**Programming process steps in fix\_1\_haplo.py**

This Python program, **simulates the fixation probability of two alleles (A and a) for a single gene, and the number of generations required**. It can be used to test assumptions commonly found in evolutionary and young earth creationist models, specifically regarding **initial allele proportions and selection coefficients**.

**Program Overview and assumptions**

The simulation operates under the following key principles:

* **Wright-Fisher Model**: The program assumes **random mating** within the population.
* **Population Growth**: Population size changes are modeled using the **discrete Beverton-Holt model**.
* **Fitness Calculation**: The average fitness of the population is recalculated every generation. This calculation is based on the relative fitness values of the three genotypes (AA, Aa, and aa) and their current proportions within the population.

**Simulation Process**

The program runs multiple parallel simulations (controlled by variable Repetitions) for various user-defined scenarios. Each scenario is configured via an input\_data.txt file, where users specify parameters such as:

* **Initial population size (Ni)**
* **Population growth rate (r)**
* **Carrying capacity (K)**
* **Selection coefficient for allele A** (parameter s\_A)
* **Number of simulation attempts** (parameter attempts)
* **Dominance coefficient for allele A** (parameter h\_A}
* **Initial frequency of allele A** (p\_A\_i)

For each simulation attempt, the program iteratively calculates the allele frequencies, genotype frequencies, and mean population fitness. It then uses a binomial distribution to model random genetic drift and determine the allele frequencies in the next generation. The simulation continues until one of the alleles fixes (reaches a frequency of 0 or effectively 1).

**Output and Analysis**

* **Individual Simulation Results**: Saved in results\_data.txt, providing details for each repetition of every scenario.
* **Averaged Results**: Average of the results across all repetitions for each scenario is stored in results\_data\_avg.txt. These nclude the probability of fixation for both alleles, the average number of generations to fixation, and the standard deviations for these metrics.

**Description of the function ‘simulate\_population’**

The ‘simulate\_population’ function models the **population dynamics of a single gene with two alleles (A and a) over multiple generations**. It simulates how the frequencies of these alleles change due to factors like population growth, selection, and genetic drift starting from an initial proportion of e.g. allele *A*. Note that proportion A + proportion a = 1.

Here's a step-by-step breakdown of what the function does:

**1. Initialization of Counters and Parameters**

The function starts by initializing several counters to track the simulation's outcomes across multiple attempts:

* a\_count, A\_count: These variables count the number of times the 'a' allele and 'A' allele, respectively, become fixed (reach 0% or nearly 100% frequency) in the population.
* sum\_A\_fix\_gens, sum\_A\_fix\_gens\_sq: These track the sum of generations and sum of squared generations at the point when the 'A' allele fixes. These are used later to calculate the average and standard deviation of fixation generations.
* sum\_a\_fix\_gens, sum\_a\_fix\_gens\_sq: Similar to the above, but for the 'a' allele. (Note: These calculation do not consider the probabilities of fixing occurring, only the generation involved when fixing does occur).
* sum\_N\_A\_final, sum\_N\_a\_final: These accumulate the population size at the point of fixation for alleles 'A' and 'a', respectively.
* per\_generation\_data: This list stores detailed population and allele frequency data for each generation, but only if the parameter ‘document\_results\_every\_generation’ is set to True (such simulation take longer to run and more computer memory).

The function calculates the **fitness values** for each genotype (AA, Aa, aa) based on the selection coefficient (s\_A) and dominance coefficient (h\_A) read in from the file ‘input\_data.txt’.

Fitness AA​ = 1 + s\_A​ (1)

Fitness Aa​ = 1 + h\_A ​ × s\_A​ (2)

Fitness aa​ = 1 (3)

These fitness values are unchanged over all the attempts executed during the same simulation (the parameters for each simulation are defined in a row in input\_data.txt).

**2. Iterating Through Simulation Attempts**

The core of the function is a loop that runs for the number of attempts requested for each simulation in the file input\_data.txt. For each attempt, it resets the population to its initial state, i.e., as specified for that simulation in input\_data.txt. This is necessary, since during an ‘attempt’ involving growing populations, some key parameters get modified.

* **Population Size (N)**: Resets to the Ni (initial population size).
* **Fixation Threshold (p\_A\_fix)**: This is defined as 1/2N​. If the frequency of allele A (p\_A\_t) exceeds this, it's considered fixed.
* **Allele 'A' Proportion (p\_A\_t)**: Resets to the initial proportion p\_A\_i (as request in input\_data.txt).

