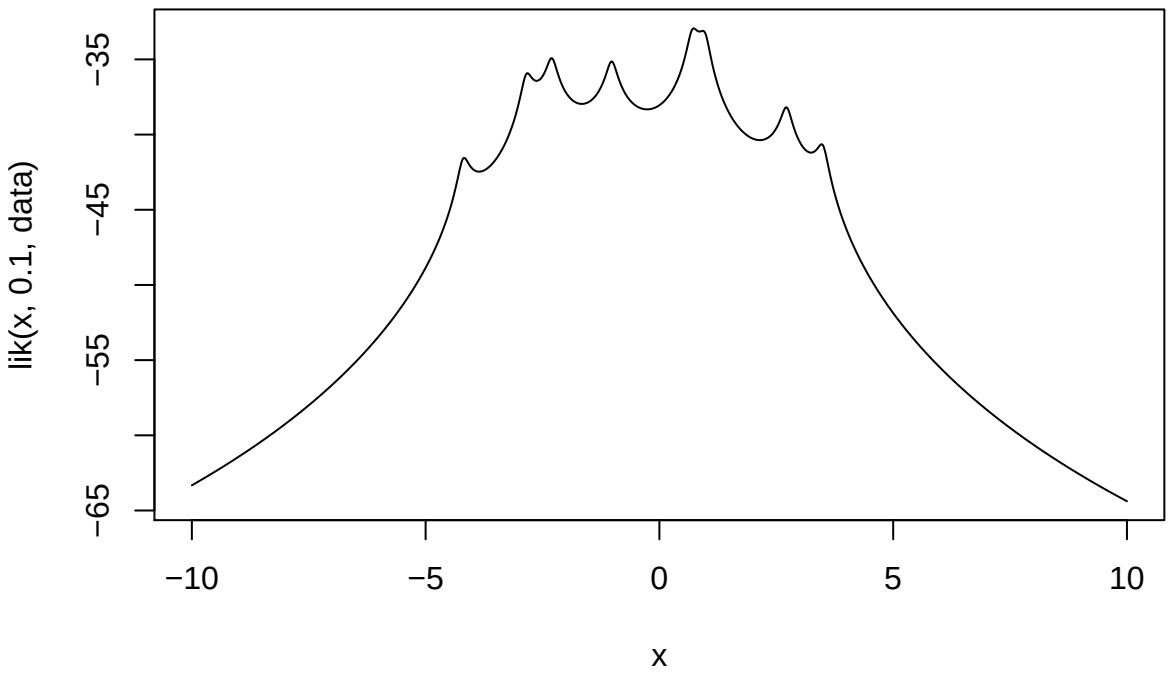
GRADUATE STUDENT STAT 840 A3

Vsevolod Ladtchenko 20895137

Problem 3

a)

