Lab 2 Genetic Algorithms

By Alejandra, Roja, and Franklin

MATH0154 Computational Statistics with Prof. 😂Gabe Chandler😂

How Genetic Algorithms Work

Genetic algorithms work by taking clues from biology on how to find solutions to certain problems. Through an iterative process of combining possible solutions, better and better solutions can be found by selecting "good" traits and ignoring "bad" traits from input solutions.

In order for a genetic algorithm to work, we must define a few entities and operations. First and foremost, an "organism" must be defined. An organism should represent a possible solution to the problem. The "fitness" of the organism should be easily computable - we should define a metric that is high for organisms close to the true solution, and low for organisms that are far away.

We then need to define a "breeding" operation: an operation that combines traits from two parent organisms to produce a child organism with traits from both.

Once this breeding operation is defined, we can select organisms that will breed based on their fitness value. That is, organisms with higher fitness will get to reproduce more often therefore will produce more children. After breeding, we eliminate a certain fraction of the least fit organisms. The children generation can then be allowed to breed with themselves and produce fitter and fitter offspring with each iteration.

Optionally, we can introduce a "mutation" operation. The purpose of this operation is to introduce characteristics into offspring that did not exist in either parent. The mutation operation occurs after breeding but before the least fit organisms are removed.

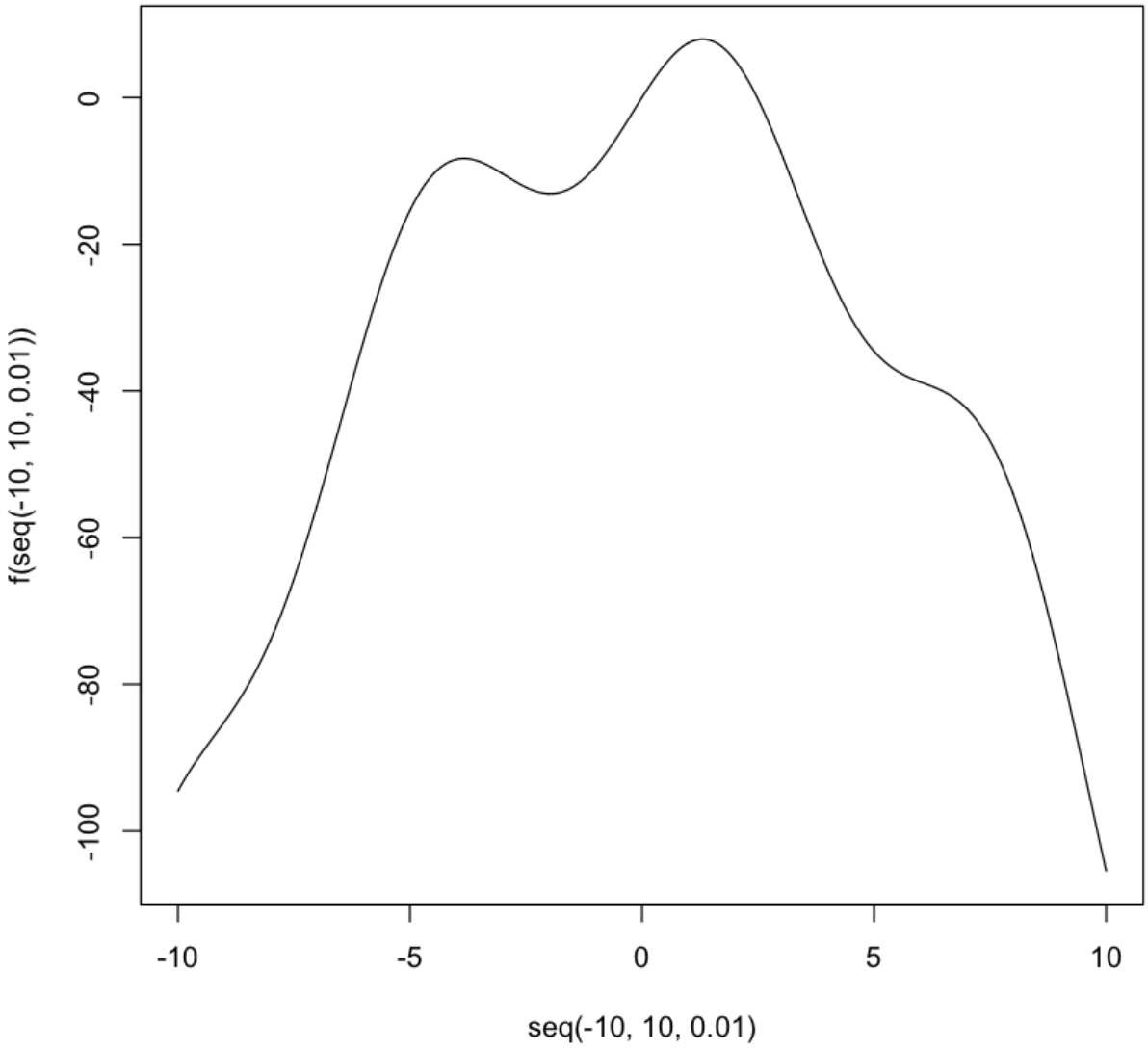
In the context of maximizing a function, we can define organisms to be x-values which may (or may not) be the global maximum of the function. The "fitness" of each organism will be computed by evaluating the function at the organism point. We define a breeding operation to be taking two organisms, and picking a random uniform between those two organisms. This allows for the creation of multiple unique children from the same two parents. We define a mutation operation to be adding a random uniform between some $[-\epsilon, \epsilon]$ to the organism. This will allow for children that are not necessarily between the two parents on the number line. In the following code blocks, we implement this algorithm.

