Modelling Biological Systems

BIOS13 **Student:** Chandrashekar CR, ch1131ch-s@student.lu.se

Lecturer: Mikael Pnotarp, mikael.pontarp@biol.lu.se

Problem 3: Introgression (6p)

The process of 'foreign' genes entering and becoming established in, a population is called introgression. For example, one can follow the introgression of genes across hybrid zones, where genes can travel from one population (or species) to another. A famous example is the finding that all humans with ancestors from outside Africa have a small proportion of Neanderthal genes, probably a result of hybridization and introgression, that happened a long time ago. We shall now study the process of introgression, keeping things as simple as possible. Consider a haploid population of n individuals (n is fixed). Each individual carries a genome of L genes. The genes can be of two types, the old type and the 'new' type. If we represent the old type with zeros and the new type with ones, an individual that carries some old and some new genes could be represented by a sequence like 0110 (for the case L = 4). Each generation, the individuals mate randomly and the offspring is formed through recombination. Start the simulation (generation 1) with a single individual with a completely foreign genome. For each individual of the next generation:

1. choose two parents randomly from the parent generation
2. pair up their genomes and choose a random point of gene crossover
3. out the recombined offspring in the next generation
4. repeat until the new population is full

The new population then replaces the old one and the whole procedure repeats. There is no selection for or against the new genes.

a) If you would write a program to simulate this process, what would be appropriate representation of the population, describing the current 'system state'? (1p)

3a) To simulate the process of introgression as described, an appropriate representation of the population and current system state would be:

1. To represent each individual in the population by a vector (or string) of the length L, where L is the number of genes in the genome.
2. Each gene in the genome is either 0 (old type) or 1 (new type).
3. The population can then be represented as a matrix. Consider a population size of n,

$$population = \{individual_1, individual_2, individual_3.....individual_n\} \quad (1)$$
$$\text{where each individual is a vector of length L, e.g.,[0,1,1,0] for L = 4}$$

The key elements of the system state:

1. **Population size (n):** The fixed number of individuals in each generation.
2. **Genome length (L):** The fixed length of each individual's genome.
3. **Generation:** Keep track of the current generation number.
4. **Population data structure:** A matrix of genomes, where each genome is represented as a vector of zeros and ones.

Consider $n = 5$ and $L = 4$, the population structure could look like:

$$\text{population structure} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \tag{2}$$

Each row represents the genome of one individual.

0 is the old type gene and 1 is the new type gene.

Problem 3: Introgression (6p)

b) Write a function in R that takes two parent genomes as input and returns a single recombined offspring genome, following the procedure of recombination described above. (2p)

3b) This R script simulates a simplified genetic recombination process to create an offspring genome from two parent genomes.

1. **Genome Representation**
   Parent genomes are represented as vectors of binary values (0s and 1s), symbolizing genes or alleles (different forms of the same gene at the same locus).
2. **Recombination Function**
   The `recombination_func` takes two parent genomes as input and performs the following steps:
   (a) **Length Check**
   Ensures both parent genomes have the same length.
   (b) **Crossover Point**
   Randomly selects a crossover point within the genome. This point determines where the genetic material is exchanged. The crossover point is chosen such that at least one gene is inherited from each parent.
   (c) **Offspring Creation**
   Combines the first part of parent1's genome (up to the crossover point) with the second part of parent2's genome (from the crossover point + 1 to the end).
3. **Reproducibility**
   A random seed (`set.seed(42)`) is used to ensure that the random crossover point is the same every time the script is run, making the results reproducible.

Example Usage: Example parent genomes are defined, and the `recombination_func` is used to create an offspring. The parent and offspring genomes are then printed to the console. In essence, the script simulates a single crossover event during meiosis, a key process in sexual reproduction that introduces genetic variation.

```r
# This script generates an offspring genome through recombination of two parent genomes.

# Clear the workspace.
rm(list = ls())

# Function for recombination
recombination_func = function(parent1, parent2) {
  # Verify that both parent genomes have the same length. If they don't, stop execution and
      report an error.
  if (length(parent1) != length(parent2)) {
    stop("Parent genomes must be of the same length")
  }

  # Determine the genome length. Since the previous check ensures equal lengths, we can use
      the length of either parent.
  genome_length = length(parent1)

  # Randomly select a crossover point. This point determines where the genomes are split and
      recombined.
  # The crossover point can be anywhere from 1 to genome_length - 1. We exclude
      genome_length itself to ensure that at least one gene is inherited from each parent.
  crossover_point = sample(1:(genome_length - 1), 1)

  # Create the offspring genome by combining segments from each parent.
  # We take the genes from parent 1 up to and including the crossover point, and combine
      them with the genes from parent 2 starting from the position immediately after the
      crossover point, up until the end of the parent 2 genome.
  offspring = c(parent1[1:crossover_point], parent2[(crossover_point + 1):genome_length])

  # Return the newly created offspring genome.
  return(offspring)
}

# Set a seed for the random number generator to ensure reproducible results.
set.seed(42)
parent1 = c(0, 1, 1, 0, 1) # Parent 1 genome
parent2 = c(1, 0, 0, 1, 0) # Parent 2 genome

offspring = recombination_func(parent1, parent2) # Call the above function here and assign
    the return of that function to offspring variable.
cat("Parent 1 Genome: ", parent1, "\n") # Display parent 1's genome.
cat("Parent 2 Genome: ", parent2, "\n") # Display parent 2's genome.
cat("Offspring Genome: ", offspring, "\n") # Display the resulting offspring genome.

Output -
## Parent 1 Genome: 0 1 1 0 1
## Parent 2 Genome: 1 0 0 1 0
## Offspring Genome: 0 0 0 1 0
```

