Speciation model for mixed-ploidy systems

Code documentation

Authors: Felipe Kauai, Frederik Mortier, Silvija Milosavljevic, Yves Van de Peer, Dries Bonte

Year: 2022

Programming language: Java

Summary

We developed a model to understand the eco-evolutionary dynamics of mixed-ploidy population, and which has been thoroughly analyzed in the manuscript "*Polyploid establishment and evolution: Understanding the eco-evolutionary dynamics of mixed-ploidy systems*". In this document we provide the documentation for the Java code developed, and pseudo-codes of routines to make it easier for researchers to implement the model in other programming languages, as well as to clarify the logic behind the simulation for those not acquainted with java syntax. The reader will find the following sections within this document:

- Model overview: The basic framework is explained, as a complement to the Methods section within the main manuscript.
- Main program: A detailed description of the main class in the Java project, design concepts and a pseudo-code for those interested in building the same simulation in another programming language.
- Main Methods: A detailed description of the core methods underlying the simulation dynamics, and their respective pseudo-codes to highlight the logic, and ease the implementation in other programming languages.

The entire project can be found on a GitHub repository at https://github.com/KauaiFe/Polyploidy-V1.0, along with a readme.txt file for further information.

Model overview

A population of clonal individuals are initialized within a square lattice and evolves under the combined forces of mutation, recombination and dispersal. At the start of the simulation, each individual is represented by a position in space and two chromosomes (diploids). These individuals produce gametes following meiosis and can mate with other individuals that inhabit a location within a certain mating radius. Offspring is then dispersed around the location of the first parent (seeker), and is part of the set of individuals that compose the next generation. Tetraploid individuals are formed by the fusion of 2n gametes formed by two diploids (produced with a certain probability) undergoing mating, or by 2n gametes produced by two tetraploid individuals. Also, tetraploids have a parameter that controls their fertility (gametes produced are not viable). A minimal form of assortative mating is implemented through a parameter that constrains mating to two individuals that share a minimum genetic similarity. Each generation is a population of individuals created in the previous generation, and therefore generations are discrete, and non-overlapping. Through constrained dispersal, mutation, recombination and assortative mating, the system evolves to a state where clusters of genetically isolated individuals can be identified. Such clusters are identified as species (see main manuscript for species definition and Main Methods section for implementation details).

Main program and design concepts

Java code description

The main program starts by defining all state variables that control the dynamics of the simulation. The user can study the influence of each parameter separately on the dynamics of the system, make them variable, or wrap them into a loop for retrieving data with multiple combinations.

```
int numGenerations = 10001; // For how many generations should the simulation run
int popSize = 2500; // Number of individuals that will inhabit the lattice
int genomeSize = 200; // Size of the genome. Each individual will have two chromosomes, each of size genomeSize
int matingRadius = 3; // The radius in which a seeker can find other potential mates (in lattice cells)
int dispersalRadius = 2; // The radius in which offspring can be dispersed (around the seeker)
double similarityThreshold = 0.95; // The minimum required genetic similarity for two individuals to be able to mate (95%)
double meiosisCoeff = 0.05; // When two chromosomes pair during meiosis, it is defined how many nucleotides are exchanged
double probPolyploid = 0.05; // Probability that a diploid individual will produce 2n gametes
double mutationRate = 0.00005; // Probability of mutation per nucleotide
double probDispersal  = 0.01; // Probability that an offspring will be dispersed instead of replacing the seeker's location
double probMate = 0.3; // Probability that an individual is not picked to reproduce (drift)
double reducedFertility = 0.12; // Probability that gametes formed by tetraploid individuals are null
```

Then, the initial population is initialized within the lattice, as well as all objects that carry the main methods of the simulation.
Population object contains methods for the initialization of the simulation. It initializes a lattice, creates a population of size *popSize* of individuals, and initializes a random genome (diploid) which will be the same for all individuals upon initialization of the simulation. It takes in as parameters *popSize*, and *genomeSize*, chosen by the user.
EvoMethods object contains methods related to the evolutionary dynamics of the system. These methods perform mutations on chromosomes, meiosis, alignment of chromosomes (see Main Methods section), as well as computation of genetic similarity and retrieval of individuals inside the mating radius of a seeker. It takes in as parameters *meiosisCoeff*, *probPolyploid*, *mutationRate*, *matingRadius* and *reducedFertility*.
SpTracker object contains a single public method for counting the number of species in the system. This is a deterministic and computationally intensive clustering algorithm that identify groups of genetically isolated individuals within the system (see Main Methods section). It takes in *similarityThreshold* as the only parameter.

```
Population pop = new Population(popSize, genomeSize);
EvoMethods solve = new EvoMethods(meiosisCoeff, probPolyploid, mutationRate, matingRadius, reducedFertility);
SpTracker sp = new SpTracker(similarityThreshold);
```

Individuals in the population have an Integer index and a two dimensional vector that stores chromosomes. The number of rows represents the ploidy (diploid or tetraploid), whereas the columns store the nucleotides of the given chromosome (row element). The positions are stored in a HashMap that stores the index of each individual and a vector with two elements (x, y).

```
HashMap<Integer, int[][]> population = pop.getPopGenomes(); // Retrieves genomes from Population object
HashMap<Integer, int[]> positions = pop.getPopPositions(); // Retrieves positions from Population object
```

The remaining of the code is a straightforward procedure that executes the simulation. The program is executed for a total of *numGenerations* iterations. At each iteration the simulation builds a new population from the previous one. This new population consists of offspring that are the results of successful mating trials between two individuals from the previous generation. A successful mating trial happens when both individuals (parents) share the same ploidy level (e.g., two diploids), their gametes share a minimum genetic similarity *similarityThreshold* and have the same ploidy (e.g., both gametes are 2n, which will produce a tetraploid offspring).

