Exam 2018-03-19

Maximilian Pfundstein

2019-03-11

Contents

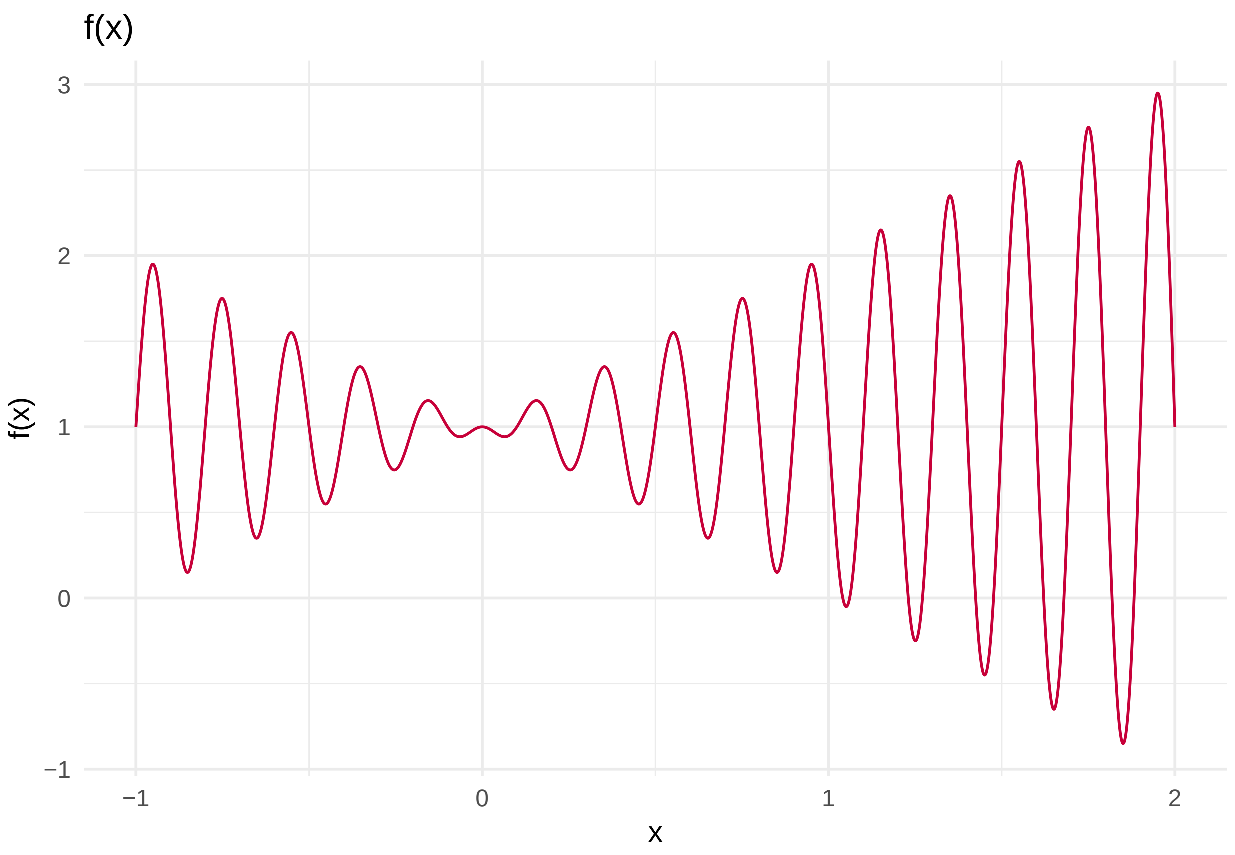
1 Assignment 1

2 Assignment 2

```
################################################################################
# Assignment 2
################################################################################

f = function(x) {
  return(-x * sin(10 * pi * x) + 1)
}
```

The function to analyse looks like this.