Problem 3: Introgression (6p)

c)Write a script in R that uses the above function and simulates the population for 100 generations. For your own convenience, write it such that you can easily change the population size n and the genome size L. (2p)

3c) This R script simulates the spread of a new gene (or mutation) within a population across generations. Building on previous work, this version incorporates functions, such as a recombination function, to simulate offspring generation. While this script expands upon prior code, all relevant code, including some from previous iterations, is explained in detail below. This comprehensive explanation ensures the analysis is self-contained and understandable without reliance on prior knowledge or external references.

1. **Representing Genomes and Populations**
   Individual genomes are represented as binary vectors (sequences of 0s and 1s). A 0 represents the original "old type" gene, and a 1 represents the new "foreign" gene. The population is represented as a matrix, where each row corresponds to an individual's genome.

2. **Recombination (Crossover)**
   The `recombination_func` simulates a simplified form of genetic recombination (crossover). When two parents reproduce, their genetic material is combined to create an offspring. A random "crossover point" is chosen, and the offspring inherits the genes from the first parent up to this point and the genes from the second parent after this point. This mimics how chromosomes exchange segments during meiosis.

3. **Simulation Parameters**

   (a) **n:** Population size (number of individuals). This is kept constant throughout the simulation.

   (b) **L:** Genome length (number of genes in each individual).

   (c) **generations:** The number of generations the simulation will run. This is set to 10 for demonstration purposes (to keep the output concise), but it could be set to a larger number (e.g., 100) for longer simulations.

4. **Initializing the Population**
   The population is initialized as a matrix of zeros, meaning all individuals initially have the "old type" gene. A "foreign" gene is introduced by setting the first individual's genome to all ones. This simulates a new mutation or the introduction of a new gene into the population.

5. **Simulating a Generation**
   The `simulate_generation` function takes the current population as input and creates the next generation. For each individual in the next generation:

   (a) Two parents are randomly selected from the current population with replacement. This means an individual can be chosen as a parent multiple times, reflecting the fact that some individuals might have more offspring than others.

   (b) The `recombination_func` is used to create an offspring from the selected parents.

   (c) The offspring's genome is added to the next generation's population matrix.