We will first define a function with many local maxima, but only one global maximum. This is the function that we will find the maximum of using a genetic algorithm.

In [276]:
```
f <- function(x) {
    #
    # an example function that we hope to find the maximum of
    #
    # Args:
    #    x: value to evaluate the function at
    # Returns:
    #    y: f(x)

  return(-x^{2} + 10*sin(x))
}

plot(seq(-10,10, 0.01),f(seq(-10,10, 0.01)), type = 'l')
```



In these next cells we will code the functions that are the building blocks of the genetic algorithm.

In [2]:
```
Gen0 <- function(n, locale) {
    #
    # a function to generate the 0th generation in a genetic algorithm
    #
    # Args:
    #    n: number of members per generation
    #    locale: a vector where the first value is the left bound of t
he generati
    #    and the second value is the right bound of the generation
    # Returns:
    #    Gen0: a vector of length n which returns the first generation

  return(runif(n, locale[1],locale[2]))
}
```

In [263]:
```
BreedProb <- function(orgs, f) {
    #
    # computes breeding probabilities when given a list of organisms a
nd a function
    #
    # Args:
    #    orgs: a vector of organisms
    #    f: the function on which to evaluate the breeding probabiliti
es
    # Returns:
    #    output: a dataframe containing the input column (orgs) and a
column named p which ar
    #    the breeding probabilities.

  p <- exp(f(orgs))/sum(exp(f(orgs))) #probability of breeding array,
length n organisms
  df <- data.frame(orgs,p) #make a dataframe out of the vectors
  output <- df[order(df$p),] # orders by probability, ascending
  return(output) # we want to return organisms ordered by probability
of reproduction
}

Breed <- function(bp, n, k) { # functionally this has stayed the same
    #
    # return the next generation in a genetic algorithm, given a dataf
rame of breeding probabilties.
    #
    # Args:
    #    bp: a dataframe of breeding probabilities as returned by Bree
dProb
    #    n: number of separate breeding interactions desired
    #    k: number of children per breeding interaction
    # Returns:
    #    gen1: a vector of length n * k of possible entries for the ne
```

```
xt generation

    parents <- list() #initialize a list of possible parent pairs
    babies <- list() #initialize a list to hold baby generations.

    for (i in (1:n)){ #pick two parents, n times
        parents[[i]] <- sort(sample(bp$orgs, size = 2, replace = FALSE
,  prob = bp$p))
    }

    for (j in (1:length(parents))) { # for each pair of parents, perfo
rm the breeding operation K times
        babies[[j]] <- runif(k, parents[[j]][1],parents[[j]][2])
    }

    return(unlist(babies))
    }

Mutate <- function(new_gen, eps, pct) {
    #
    # induce mutation in a percentage of the input generation
    #
    # Args:
    #    new_gen: a vector of input organisms to be mutated
    #    eps: the maximum magnitude of the mutation
    #    pct: the percentage of the generation to be mutated
    # Returns:
    #    new_gen: the new generation with mutations in place


    n <- as.integer((pct/100.0)*length(new_gen))

    mutated_indices <- sample((1:length(new_gen)), n)
    mutated <- new_gen[mutated_indices] + runif(n, -eps, eps)
    new_gen[mutated_indices] <- mutated

    return(new_gen)

    }

Select <- function(bp, cutoff) {
    #
    # a function to select only the fittest individuals.
    #
    # Args:
    #    bp: dataframe of organisms and breeding probabilities
    #    cutoff: the number of organisms that will survive
    # Returns:
    #    survived: vector of survived organisms with length cutoff.
```

```
    return(tail(bp, cutoff)$orgs)
}
```