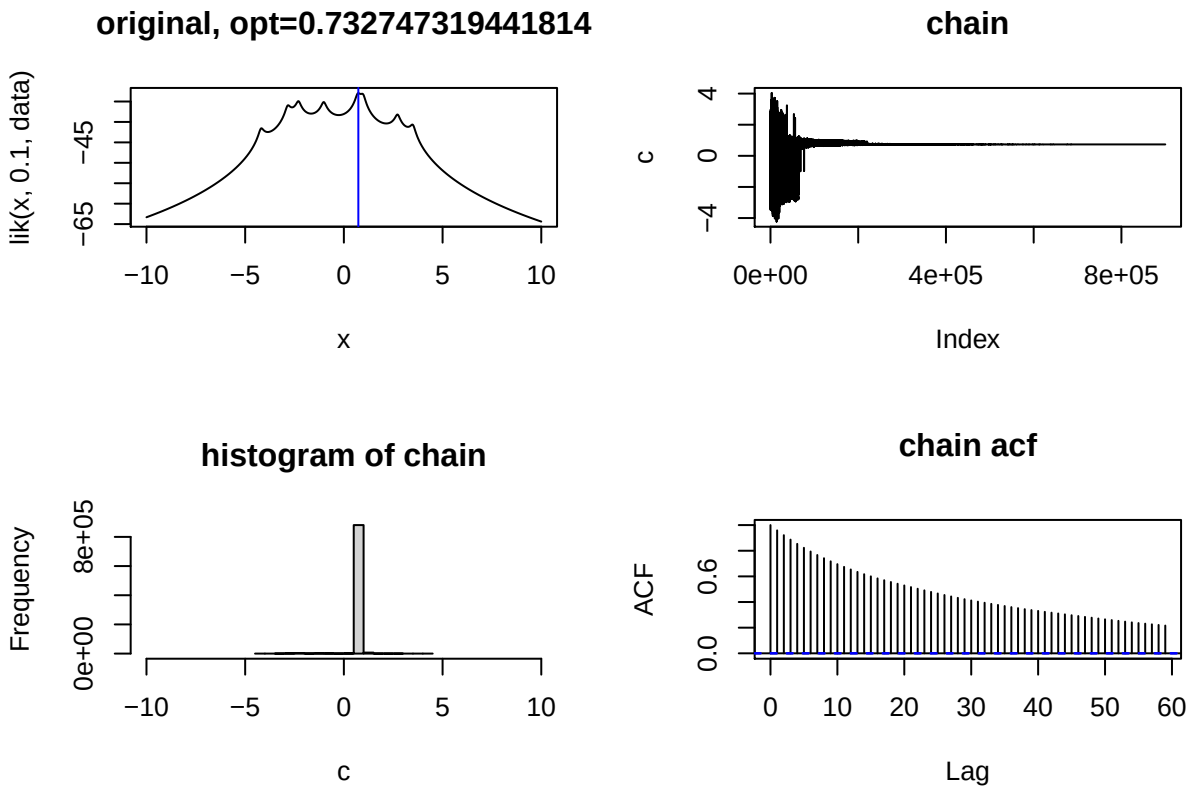
```
data = c(-4.20, -2.85, -2.30, -1.02, 0.70, 0.98, 2.72, 3.50)

lik = function(a,b,data)
{
  n = length(data)
  s = n*log(b) - n*log(pi)
  for (i in 1:n)
    s = s - log(b^2 + (data[i] - a)^2)
  return(s)
}

x = seq(-10,10,length=1000)
plot(x=x,y=lik(x,0.1, data),type='l')
```