2.1 Question 2.1 (2 points)

The helper functions look like this.

```
##################################################################################
# Assignment 2.1
##################################################################################

to_base_10 = function(v) {
  return(Reduce(function(s,r) {s*2+r}, v))
}

transform_to_interval = function(u, a = -1, b = 2, m = 15) {
  return(a + u * ((b - a)/(2^m - 1)))
}

chrom_to_x = function(chromosome) {
  return(transform_to_interval(to_base_10(chromosome)))
}

chrom_to_y = function(chromosome) {
  return(f(transform_to_interval(to_base_10(chromosome))))
}
```

2.2 Question 2.2 (3 points)

```r
################################################################################
# Assignment 2.2
################################################################################

tournament_selection = function(population) {

  # Select fighting partners / groups
  fighters = sample(1:nrow(population), 4)

  # Get their values
  fighter_A = population[fighters[1],]
  fighter_B = population[fighters[2],]
  fighter_C = population[fighters[3],]
  fighter_D = population[fighters[4],]

  # Let them fight, group A
  if (chrom_to_y(fighter_A[2:length(fighter_A)]) <
      chrom_to_y(fighter_A[2:length(fighter_A)])) {
    winner_group_A = fighter_A
    looser_group_A = fighter_B
  }
  else {
    winner_group_A = fighter_B
    looser_group_A = fighter_A
  }

  # Let them fight, group B
  if (chrom_to_y(fighter_A[2:length(fighter_C)]) <
      chrom_to_y(fighter_A[2:length(fighter_D)])) {
    winner_group_B = fighter_C
    looser_group_B = fighter_D
  }
  else {
    winner_group_B = fighter_D
    looser_group_B = fighter_C
  }

  # Return them, first two are winners, last two are looser
  results = data.frame()
  results = rbind(results, winner_group_A)
  results = rbind(results, winner_group_B)
  results = rbind(results, looser_group_A)
  results = rbind(results, looser_group_B)

  return(results)
}
```

Question: Sometimes one varies this tournament selection and the best individual is chosen with a certain probability lesser than 1. Can you provide a motivation for this?

Answer: If we have some knowledge about the function, that it's mostly monotonic, it might sense to put more emphasis on the best individial. The tournament selection might work better if a lot of individuals get stuck, like here, in local minima. Thus in this case the best individial (with a probability lesser than 1) might work better.

2.3 Question 2.3 (5 points)

```
################################################################################
# Assignment 2.3
################################################################################

# Generate a population of size n with m digits as genes
generate_population = function(n, m = 15) {

  population = data.frame()

   for (i in 1:n) {
     individual = c(i, ifelse(round(rnorm(m, mean = 0, sd = 1)) > 0, 1, 0))
     population = rbind(population, individual)
   }

  colnames(population) = c("index", as.character(seq(from = 1, to = m, by = 1)))
  return(population)
}

# Does a crossover where the children take one half of each parent
crossover = function(chrom_A, chrom_B) {
  chrom_A = unlist(chrom_A)
  chrom_B = unlist(chrom_B)

  cut = round(length(chrom_A)/2)

  child_A = c(chrom_A[2:cut], chrom_B[(cut+1):length(chrom_B)])
  child_B = c(chrom_B[2:cut], chrom_A[(cut+1):length(chrom_B)])

  return(list(child_A, child_B))
}

# Mutates a single bit based on prob
mutate_with_prob = function(x, prob) {
  if (runif(n = 1) < prob) {
    return(ifelse(x == 0, 1, 0))
  }
  else {
    return(x)
  }
}

# Mutates on average prob genes, each gene has a prob of prob/length to mutate
mutate = function(chrom, prob = 0.05) {
  chrom = unlist(chrom)
  # To get an average of 0.05 bits mutated
  digit_prob = prob / length(chrom)
  chrom[2:length(chrom)] = sapply(chrom[2:length(chrom)], mutate_with_prob, prob)
  return(chrom)
}
```

```
# Helper function to get the best individial from a population
get_best_individual = function(population) {
  index_best = which.min(apply(population[,2:ncol(population)], 1, chrom_to_y))
  return(population[index_best,])
}

# Use all pre defined methods to actualy implement the genetic algorithm
genetic_algorithm = function(population_size = 55, mutprob = 0.05, runs = 100) {

  # Generate population
  population = generate_population(population_size)

  best_individual = get_best_individual(population)

  best_individual_iteration = 0

  for (i in 1:runs) {

    # Selection
    fighters = tournament_selection(population)

    # Create children of winners
    children = crossover(fighters[1,], fighters[2,])

    # Mutate the new individuals
    children[[1]] = mutate(children[1], prob = mutprob)
    children[[2]] = mutate(children[2], prob = mutprob)

    # Replace the loosers with the new winners
    population[fighters[3,1], 2:ncol(population)] = children[[1]]
    population[fighters[4,1], 2:ncol(population)] = children[[2]]

    # Save the overall best individual
    current_best = get_best_individual(population)
    if (chrom_to_y(current_best[2:length(current_best)]) <
        chrom_to_y(best_individual[2:length(best_individual)])) {
      best_individual = current_best
      best_individual_iteration = i
    }
  }

  return(list(best = best_individual, population = population,
              iteration = best_individual_iteration))
}
```