**3. Simulating Generations Within Each Attempt**

Inside each attempt loop, another loop simulates the progression of generations:

* **Data Recording**: If parameter ‘document\_results\_every\_generation’ is set to True, the function records the current attempt, generation number, population size, and allele frequencies.
* **Fixation Check**:
  + It checks if the frequency of allele 'A' (p\_A\_t) has reached 0.0 (defined as the fixation of 'a' allele) or exceeded the p\_A\_fix threshold value (defined as the fixation of 'A' allele:

p\_A\_fix = 1 - (1 / (2 \* N)), (4)

where N is the population size.

* + If fixation occurs, it increments the corresponding counter (a\_count or A\_count), records the generation number at which fixation happened, and adds it to the respective sum for average generation calculation. It then breaks out of the generation loop for the current attempt, as the simulation for this attempt is complete.
* **Calculate Genotype Frequencies**: It determines the frequencies of AA, Aa, and aa genotypes based on the current frequency of allele A (p\_A\_t) assuming Hardy-Weinberg equilibrium, namely:

freq\_AA ​= p\_A\_t​2​ (5)

freq Aa​ = 2 × p\_A\_t​​ × (1−p\_A\_t​​) (6)

freq aa​ = (1−p\_A\_t​​)2 (7)

* **Calculate Mean Population Fitness**: It computes the average fitness of the population by weighting the fitness of each genotype (AA, Aa, and aa) by its frequency at the current time during the simulation.
  + Mean Fitness = (Freq AA​ × Fitness AA ​) + (Freq Aa​ × Fitness Aa​) + (Freq aa​×Fitness aa​)
* **Calculate Relative Fitness of the A Allele**: This determines the reproductive success of the A allele relative to the mean fitness of the population. This is a standard result from frequency-dependent selection theory in population genetics.



The python code that implements this is:

numerator\_A = 2.0 \* freq\_AA \* fitness\_AA + freq\_Aa \* fitness\_Aa

fit\_A = numerator\_A / (2.0 \* mean\_fitness) (8)

* **Population Growth (Beverton-Holt Model)**: If the growth rate *r* is not 0 (i.e., the population size is not fixed for these simulations), it updates the population size *N* using the Beverton-Holt model, which takes the carrying capacity *K* (maximum population size) into account to produce a logistic shape curve.



* + The fixation threshold p\_A\_fix is also updated if the population size changes, since eqn. (4) must now use a changing value of *N*.
* **Genetic Drift (Binomial Sampling)**: This is a crucial step where genetic drift is simulated. The number of 'A' alleles in the next generation is determined by a **binomial distribution**. This introduces randomness in allele frequencies due to chance events in reproduction.
  + The n\_A\_alleles (number of A alleles in the next generation) is sampled from a binomial distribution with 2×N trials (total number of alleles in the population) and a probability of success equal to the fit\_A (relative fitness of A allele). (Although the probability of each homologous chromosome being segregated into a new egg or sperm is 50%, the exact proportion generated each generation fluctuates randomly).
  + The new frequency of allele A (p\_A\_t) is then calculated by dividing n\_A\_alleles by the total number of alleles (2×N).

The python code that implements this is:

n\_A\_alleles = np.random.binomial(2 \* N, float(fit\_A))

p\_A\_t = n\_A\_alleles / (2 \* N)

Note that the population size *N* used considers the cases of either fixed or growing populations.

**4. Calculating and Returning Statistics**

After all attempts are completed, the function calculates various statistical summaries:

* **Average Final Population Size**: avg\_N\_A and avg\_N\_a are calculated by dividing the sum of final population sizes by the count of fixations for each allele. If no fixation occurred for an allele, the average is NaN (Not a Number).
* **Fixation Probabilities**: A\_fix\_prob and a\_fix\_prob represent the proportion of attempts where each allele fixed.
* **Standard Deviation of Fixation Probabilities**: A\_fix\_sd and a\_fix\_sd provide a measure of the variability in the fixation probabilities across attempts.
* **Average and Standard Deviation of Fixation Generations**: avg\_A\_fix\_gen, std\_A\_fix\_gen, avg\_a\_fix\_gen, and std\_a\_fix\_gen calculate the mean and standard deviation of the number of generations it took for each allele to fix. If an allele didn't fix in any attempt, these values are NaN.

Finally, the function **returns all these calculated statistics** along with the per\_generation\_data (if requested).

**Chronological steps performed in the full program**

**Steps carried out chronologically**

1. Import necessary Python modules including numpy, os, sys, time, multiprocessing, tempfile, and warnings.

2. Record the program start time.

3. Suppress runtime warnings about taking the mean of an empty slice.

4. Define a hardcoded MASTER\_SEED for reproducible results.

5. Specify filenames for configuration, input, and output files.

6. Attempt to import the yaml module; exit with an error if it is not installed.

7. Check if the configuration file exists; exit with an error if missing.

8. Load and parse the YAML configuration file; exit if parsing fails.

9. Extract and validate Repetitions, max\_generations, and document\_results\_every\_generation from the config; exit on missing or invalid values.