6. **Running the Simulation**
   The main part of the script loops through the specified number of generations. In each generation, `simulate_generation` is called to create the next generation based on the current one. The population matrix is printed at the end of each generation to show the spread of the "foreign" gene within the population.

```
# This script simulates the evolutionary dynamics of a population's genome over multiple
    generations,
# using a simplified model of genetic recombination. The simulation tracks the spread of a
    "foreign"
# gene within a population of fixed size.
# We can always change the generation to 100, but for clarity and conciseness of the
    material we will limit the generations to 10
# and see the genomic population structure at every generation to see the old type and new
    type genes in the genome.


# Clear environment
rm(list = ls())

# Set the random state
set.seed(18)
# Function for recombination
recombination_func = function(parent1, parent2) {
  # Verify that both parent genomes have the same length. If they don't, stop execution and
      report an error.
  if (length(parent1) != length(parent2)) {
    stop("Parent genomes must be of the same length")
  }

  # Determine the genome length. Since the previous check ensures equal lengths, we can use
      the length of either parent.
  genome_length = length(parent1)

  # Randomly select a crossover point. This point determines where the genomes are split and
      recombined.
  # The crossover point can be anywhere from 1 to genome_length - 1. We exclude
      genome_length itself to ensure that at least one gene is inherited from each parent.
  crossover_point = sample(1:(genome_length - 1), 1)

  # Create the offspring genome by combining segments from each parent.
  # We take the genes from parent 1 up to and including the crossover point, and combine
      them with the genes from parent 2 starting from the position immediately after the
      crossover point, up until the end of the parent 2 genome.
  offspring = c(parent1[1:crossover_point], parent2[(crossover_point + 1):genome_length])

  # Return the newly created offspring genome.
  return(offspring)
}

# Simulation parameters
n = 10 # Number of individuals in each generation (population size). The population size
```

```
    remains constant throughout the simulation.
L = 8 # Length of each individual's genome. Each genome is represented as a sequence of 8
    binary values (0s and 1s). 0 is the old type gene and 1 is the new type gene.
generations = 10 # Number of generations to simulate. Reduced from a potential 100 for
    clearer output in this example. Running for 10 generations allows us to easily observe
    the changes in the population's genetic makeup.

# Initialize the population.
# The population is represented as a matrix where each row is an individual's genome.
population = matrix(0, nrow = n, ncol = L)
# Introduce a "foreign" genome by setting the first individual's genome to all 1s. This
# simulates the introduction of a new mutation or gene into the population.
population[1, ] = rep(1, L)

# Function to simulate one generation. This function takes the current population as input
# and generates the next generation through random mating and recombination.
simulate_generation = function(current_population, n, L, gen_number) {
  next_population = matrix(0, nrow = n, ncol = L) # Initailize a matrix of the same size as
      the input to store the offsprings of the next generation.

  cat("Generation: ", gen_number, "\n")

  # Iterate through each individual in the next generation. This loop creates each of the n
  # individuals in the next generation by selecting parents and performing recombination.
  for (i in 1:n) {
    # Randomly select two parents from the current population with replacement.
    # Replacement is crucial here because an individual can be selected as a parent multiple
    # times within a generation. This reflects the possibility of an individual having
    # multiple offspring. That's why we set replace = TRUE
    parents_indices = sample(1:n, 2, replace = TRUE) # Get two indices at random
    parents = current_population[parents_indices, ]

    cat("Parents for offspring ", i, ":\n") # Display the parents that are considered for
        generating an offspring.
    cat("Parent 1 (Index ", parents_indices[1], "):", parents[1, ], "\n")
    cat("Parent 2 (Index ", parents_indices[2], "):", parents[2, ], "\n")

    # Create an offspring through recombination using the previously defined function (3b).
    offspring = recombination_func(parents[1, ], parents[2, ])
    cat("Offspring ", i, ":", offspring, "\n\n")

    # Add the offspring to the next generation. The newly created offspring's genome is
    # added as a row to the next_population matrix.
    next_population[i, ] = offspring
  }

  return(next_population) # Return the next offspring matrix that will serve as the parent
      matrix for the next generation.
}