In the next cell, we have constructed these functions in a while loop, which will run the genetic algorithm until a certain stopping criterion has been reached. The stopping criterion that we have selected is that the intragenerational standard deviation. If all organisms in one generation have a standard deviation that is less than a certain tolerance, then we will stop the algorithm because we have a set of solutions that is good enough for our needs.

In [282]:
```
generations <- list(Gen0(10, c(-10,10))) #intialize the first generati
on as the first entry in a list of generations

std <- sd(generations[[1]]) # the starting standard deviation
tol <- 0.001 #the tolerance for stopping the algorithm
i <- 2 #set the starting i to initialize the loop

while (std > tol) { # if the intragenerational standard deviation less
than a certain tolerance, stop
    bp <- BreedProb(generations[[i-1]], f) #compute initial breeding p
robabilities
    new_gen <- Breed(bp,10,10) #perform the breeding operation, return
ing 100 children
    mutated <- Mutate(new_gen, std, 10) #perform a mutation algorithm
on 10% of the sample

    new_bp <- BreedProb(mutated, f) #compute breeding probabilities fo
r children
    generations[[i]] <- Select(new_bp,10) #append the 10 fittest organ
isms in the children to the new generation
    std <- sd(generations[[i]]) #compute the standard deviation of the
10 new organisms
    i <- i+1 #add one to the index
}

gen_vector <- unlist(generations) #turn the list of all organisms into
a vector
final_xy <- data.frame(gen_vector, f(gen_vector)) #evaluate the functi
on for each organism
sol <- tail(final_xy[order(final_xy$f.gen_vector),],1)$gen_vector #pic
k the fittest organism in al generations
```