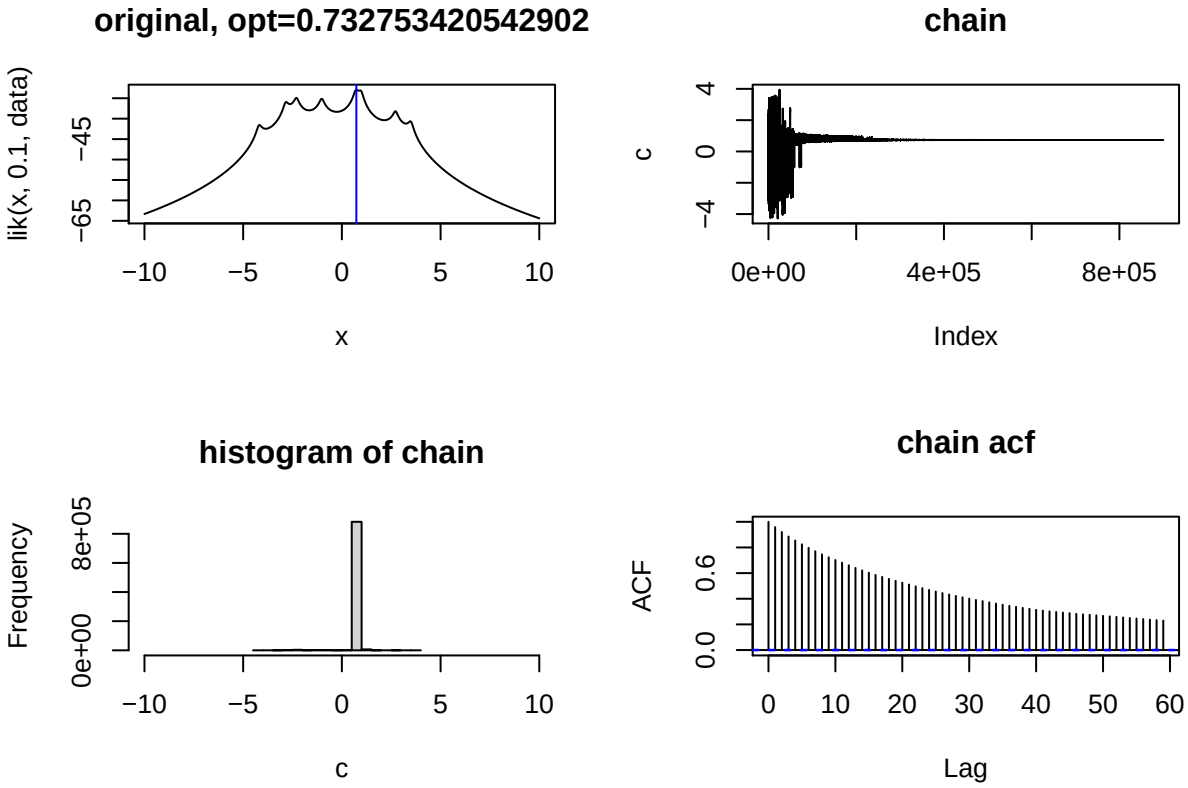


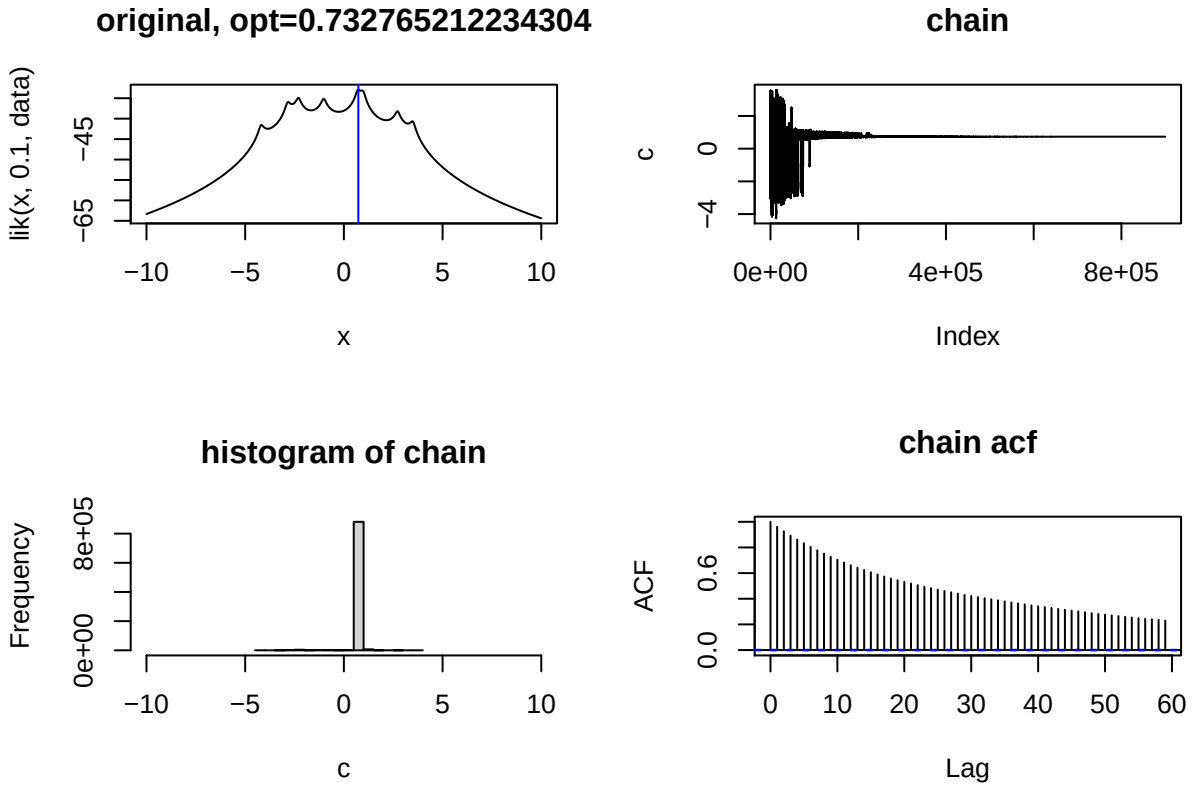
b)

```
simulated_annealing = function(start)
{
  T0 = 10
  Tf = 1/10000000
  n = 1e6

  chain = rep(NA,n)
  chain[1] = start

  for (i in 2:n)
  {
    Ti = T0 * (Tf / T0)^((i-1)/n)
    old = chain[i-1]
    new = rnorm(1,old,1)

    if (runif(1) <= exp((lik(new,0.1,data) - lik(old,0.1,data)) / Ti))
      chain[i] = new
    else
      chain[i] = old
  }
  return(chain)
}
```

```r
par(mfrow=c(2,2))

for (start in c(-2.5, 0, -0.30875))
{
  c = simulated_annealing(start)
  c = c[1e5:1e6] # burn in
  opt = c[length(c)] # optimal value

  plot(x=x,y=lik(x,0.1, data),type='l',main=paste('original, opt=',opt,sep=''))
  abline(v=opt, col="blue")
  plot(c, type='l',main='chain')
  hist(c,xlim=c(-10,10),main='histogram of chain')
  acf(c,main='chain acf')
}
```