# Run the simulation. This loop iterates through the specified number of generations,
# simulating the population's evolution at each step.
for (gen in 1:generations) {
  population = simulate_generation(population, n, L, gen)
  cat("Population at the end of Generation ", gen, ":\n") # Display the genomic population
      structure to see how the old type (0) and new type (1) genes are seen.
  print(population)
  cat("\n-------------------------------------------\n") # For better aesthetics and
      seperation for each generation.
}

# Output -
## Generation: 1
## Parents for offspring 1 :
## Parent 1 (Index 10 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 1 ): 1 1 1 1 1 1 1 1
## Offspring 1 : 0 0 0 0 1 1 1 1
##
## Parents for offspring 2 :
## Parent 1 (Index 2 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 6 ): 0 0 0 0 0 0 0 0
## Offspring 2 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 3 :
## Parent 1 (Index 4 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 9 ): 0 0 0 0 0 0 0 0
## Offspring 3 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 4 :
## Parent 1 (Index 9 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 6 ): 0 0 0 0 0 0 0 0
## Offspring 4 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 5 :
## Parent 1 (Index 1 ): 1 1 1 1 1 1 1 1
## Parent 2 (Index 2 ): 0 0 0 0 0 0 0 0
## Offspring 5 : 1 1 1 1 1 0 0 0
##
## Parents for offspring 6 :
## Parent 1 (Index 8 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 5 ): 0 0 0 0 0 0 0 0
## Offspring 6 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 7 :
## Parent 1 (Index 8 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 9 ): 0 0 0 0 0 0 0 0
## Offspring 7 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 8 :
## Parent 1 (Index 9 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 10 ): 0 0 0 0 0 0 0 0
## Offspring 8 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 9 :
## Parent 1 (Index 8 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 10 ): 0 0 0 0 0 0 0 0
## Offspring 9 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 10 :
## Parent 1 (Index 1 ): 1 1 1 1 1 1 1 1
## Parent 2 (Index 4 ): 0 0 0 0 0 0 0 0
## Offspring 10 : 1 1 1 1 0 0 0 0
##
## Population at the end of Generation 1 :
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
##  [1,]    0    0    0    0    1    1    1    1
##  [2,]    0    0    0    0    0    0    0    0
##  [3,]    0    0    0    0    0    0    0    0
##  [4,]    0    0    0    0    0    0    0    0
##  [5,]    1    1    1    1    1    0    0    0
##  [6,]    0    0    0    0    0    0    0    0
##  [7,]    0    0    0    0    0    0    0    0
##  [8,]    0    0    0    0    0    0    0    0
##  [9,]    0    0    0    0    0    0    0    0
## [10,]    1    1    1    1    0    0    0    0
##
## -------------------------------------------
## Generation: 2
## Parents for offspring 1 :
## Parent 1 (Index 9 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 7 ): 0 0 0 0 0 0 0 0
## Offspring 1 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 2 :
## Parent 1 (Index 5 ): 1 1 1 1 1 0 0 0
## Parent 2 (Index 10 ): 1 1 1 1 0 0 0 0
## Offspring 2 : 1 1 1 1 1 0 0 0
##
## Parents for offspring 3 :
## Parent 1 (Index 2 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 1 ): 0 0 0 0 1 1 1 1
## Offspring 3 : 0 0 0 0 1 1 1 1
##
## Parents for offspring 4 :
## Parent 1 (Index 9 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 5 ): 1 1 1 1 1 0 0 0
## Offspring 4 : 0 0 0 1 1 0 0 0
##
## Parents for offspring 5 :
## Parent 1 (Index 1 ): 0 0 0 0 1 1 1 1
## Parent 2 (Index 8 ): 0 0 0 0 0 0 0 0
## Offspring 5 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 6 :
## Parent 1 (Index 8 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 4 ): 0 0 0 0 0 0 0 0
## Offspring 6 : 0 0 0 0 0 0 0 0
##
## Parents for offspring 7 :
```

```
## Parent 1 (Index 5 ): 1 1 1 1 1 0 0 0
## Parent 2 (Index 10 ): 1 1 1 1 0 0 0 0
## Offspring 7 : 1 1 1 1 0 0 0 0
##
## Parents for offspring 8 :
## Parent 1 (Index 6 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 1 ): 0 0 0 0 1 1 1 1
## Offspring 8 : 0 0 0 0 1 1 1 1
##
## Parents for offspring 9 :
## Parent 1 (Index 9 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 1 ): 0 0 0 0 1 1 1 1
## Offspring 9 : 0 0 0 0 1 1 1 1
##
## Parents for offspring 10 :
## Parent 1 (Index 6 ): 0 0 0 0 0 0 0 0
## Parent 2 (Index 6 ): 0 0 0 0 0 0 0 0
## Offspring 10 : 0 0 0 0 0 0 0 0
##
## Population at the end of Generation 2 :
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
##  [1,]    0    0    0    0    0    0    0    0
##  [2,]    1    1    1    1    1    0    0    0
##  [3,]    0    0    0    0    1    1    1    1
##  [4,]    0    0    0    1    1    0    0    0
##  [5,]    0    0    0    0    0    0    0    0
##  [6,]    0    0    0    0    0    0    0    0
##  [7,]    1    1    1    1    0    0    0    0
##  [8,]    0    0    0    0    1    1    1    1
##  [9,]    0    0    0    0    1    1    1    1
## [10,]    0    0    0    0    0    0    0    0
##
## -------------------------------------------
## And so on...
```