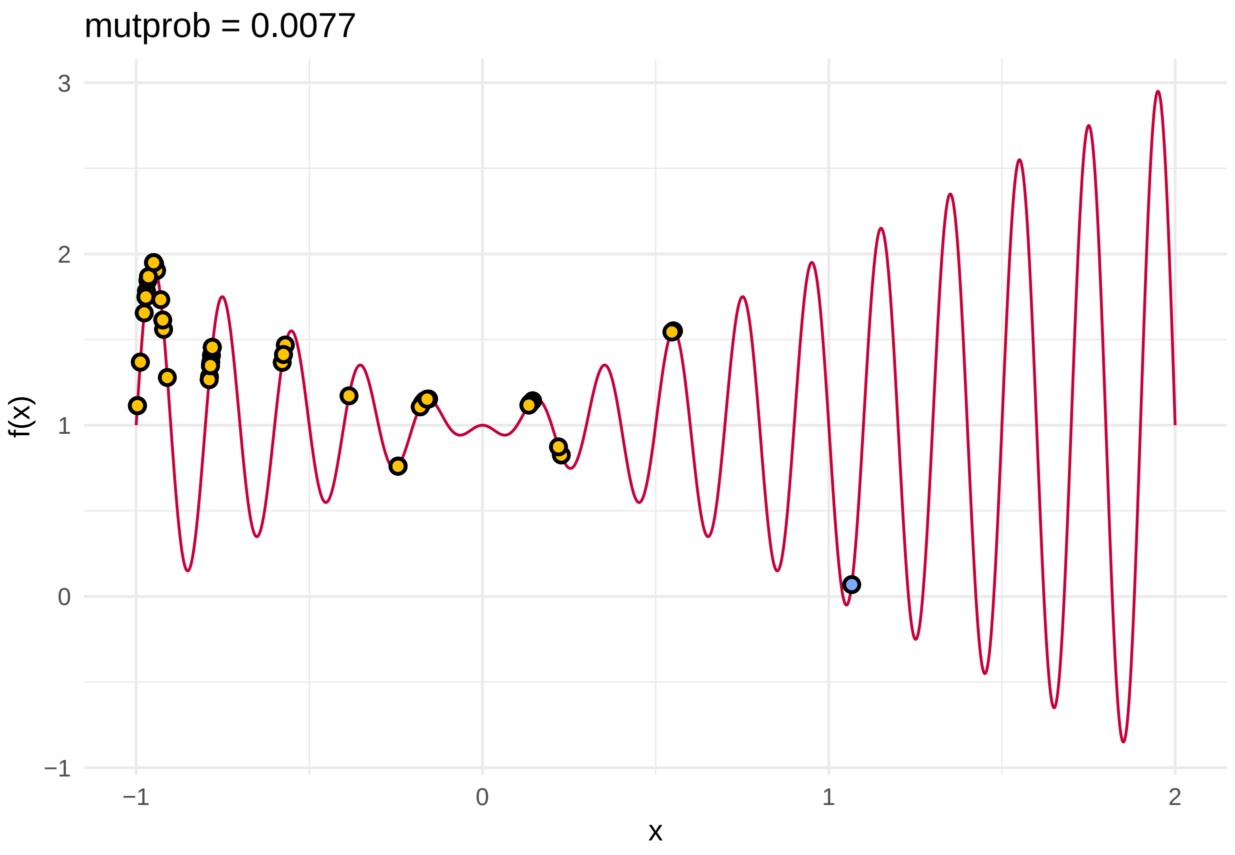
The example call for `mutprob = 0.0077` looks like this.

```
results0077 = genetic_algorithm(population_size = 55, mutprob = 0.0077, runs = 100)
print(results0077$best)
```

```
##    index 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
## 14    14 1 0 1 1 0 0 0 0 0  1  0  1  0  0  0
```

```
print(results0077$iteration)
```

```
## [1] 2
```