original, opt=0.732753420542902
lik(x, 0.1, data)
−65
−45
−10
−5
0
5
10
x
chain
c
−4
0
4
0e+00
4e+05
8e+05
Index
histogram of chain
Frequency
0e+00
8e+05
−10
−5
0
5
10
c
chain acf
ACF
0.0
0.6
0
10
20
30
40
50
60
Lag



We observe that the starting value does not have an effect on the chain. The chain appears to converge smoothly to the function maximum, when we look at its time series plot. We also note that the amount of jumping of the time series is proportional to the current temperature value, which is a decreasing geometric function.

c)

```
# copy from part a
data = c(-4.20, -2.85, -2.30, -1.02, 0.70, 0.98, 2.72, 3.50)

lik = function(a,b,data)
{
  n = length(data)
  s = n*log(b) - n*log(pi)
  for (i in 1:n)
    s = s - log(b^2 + (data[i] - a)^2)

  if (is.nan(s)) # for the sake of the fitness ordering, replace NaN w -Inf
    return(-Inf)
  else
    return(s)
}

# a series of functions for dealing with strings representing 64 bits

empty_str = function() paste(rep('0',64),collapse='')
```

```
random_str = function() paste(sample(c("0","1"),64,replace=T), collapse='')

# https://gallery.rcpp.org/articles/strings_with_rcpp/
# https://stackoverflow.com/questions/50217954/double-precision-64-bit-representation-of-numeric-value-

Rcpp::cppFunction("
std::string dbl_to_str(double d, std::string s)
{
    uint64_t bits = *((uint64_t*)(&d));
    uint64_t bit;
    for (int i = 0; i < 64; i++)
    {
        bit = (bits >> i) & 0x1;
        s[i] = (char)(bit + 48);
    }
    return s;
}", plugins = "cpp11", includes = "#include <Rcpp.h>")

Rcpp::cppFunction("
double str_to_dbl(std::string s)
{
    uint64_t bits = 0;
    uint64_t bit; // must be uint64_t since it gets shifted 63 times
    for (int i = 0; i < 64; i++)
    {
        bit = s[i] - 48;
        bits |= (bit << i);
    }
    return *((double*)(&bits));
}", plugins = "cpp11", includes = "#include <Rcpp.h>")

# test the string to double function
str = empty_str() # allocate string on R side for gc
str = dbl_to_str(0.1, str)
str
```

```
## [1] "0101100110011001100110011001100110011001100110011001100110011110111111100"
```

```
str_to_dbl(str)
```

```
## [1] 0.1
```

```
maybe_mutate = function(s, mutate_p) # mutate a string with given probability
{
  if (runif(1) <= mutate_p) # prob of mutation
  {
    idx = sample(seq(1,64),1) # index where to mutate
    if (substr(s,idx,idx)  == '0')
      substr(s,idx,idx) = '1'
    else
      substr(s,idx,idx) = '0'
  }
  return(s)
}
```

```
update_row = function(str) # generate alpha and fitness given a string
{
  alpha = str_to_dbl(str)
  fitness = lik(alpha, 0.1, data)
  return(list(alpha, fitness, str))
}

get_best_ever = function(best_ever, df)
{
  if (best_ever[[2]] > df[1,2]) # compare best to this generations most fit
    return(best_ever)
  else
    return(df[1,])
}

genetic = function(N = 10, mutate_p = 0.1, epochs = 1000)
{
  # initialize population. fitness function is likelihood, want to maximize.
  df = data.frame(matrix(nrow=N,ncol=3))
  colnames(df) = c("alpha","fitness","string")
  for (i in 1:N)
    df[i,] = update_row(random_str())

  # the next generation
  df2 = df

  # keep track of best ever seen
  best_ever = list(-Inf,-Inf,'.')

  for (epoch in 1:epochs)
  {
    # sort by fitness
    df = df[order(-df$fitness), ]

    # update the best_ever
    best_ever = get_best_ever(best_ever, df)

    # crossover: pair top N/2 with the rest (ignore duplicates)
    mate = matrix(nrow=2,ncol=N/2)
    mate[1,] = sample(seq(1,N/2),N/2,replace=T)
    mate[2,] = sample(seq(1,N),N/2,replace=T)

    for (i in 1:(N/2)) # for each mating pair
    {
      pair = mate[,i]
      p1 = pair[1] # the individual's index
      p2 = pair[2]
      s1 = df$string[p1] # the individual's string
      s2 = df$string[p2]

      # crossover: cut and paste genetic strings
      idx = sample(seq(1,64),1)
      n1 = paste(substring(s1,1,idx),substring(s2,idx+1),sep='')
```

```
      n2 = paste(substring(s2,1,idx),substring(s1,idx+1),sep='')

      # mutate
      m1 = maybe_mutate(n1, mutate_p)
      m2 = maybe_mutate(n2, mutate_p)

      # update rows in the next generation: df2