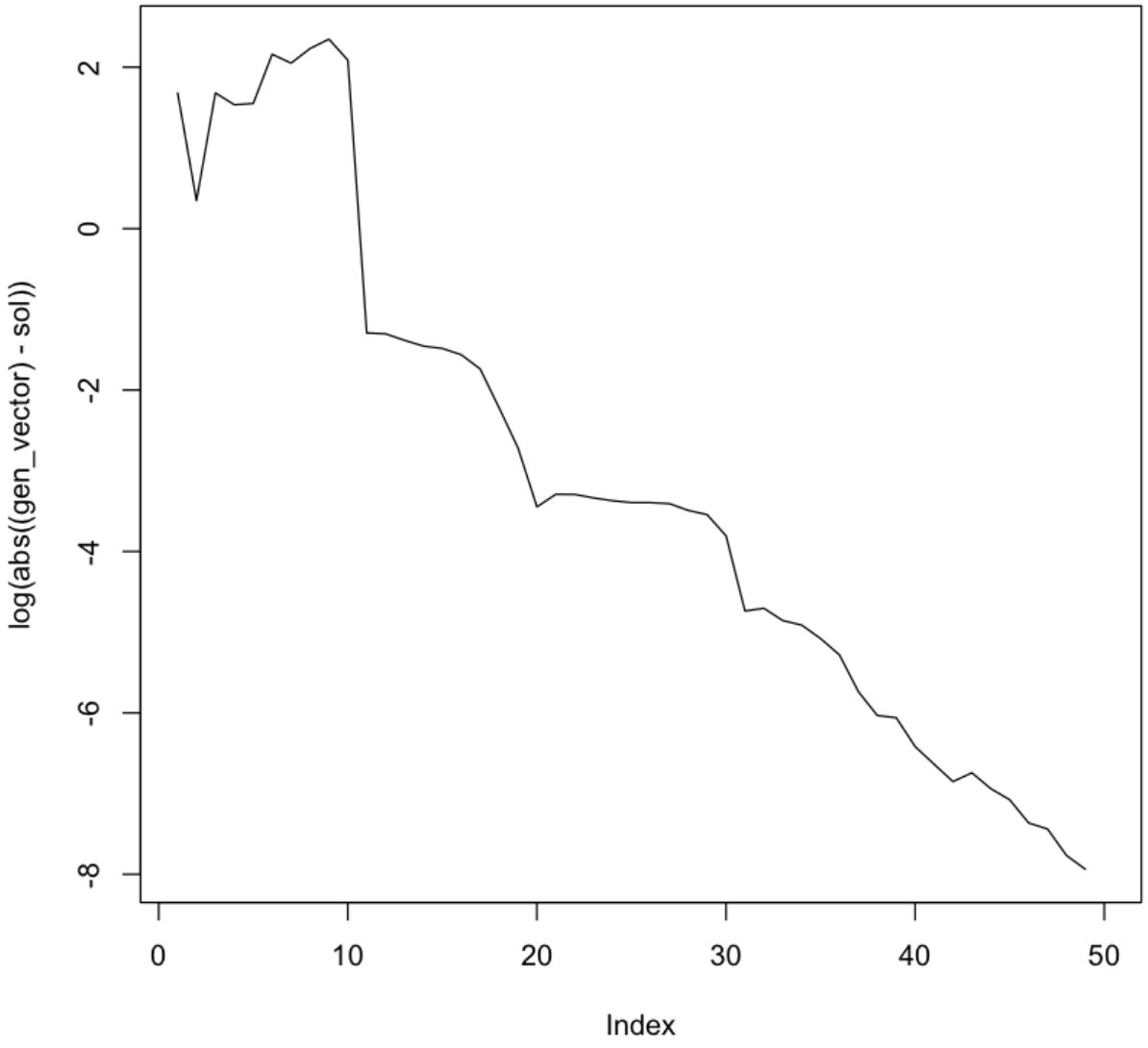
We will print the list of the organisms in each generation. It looks like it took five generations to acheive convergence within our tolerance of 0.001.

In [287]:
```
generations
```

1. 6.67555944994092 2.72011043969542 -4.06742759048939 -3.33571522962302 6.02237987332046 -7.3685416765511 -6.46771067287773 -8.00046515651047 -9.15822262410074 9.37503722030669
2. 1.58083662706359 1.03460446246817 1.05570946890243 1.07302918182817 1.07962394441913 1.51594080526871 1.48195259360605 1.1976991383714 1.37233086128421 1.33811339874677
3. 1.34347791874797 1.3434151440923 1.34183924442567 1.34063042823777 1.27270601157551 1.33983496791323 1.27313582772249 1.3367699020574 1.33517586095657 1.28404746494368
4. 1.29752096918564 1.31534503060952 1.29850254367821 1.29892344252189 1.30005868165666 1.31136278098705 1.30950362885675 1.30388252256196 1.3086163330414 1.30791576920926
5. 1.30497077926303 1.30522443197213 1.30746288436489 1.30725213540818 1.30712750803471 1.30691712383577 1.30687110792766 1.30670688993766 1.30663998439333 1.30628299233237