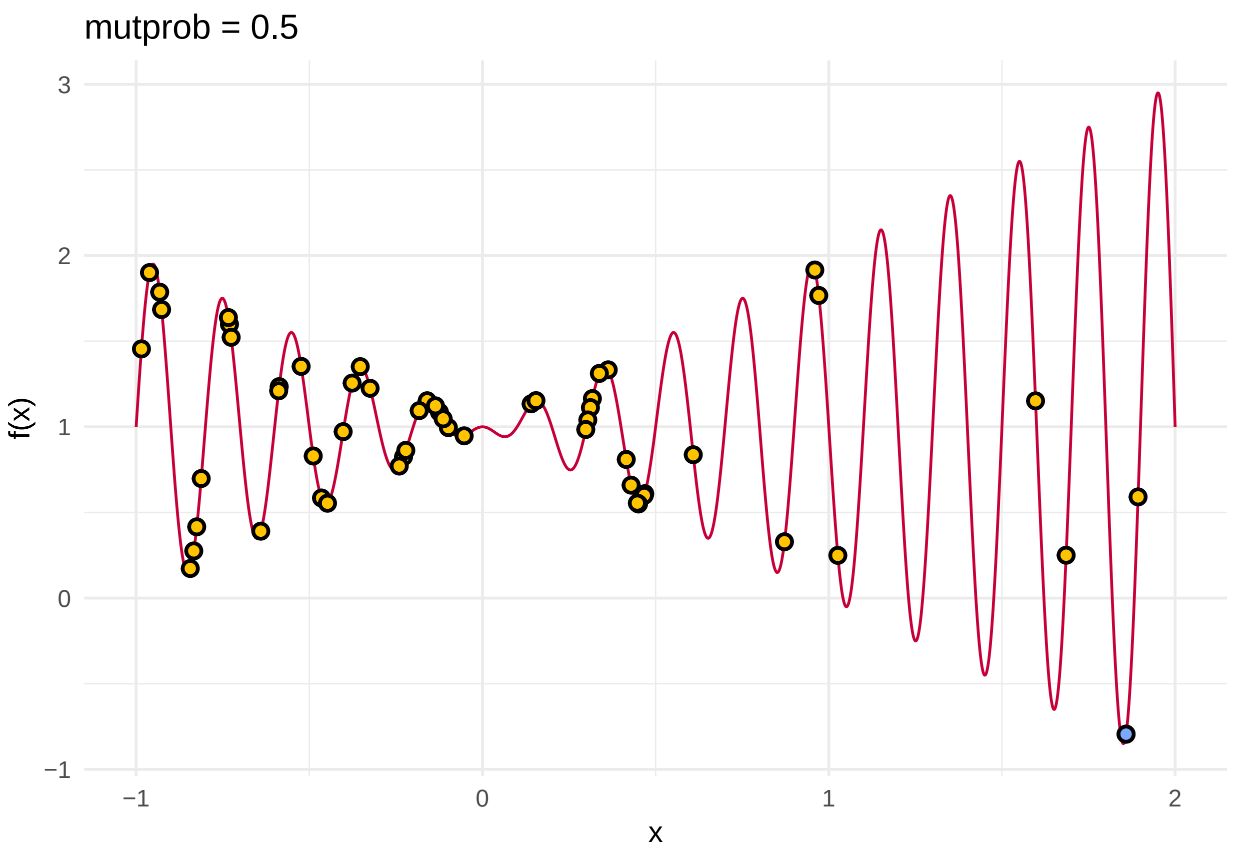
The example call for `mutprob = 0.5` looks like this.

```
results05 = genetic_algorithm(population_size = 55, mutprob = 0.5, runs = 100)
print(results05$best)
```

```
##    index 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
## 32    32 1 1 1 1 0 0 1 1 1  1  1  0  1  0  0
```

```
print(results05$iteration)
```

```
## [1] 61
```



Question: At which generation was the best value found?