      df2[p1,] = update_row(m1)
      df2[p2,] = update_row(m2)
    }
    # switch to next generation
    df = df2
  }
  print(best_ever)
  return(df)
}


# the global optimum is about 0.73 according to part (b)
for (pop in c(10, 20, 30))
{
  for (mut in c(0.05, 0.1, 0.15, 0.3, 0.5))
  {
    print(paste("population: ", pop, " mutation: ", mut))
    df_ = genetic(pop, mut, 10000)
  }
}
```

```
## [1] "population:  10  mutation:  0.05"
##   alpha   fitness
## 1   0.5 -35.36936
##                                                                          string
## 1 0011111111111111111111111111111111111111111111111111111101111111100
## [1] "population:  10  mutation:  0.1"
##          alpha   fitness
## 1 4.656563e-10 -38.04776
##                                                                          string
## 1 10101001110110111011010011001010100101111111111111111111110111100
## [1] "population:  10  mutation:  0.15"
##     alpha   fitness
## 1 0.71875 -32.94936
##                                                                          string
## 1 01011111111111111111111111111111111111111111111111101100111111111100
## [1] "population:  10  mutation:  0.3"
##   alpha   fitness
## 4  0.75 -32.95177
##                                                                          string
## 4 00111100000000000000000000000000000000000000000000001011111111100
## [1] "population:  10  mutation:  0.5"
##        alpha   fitness
## 10 0.7327723 -32.93596
##                                                                           string
## 10 10000000101110011010111101110111011110110100111011100111111111100
## [1] "population:  20  mutation:  0.05"
```

```
##   alpha  fitness
## 1 0.125 -37.7069
##                                                                                 string
## 1 1101010111111111111111111111111111111111111111111111111110111111100
## [1] "population:  20  mutation:  0.1"
##      alpha   fitness
## 1 0.749939 -32.95167
##                                                                                 string
## 1 1010001111111111111111111111111111111111101111111111110011111111100
## [1] "population:  20  mutation:  0.15"
##   alpha   fitness
## 1  0.75 -32.95177
##                                                                                 string
## 1 0111000000000000000000000000000000000000000000000001011111111100
## [1] "population:  20  mutation:  0.3"
##      alpha   fitness
## 1 0.734375 -32.93612
##                                                                                 string
## 1 0011010000000000000000000000000000000000000000011110011111111100
## [1] "population:  20  mutation:  0.5"
##    alpha   fitness
## 12  0.75 -32.95177
##                                                                                  string
## 12 1010011110001100000000000000000000000000000000000001011111111100
## [1] "population:  30  mutation:  0.05"
##    alpha  fitness
## 15 0.125 -37.7069
##                                                                                  string
## 15 1111101111110111111111111111111111111111111111111110111111100
## [1] "population:  30  mutation:  0.1"
##   alpha   fitness
## 1   0.5 -35.36936
##                                                                                 string
## 1 1111111111111111111111111111111111111111111111111101111111100
## [1] "population:  30  mutation:  0.15"
##   alpha   fitness
## 1  0.75 -32.95177
##                                                                                 string
## 1 0000110000000000000000000000000000000000000000001011111111100
## [1] "population:  30  mutation:  0.3"
##       alpha   fitness
## 2 0.7324219 -32.93597
##                                                                                 string
## 2 0011011101111111111111111111111111111111111101101110011111111100
## [1] "population:  30  mutation:  0.5"
##       alpha   fitness
## 1 0.7327724 -32.93596
##                                                                                 string
## 1 0000100010010001111100000010000011111011010011101110011111111100
```