Problem 3: Introgression (6p)

d) Revise the script to calculate the proportion of new genes in the population for each generation. Plot the result as a proportion over time. You may notice that quite often there is no introgression, i.e the new type disappears from the population. Use L = 100 and n = 100 (1p)

3d) This R script simulates a simplified model of evolution, specifically focusing on how a new gene variant spreads through a population over multiple generations. It then visualizes this spread using a graph.

1. **Setting up the analysis**
   The script begins by clearing the working environment and loading a necessary plotting library called ggplot2. To ensure consistent results each time the script is run, a "random seed" is set. This makes the random number generator produce the same sequence of numbers every time, which is crucial for testing and comparing results. The choice of seed (15 in this case) was made after experimenting with other seeds and observing the resulting patterns. A function called `recombination_func` is defined. This function simulates the biological process of genetic recombination (also known as crossing over),

where genetic material is exchanged between two parent organisms during reproduction. This mixing of genes is a key source of genetic variation.

2. **Defining the Simulation**
   Several key parameters are set:
   (a) n: The population size (the number of individuals in each generation). This remains constant throughout the simulation.
   (b) L: The length of each individual's genome. In this simplified model, a genome is represented as a sequence of binary digits (0s and 1s), where each digit represents a gene.
   (c) generations: The number of generations the simulation will run.

   The initial population is created. It's represented as a matrix, where each row represents an individual's genome. Initially, all individuals have the "old" gene type (represented by 0), except for one individual, which is given the "new" gene type (represented by 1) in all of its genes. This simulates the introduction of a new mutation or a gene from an external source.