Answer: The best value for `mutprob = 0.077` was found in iteration 2 and the best value for `mutprob = 0.5` was found in iteration 61.

Question: Was the minimum found?

Answer: For `mutprob = 0.077` no, for `mutprob = 0.5` yes. If we coun't in the error the encoding does, then `mutprob = 0.5` also did not find the minimum, but it found (probably) the best one the encoding of 15 bits could do.

Question: Can you explain the observed behaviour, especially when taking into account the mutation probability?

Answer: The mutation probability increases the randomness, which means how the space is being explored. A really high mutation makes the discovery mostly random, while a small one relies more on the characteristics of the parents. Here the function has a lot of *traps* which means relying on the parents is misleading. This means a higher mutation probability suits better for the problem we have here, which could change for other functions. A good way would be having a high mutation rate in the beginning of the algorithm ans slowly dicreasing on towards the end of the iterations, thus changed from global search to local search.

3 Source Code

```
knitr::opts_chunk$set(echo = TRUE)
library(ggplot2)
set.seed(42)
```

```
################################################################################
# Assignment 2
################################################################################

f = function(x) {
  return(-x * sin(10 * pi * x) + 1)
}
sequence = seq(from = -1, to = 2, by = 0.001)
f.sequence = f(sequence)

df = data.frame(sequence, f.sequence)

ggplot(df) + geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  labs(title = "f(x)", y = "f(x)", x = "x") + theme_minimal()

################################################################################
# Assignment 2.1
################################################################################

to_base_10 = function(v) {
  return(Reduce(function(s,r) {s*2+r}, v))
}

transform_to_interval = function(u, a = -1, b = 2, m = 15) {
  return(a + u * ((b - a)/(2^m - 1)))
}

chrom_to_x = function(chromosome) {
  return(transform_to_interval(to_base_10(chromosome)))
}

chrom_to_y = function(chromosome) {
  return(f(transform_to_interval(to_base_10(chromosome))))
}


################################################################################
# Assignment 2.2
################################################################################

tournament_selection = function(population) {

  # Select fighting partners / groups
  fighters = sample(1:nrow(population), 4)

  # Get their values
  fighter_A = population[fighters[1],]
  fighter_B = population[fighters[2],]
  fighter_C = population[fighters[3],]
  fighter_D = population[fighters[4],]

  # Let them fight, group A
  if (chrom_to_y(fighter_A[2:length(fighter_A)]) <
      chrom_to_y(fighter_A[2:length(fighter_A)])) {
    winner_group_A = fighter_A
    looser_group_A = fighter_B
  }
  else {
    winner_group_A = fighter_B
    looser_group_A = fighter_A
  }

  # Let them fight, group B
  if (chrom_to_y(fighter_A[2:length(fighter_C)]) <
      chrom_to_y(fighter_A[2:length(fighter_D)])) {
    winner_group_B = fighter_C
    looser_group_B = fighter_D
  }
  else {
    winner_group_B = fighter_D
    looser_group_B = fighter_C
  }

  # Return them, first two are winners, last two are looser
  results = data.frame()
  results = rbind(results, winner_group_A)
  results = rbind(results, winner_group_B)
  results = rbind(results, looser_group_A)
  results = rbind(results, looser_group_B)

  return(results)
}


################################################################################
# Assignment 2.3
################################################################################

# Generate a population of size n with m digits as genes
generate_population = function(n, m = 15) {

  population = data.frame()

   for (i in 1:n) {
     individual = c(i, ifelse(round(rnorm(m, mean = 0, sd = 1)) > 0, 1, 0))
     population = rbind(population, individual)
   }