The fitness function is the likelihood, which we seek to maximize. For this problem we decided that a 64 bit string would be a natural way to represent a double (seemingly the only numerical data type in R). We can generate such strings randomly, then perform crossover in the genetic algorithm, and mutate them too. To convert them to an actual number, we dove into C++ to write the bit field directly. Since this bit field

represents α, we then feed it to the likelihood to measure the fitness.

Another approach would be to decide on a fixed length string, say 64 again, and use that as a sequence of binary coefficients (1, 0.5, 0.25, etc) on a predefined interval. But that would be a "continuous" problem, and it is interesting to see how the Genetic algorithm performs on the more "crude" problem of generating random doubles. Plus, this way we do not have to make any limiting choices or assumptions on the representation, because our sample space is all possible double precision floating point numbers that the likelihood function can accept.

Increasing the population size between 10,20,30 does not seem to be a very important aspect. At least in our algorithm we have the top N/2 individuals perform crossover with the rest. More individuals would allow more variety and more chances of mutations, but we found that changing the mutation probability and number of iterations (epochs) more effective than the population size. Even with a decent number of epochs, it is a game of chance, and there is no guarantee of success. Increasing the mutation probability is the best for exploring the sample space. But in that case, it is possible to achieve a near global optimum and then lose it (especially with small population sizes). That is why we kept a "best_ever" variable, which showed us that sometimes the population achieved a global optimum and then the population evolved out of it, losing that fitness due to high mutation rates.

Another interesting aspect of this problem is that about half of the initial randomly generated bit strings correspond to values near 0, by nature of floating point being accurate in that range. This automatically put their fitness at a local maximum, for this specific problem's likelihood function. When they crossover with each other, they mostly make more numbers near 0. That is why it is essential for them to crossover with the less fit rest of the population, to introduce genetic variety. Hence our crossover code had a random choice of the top N/2 most fit individuals paired with a random choice of N/2 individuals from the population. Having both halves be from the top N/2 resulted in much worse performance. Similarly, having both halves randomly chosen is also bad. It is interesting that requiring exactly half to be fit is what makes this algorithm work.