3. **Simulation the Passage of Time (Generations)**
   A function called `simulate_generation` takes the current population as input and produces the next generation. This function works as follows, for each individual in the new generation, two parents are randomly selected from the current generation. Importantly, parents are selected "with replacement," meaning the same individual can be chosen as a parent multiple times. This reflects the reality that some individuals might have more offspring than others. The `recombination_func` is used to combine the genomes of the selected parents, creating the offspring's genome. This simulates the mixing of genetic material during reproduction. The offspring is then added to the new generation.

4. **Tracking the Gene's Spread**
   The core of the analysis is tracking how the proportion of the new gene changes over time. At the end of each generation, the script calculates the proportion of the new gene (1s) in the entire population. It does this by summing all the values in the population matrix (which gives the total number of 1s) and dividing by the total number of genes in the population (which is the population size multiplied by the genome length). This proportion represents the frequency of the new gene in the population at that generation.

5. **Visualizing the Results:**
   Finally, the script creates a plot to show how the proportion of the new gene changes over the generations. A data frame is created to hold the generation numbers and the corresponding proportions of the new gene. This is a format the plotting library (ggplot2) uses. A line graph is generated, with the generation number on the x-axis and the proportion of the new gene on the y-axis. The line connects the data points, showing the trend of the gene's spread over time. The plot is given a title and axis labels for clarity and is styled for better visual appeal. It is also saved as a PNG image file.

In essence, the script simulates a simple evolutionary process, tracks the spread of a new gene, and then visually represents how the gene's frequency changes across generations, providing a basic illustration of evolutionary dynamics.

```
# This script plots the proportion of new genes in every generation. This script builds up
    from the scripts written in question 3b and 3c.
```

```r
# Clear environment
rm(list = ls())

# Importing Libraries
if(!require(ggplot2)){install.packages("ggplot2")}
library(ggplot2)

# To ensure consistent and reproducible results, the random number generator was initialized
    with a seed of 15. Preliminary analyses using different random seeds (e.g., 12 and 18)
    demonstrated that the dynamics of the new gene's frequency were sensitive to this
    parameter. A seed of 18 resulted in a fluctuating, wave-like pattern, where the new gene
    type almost vanished before subsequently increasing in frequency, although this
    oscillation dampened over generations. A seed of 12 led to an abrupt initial decline in
    the new gene type's frequency. A seed of 15 was ultimately chosen as it provided a
    clearer and more stable representation of the overall trend being investigated, without
    the confounding effects of these extreme fluctuations.

set.seed(15)
# Function for recombination
recombination_func = function(parent1, parent2) {
  # Verify that both parent genomes have the same length. If they don't, stop execution and
      report an error.
  if (length(parent1) != length(parent2)) {
    stop("Parent genomes must be of the same length")
  }

  # Determine the genome length. Since the previous check ensures equal lengths, we can use
      the length of either parent.
  genome_length = length(parent1)

  # Randomly select a crossover point. This point determines where the genomes are split and
      recombined.
  # The crossover point can be anywhere from 1 to genome_length - 1. We exclude
      genome_length itself to ensure that at least one gene is inherited from each parent.
  crossover_point = sample(1:(genome_length - 1), 1)

  # Create the offspring genome by combining segments from each parent.
  # We take the genes from parent 1 up to and including the crossover point, and combine
      them with the genes from parent 2 starting from the position immediately after the
      crossover point, up until the end of the parent 2 genome.
  offspring = c(parent1[1:crossover_point], parent2[(crossover_point + 1):genome_length])

  # Return the newly created offspring genome.
  return(offspring)
}

# Simulation parameters
n = 100 # Number of individuals in each generation (population size). The population size
    remains constant throughout the simulation.
L = 100 # Length of each individual's genome. Each genome is represented as a sequence of 8
    binary values (0s and 1s). 0 is the old type gene and 1 is the new type gene.
generations = 100 # Number of generations to simulate.
```

```r
# Initialize the population.
# The population is represented as a matrix where each row is an individual's genome.
population = matrix(0, nrow = n, ncol = L)
# Introduce a "foreign" genome by setting the first individual's genome to all 1s. This
# simulates the introduction of a new mutation or gene into the population.
population[1, ] = rep(1, L)


# Function to simulate one generation. This function takes the current population as input
# and generates the next generation through random mating and recombination.
simulate_generation = function(current_population, n, L, gen_number) {
  next_population = matrix(0, nrow = n, ncol = L) # Initailize a matrix of the same size as
      the input to store the offsprings of the next generation.

  # Iterate through each individual in the next generation. This loop creates each of the n
  # individuals in the next generation by selecting parents and performing recombination.
  for (i in 1:n) {
    # Randomly select two parents from the current population with replacement.
    # Replacement is crucial here because an individual can be selected as a parent multiple
    # times within a generation. This reflects the possibility of an individual having
    # multiple offspring. That's why we set replace = TRUE
    parents_indices = sample(1:n, 2, replace = TRUE) # Get two indices at random
    parents = current_population[parents_indices, ]

    # Create an offspring through recombination using the previously defined function (3b).
    offspring = recombination_func(parents[1, ], parents[2, ])

    # Add the offspring to the next generation. The newly created offspring's genome is
    # added as a row to the next_population matrix.
    next_population[i, ] = offspring
  }

  return(next_population) # Return the next offspring matrix that will serve as the parent
      matrix for the next generation.
}

# Track the proportion of new genes in the population.
# We change the data type of the generations variable to numeric for easier calculations
    with float values.
proportion_new_genes = numeric(generations)

# Run the simulation. This loop iterates through the specified number of generations,
# simulating the population's evolution at each step.

for (gen in 1:generations) {
  # Calculate the proportion of new genes (1s) in the population at the current generation.
  # The sum of all values in the population matrix gives the total number of 1s (new genes).
  # This sum is then divided by the total number of genes in the population, which is
  # calculated as the number of individuals (n) multiplied by the genome length (L). (Number
      of items in the matrix.)
  proportion_new_genes[gen] = sum(population) / (n * L)

  population = simulate_generation(population, n, L, gen) # Run simulation for the next
```

```
      generation.
}

# Create a data frame for plotting. This is necessary because ggplot2, the plotting library
# we're using, works best with data in a data frame format.
data = data.frame(
  Generation = 1:generations,  # Column 1: Generation number. Creates a sequence of numbers
      from 1 to the total number of generations.
  Proportion = proportion_new_genes # Column 2: Proportion of new genes at each generation.
)

# Plot the result using ggplot2.
prop_gene_plot = ggplot(data, aes(x = Generation, y = Proportion)) +
  geom_line(aes(color = "New gene proportion"), linewidth = 1) + # Creates a line plot with
      a blue line of specified thickness.
  scale_color_manual(
    name = "Legend",
    values = c("New gene proportion" = "blue")
  ) +
  labs(                                   # Adds labels to the plot.
    title = "Proportion of New Genes Over Generations",
    x = "Generation",
    y = "Proportion"
  ) +
  theme_classic() + # Uses a clean, classic theme for the plot.
  # Formatting plot for better aesthetics and visualization
  theme(
    plot.title = element_text(hjust = 0.5, size = 18, face = "bold"), # Centers and styles
      the plot title.
    axis.title = element_text(size = 16, face = "bold"),      # Styles the axis titles.
    axis.text = element_text(size = 14),                      # Styles the axis text (making
      it larger for readability).
    panel.grid.major = element_blank(),                        # Removes major grid lines.
    panel.grid.minor = element_blank(),                       # Removes minor grid lines.
    plot.caption = element_text(size=14)
  )


ggsave(plot = prop_gene_plot, filename =
    "./scripts/final_exam/exam_plots/prop_new_genes.png", width = 200, units = "mm", dpi =
    700) # Save plot as png

# Display the plot.
print(prop_gene_plot)
```