10. Prevent system sleep on Windows using Windows API calls.

11. Define the expected input file header string with seven parameters.

12. Define the detailed results file header string.

13. Define the averaged results file header string.

14. If the input file does not exist, create it with headers and example rows, then exit.

15. Read all non-empty lines from the input file.

16. If the input file is empty, repopulate it with headers and example rows, then exit.

17. Verify that the first line matches the expected header; exit if it does not.

18. If only the header is present, append example rows and exit.

19. Validate each data line to ensure it contains exactly seven semicolon-separated fields.

20. Validate that initial population size (Ni) is a positive integer within allowed bounds.

21. Validate that growth rate (r) is greater than –1.0.

22. Validate that carrying capacity (K) is an integer ≥ Ni.

23. Validate that selection coefficient (s\_A) is a float between –2 and 2.

24. Validate that the number of attempts is a positive integer within allowed bounds.

25. Validate that dominance coefficient (h\_A) is a float between –1 and 1.

26. Validate that initial allele frequency (p\_A\_i) is between 0.0 and 1.0.

27. Store valid parameter sets in a list; exit if any validation errors occurred.

28. Define the simulate\_population function to model allele dynamics with density-dependent population growth and selection.

29. Initialize counters and accumulators for fixation events and population sizes.

30. Precompute constants for population growth (r1 and rK).

31. If per-generation output is enabled, open a temporary file for writing per-generation data.

32. Precompute genotype fitness values based on s\_A and h\_A.

33. For each simulation attempt, initialize population size and allele frequency.

34. For each generation, compute and write current state (N, allele frequencies) to the per-generation file if enabled.

35. After writing, check for fixation (p\_A\_t = 0 or 1); record fixation type, generation, and population size if detected.

36. If no fixation, compute genotype frequencies, mean fitness, and post-selection allele frequency.

37. Update population size using Beverton-Holt (r ≥ 0) or exponential decay (r < 0), with stochastic rounding.

38. Break simulation if population size drops to zero or below.

39. Apply genetic drift via binomial sampling to determine next-generation allele frequency.

40. Include a defensive check to break if allele frequency becomes NaN.

41. Close the per-generation temporary file if it was opened.

42. Compute and return summary statistics: fixation probabilities, average fixation generations, and average final population sizes, along with the temp filename.

43. Define a worker function that sets a unique random seed per (SimNr, Rep) and calls simulate\_population.

44. In the main block, determine the number of available CPU cores.

45. If per-generation output is enabled, initialize the per-generation results file with a header.

46. Initialize the individual results file with a header and delete any existing version.

47. If Repetitions > 1, initialize the averaged results file with a header and delete any existing version.

48. Loop over each valid parameter set (SimNr).

49. For each SimNr, create a job for every repetition.

50. Execute all repetitions for the current SimNr in parallel using multiprocessing, preserving order.

51. If per-generation output is enabled, read each repetition’s temporary file in order and append its contents to the main per-generation results file.

52. Delete each temporary per-generation file after successful reading.

53. Write individual repetition results (fixation stats) to the main results file in repetition order.

54. Compute a combined homozygous/heterozygous loss metric as a weighted average of fixation outcomes.

55. If Repetitions > 1, aggregate raw counts across all repetitions to compute unbiased overall averages.

56. Reconstruct total fixation counts and sums from per-repetition results to avoid averaging bias.

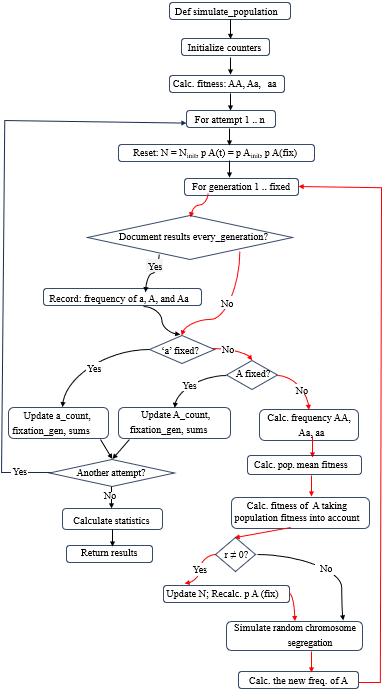
57. Calculate true overall fixation probabilities, mean fixation generations, and mean final population sizes.

58. Compute the combined loss metrics using the overall averages, handling edge cases where one allele fixes with probability 1.

59. Write the averaged results for the current SimNr to the averaged results file.

60. Print confirmation messages indicating where output files were saved.

61. Print the total program execution time.



**Figure 1**. Flow logic of the key function *simulate\_population* in fix\_1\_haplo.py