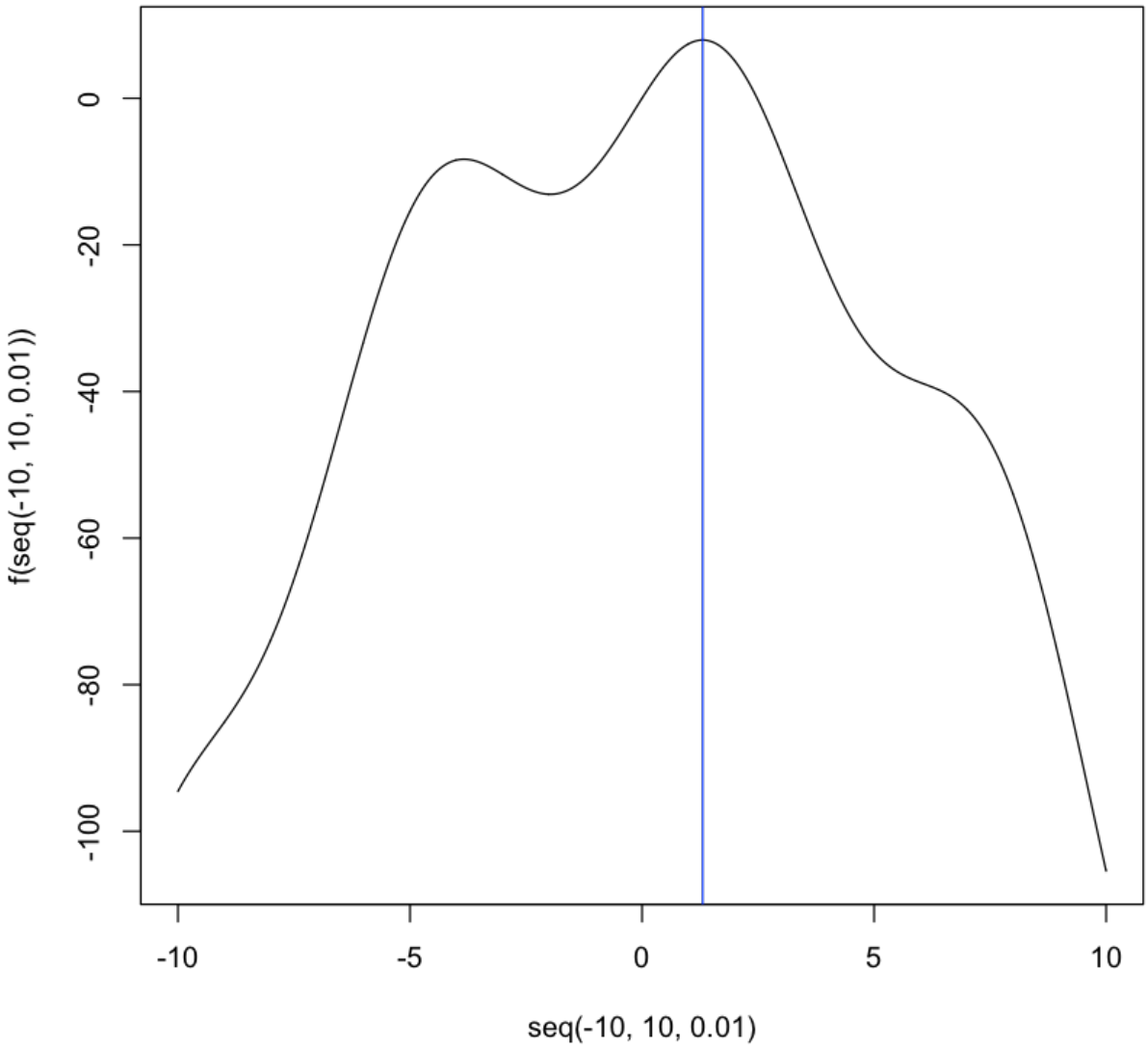
We can see how fast the algorithm converges by plotting the difference between each organism and the best organism, as a function of iteration. In the following log plot, a new generation occurs every 10 organisms.

In [289]:
```
plot(log(abs((gen_vector) - sol)), type = 'l')
```



By plotting the value of the best solution found by the genetic algorithm and the original function we can verify that we have found a global maximum.

In [286]:
```
print(sol)
plot(seq(-10,10, 0.01),f(seq(-10,10, 0.01)), type = 'l')
abline(v = sol, col = 'blue')
```



Looks good! 👏🙌🍻