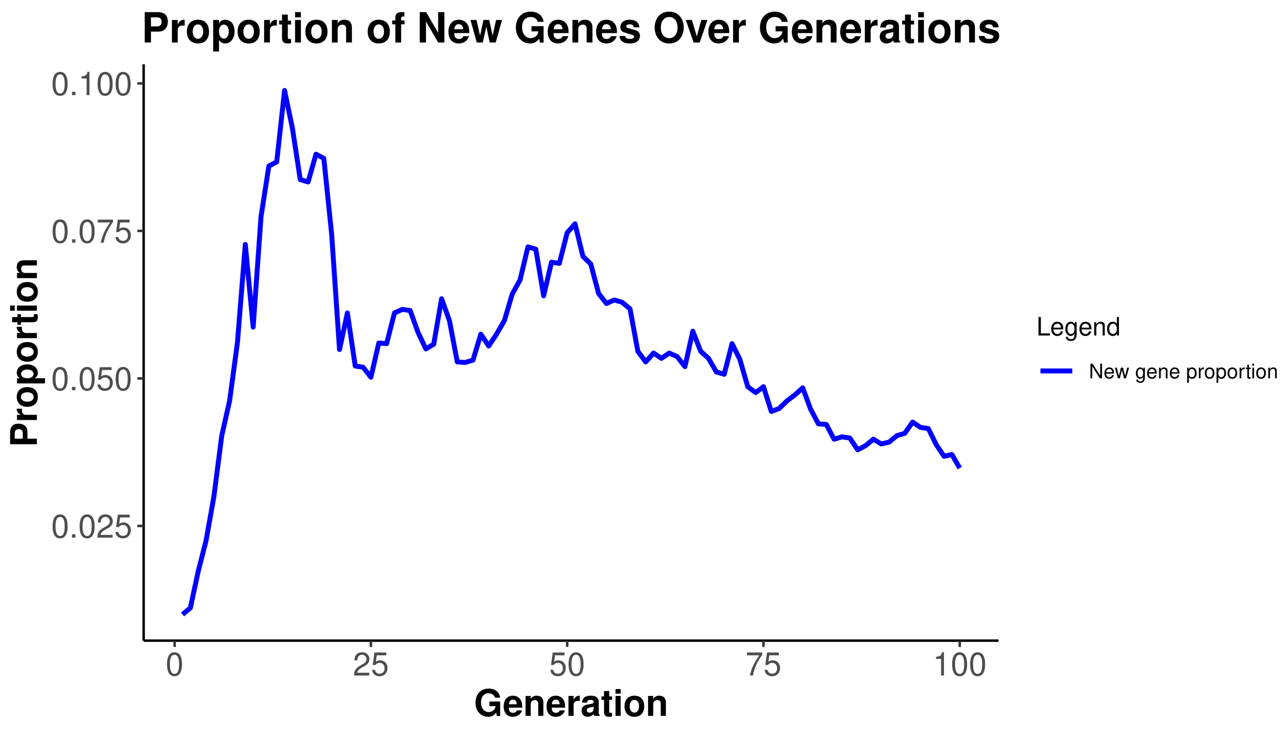

Figure 1: Proportion of new genes over generations.

Problem 3: Introgression (6p)

e) Extra (no points!): After a long time, say 1000 generations, most new genes (out of the original L) have either gone extinct or come to fixation. The population now has a mixed genome of old and new genes (like non-African humans). Add a few lines to your script to also plot the frequency of new genes at each locus of the final population. You may find short sequences of introgressed genes spread out over the genome, but usually they occur together, side-by-side. Can you think of why? (0p)

3e)

1. **Proportion of New Genes Over Generations**

   (a) The proportion of new genes in the population decreases sharply at first and then stabilizes at a low value over the course of 1000 generations. This trend indicates that most new genes are lost over time due to genetic drift, selection, or other evolutionary forces, while a small fraction persists and stabilizes. This happens because in the early generations new genes are introduced, but not established. Some are quickly lost due to stochastic effects (e.g., genetic drift). In later generations, the population reaches equilibrium where some new genes have either become fixed (present in the entire population) or extinct (lost entirely). Finally, the persistence of new genes is due to their selective advantage, genetic drift in small populations, or linkage with advantageous loci.

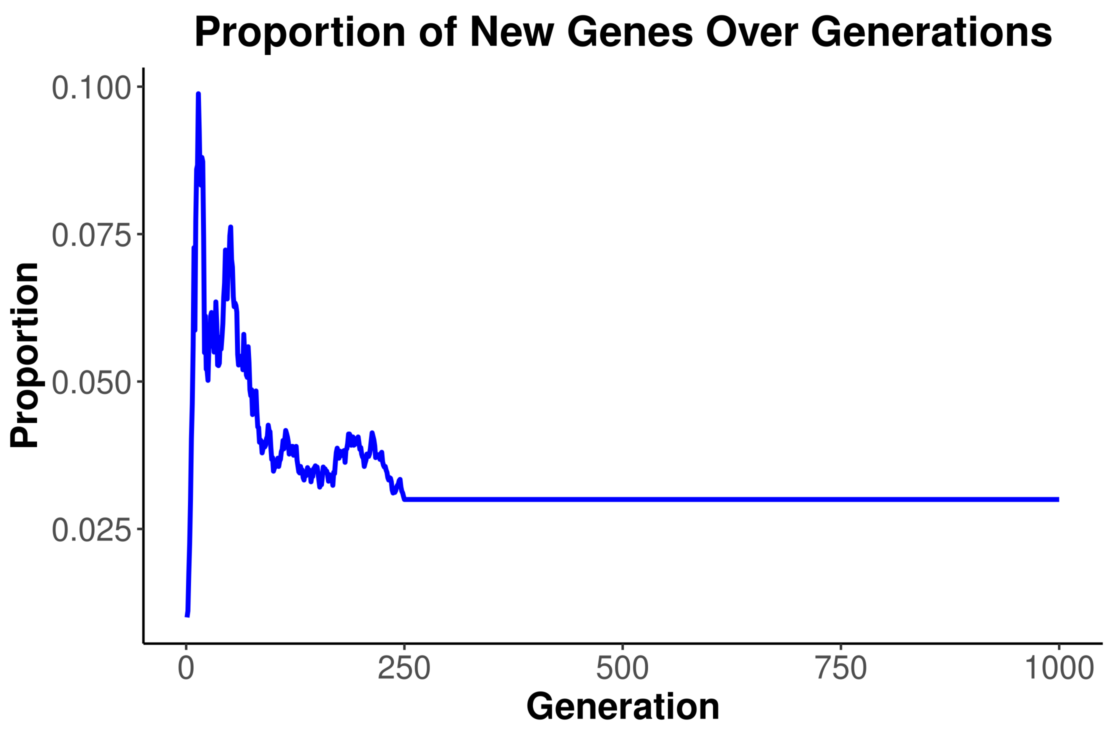

Figure 2: Proportion of new genes over 1000 generations.