  colnames(population) = c("index", as.character(seq(from = 1, to = m, by = 1)))
  return(population)
}

# Does a crossover where the children take one half of each parent
crossover = function(chrom_A, chrom_B) {
  chrom_A = unlist(chrom_A)
  chrom_B = unlist(chrom_B)
```

```r
  cut = round(length(chrom_A)/2)

  child_A = c(chrom_A[2:cut], chrom_B[(cut+1):length(chrom_B)])
  child_B = c(chrom_B[2:cut], chrom_A[(cut+1):length(chrom_B)])

  return(list(child_A, child_B))
}

# Mutates a single bit based on prob
mutate_with_prob = function(x, prob) {
  if (runif(n = 1) < prob) {
    return(ifelse(x == 0, 1, 0))
  }
  else {
    return(x)
  }
}

# Mutates on average prob genes, each gene has a prob of prob/length to mutate
mutate = function(chrom, prob = 0.05) {
  chrom = unlist(chrom)
  # To get an average of 0.05 bits mutated
  digit_prob = prob / length(chrom)
  chrom[2:length(chrom)] = sapply(chrom[2:length(chrom)], mutate_with_prob, prob)
  return(chrom)
}


# Helper function to get the best individial from a population
get_best_individual = function(population) {
  index_best = which.min(apply(population[,2:ncol(population)], 1, chrom_to_y))
  return(population[index_best,])
}

# Use all pre defined methods to actualy implement the genetic algorithm
genetic_algorithm = function(population_size = 55, mutprob = 0.05, runs = 100) {

  # Generate population
  population = generate_population(population_size)

  best_individual = get_best_individual(population)

  best_individual_iteration = 0

  for (i in 1:runs) {

    # Selection
    fighters = tournament_selection(population)

    # Create children of winners
    children = crossover(fighters[1,], fighters[2,])

    # Mutate the new individuals
    children[[1]] = mutate(children[1], prob = mutprob)
    children[[2]] = mutate(children[2], prob = mutprob)

    # Replace the loosers with the new winners
    population[fighters[3,1], 2:ncol(population)] = children[[1]]
    population[fighters[4,1], 2:ncol(population)] = children[[2]]

    # Save the overall best individual
    current_best = get_best_individual(population)
    if (chrom_to_y(current_best[2:length(current_best)]) <
        chrom_to_y(best_individual[2:length(best_individual)])) {
      best_individual = current_best
      best_individual_iteration = i
    }
  }

  return(list(best = best_individual, population = population,
              iteration = best_individual_iteration))
}


results0077 = genetic_algorithm(population_size = 55, mutprob = 0.0077, runs = 100)
print(results0077$best)
print(results0077$iteration)


X = apply(results0077$population[,2:ncol(results0077$population)], 1, chrom_to_x)
Y = f(X)

X_best = chrom_to_x(results0077$best[2:length(results0077$best)])
Y_best = f(X_best)

population = data.frame(X, Y)
best = data.frame(X_best, Y_best)

ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = population$X, y = population$Y),
             data = population, color = "black",fill = "#FFC300", shape = 21,
             size = 2, stroke = 1) +
  geom_point(aes(x = best$X, y = best$Y), data = best, color = "black",
             fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "mutprob = 0.0077", y = "f(x)", x = "x") +
  theme_minimal()

results05 = genetic_algorithm(population_size = 55, mutprob = 0.5, runs = 100)
print(results05$best)
print(results05$iteration)


X = apply(results05$population[,2:ncol(results05$population)], 1, chrom_to_x)
Y = f(X)
```

```r
X_best = chrom_to_x(results05$best[2:length(results05$best)])
Y_best = f(X_best)

population = data.frame(X, Y)
best = data.frame(X_best, Y_best)

ggplot(df) +
  geom_line(aes(x = sequence, y = f.sequence), color = "#C70039") +
  geom_point(aes(x = population$X, y = population$Y),
             data = population, color = "black",fill = "#FFC300", shape = 21,
             size = 2, stroke = 1) +
  geom_point(aes(x = best$X, y = best$Y), data = best, color = "black",
             fill = "#7BA9FF", shape = 21, size = 2, stroke = 1) +
  labs(title = "mutprob = 0.5", y = "f(x)", x = "x") +
  theme_minimal()
```