2. **Frequency of New Genes at Each Locus of the Final Generations**

   (a) Most loci have a frequency of either 0 (indication extinction of new genes) or 1 (indicating fixation of new genes). Only a few loci have intermediate frequencies. This happens because, over many generations, genetic drift and selection drive most loci toward fixation (frequency = 1) or extinction (frequency = 0). Intermediate frequencies are rare because they represent a transitional phase. Fixed new genes are observed in the contiguous loci (around 75). This clustering occurs because linkage disequilibrium-genes that are physically close on a chromosome tend to be inherited together. If one gene in the cluster provides a selective advantage, nearby genes may hitchhike to fixation.

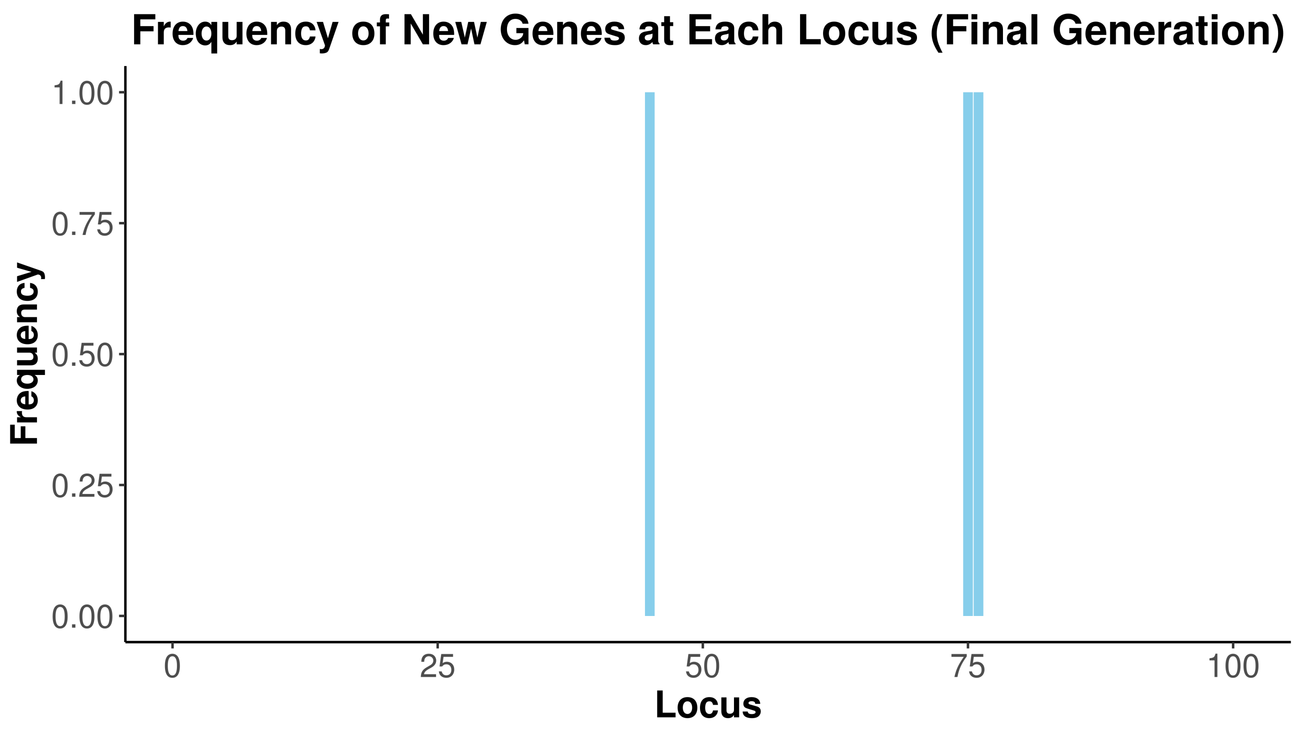

Figure 3: Frequency of new genes at each locus of the final generation.

3. **Why are short sequences of introgressed genes spread over the genome?**

   (a) During introgression, new genes are introduced into the population and spread over generations. Recombination breaks up long sequences, leaving short stretches of introgressed genes dispersed across the genome. Genes that provide an adaptive advantage may spread and persist even in short segments, while neutral or deletarious genes are lost.

4. **Why do these short introgressed genes often occur side-by-side?**

   (a) Genes physically close to each other are less likely to be separated by recombination. If one gene in a cluster is advantageous, the entire cluster can hitchhike to fixation. When a beneficial muation spreads through a population, nearby genes are carried along, leading to clusters of fixed loci.