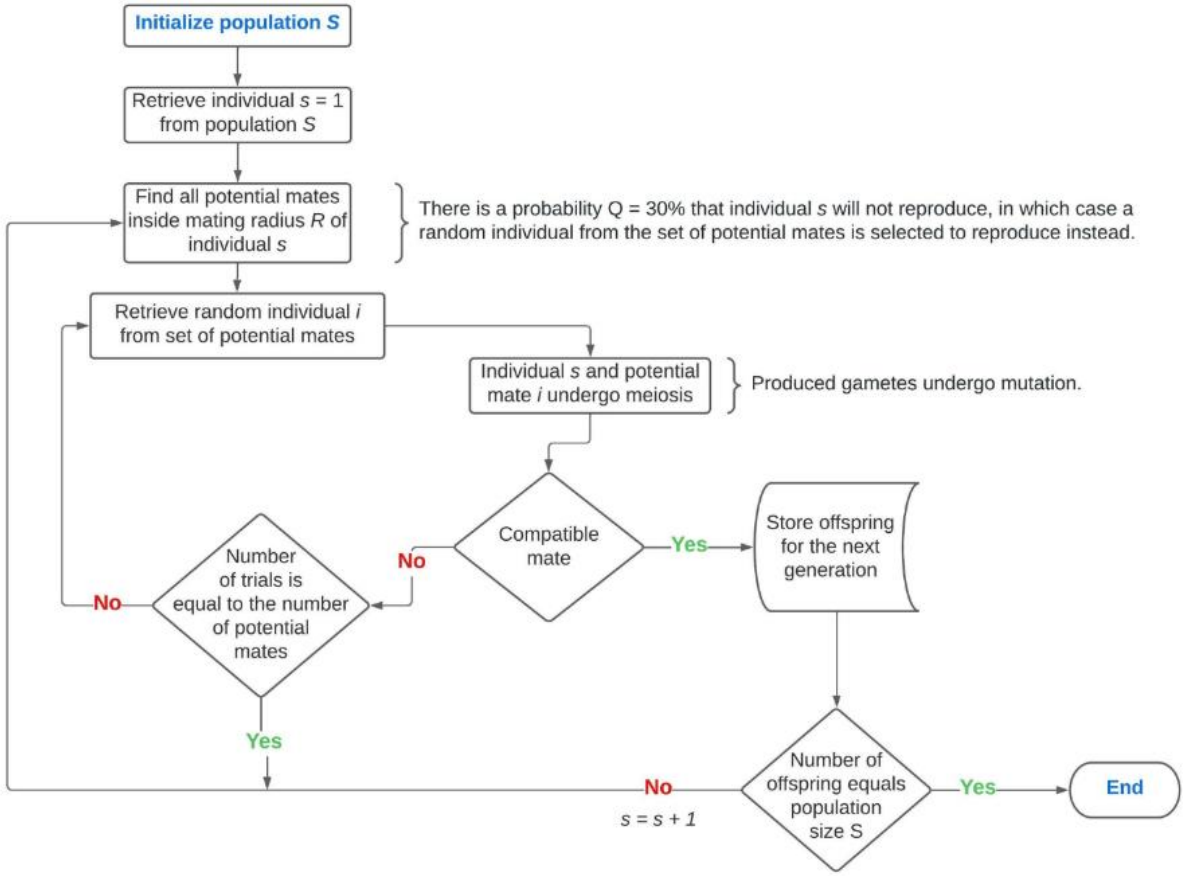
```
/* Empty hashmaps are initialized to store next generation iteratively. They will substitute the previous population and positions
hashmaps iteratively */
HashMap<Integer, int[][]> newPopulation = new HashMap<>();
HashMap<Integer, int[]> newPositions = new HashMap<>();

int offspringCount = 0; // control the index of offspring to be stored in the next generation
while(newPopulation.size() < population.size()) {

    int parent01 = offspringCount;
    int[][] parent01Chromosome = population.get(parent01); // Pick first individual from the population

/* The method findMates() takes in as parameters the positions HashMap and the first parent (seeker) index and finds all individuals
within matingRadius around the seeker. */
    ArrayList<Integer> mates = solve.findMates(positions, parent01);

    double pMate = Math.random(); // Probability that parent01 will be substituted (drift)
    if(pMate < probMate) {
        int newMate = (int) (Math.random()*mates.size());
        parent01Chromosome = population.get(mates.get(newMate));
    }
// Iterate over all mates inside matingRadius
    for(int i = 0; i < mates.size(); i++) {

        // A new gamete for parent01 is generated for every mating trial
        int[][] gameteP1 = solve.meiosis(parent01Chromosome);

       // Select a random mate inside the mating radius
        int mate = (int) (Math.random()*mates.size());
        int[][] mate02Chromosome = population.get(mates.get(mate));
        /*See main manuscript for a detailed description of how meiosis is performed*/
        int[][] gameteP2 = solve.meiosis(mate02Chromosome);

        // If gametes from parent 01 or 02 are null (reduced fertility)
        // then throw NullPointerException and continue execution
        try {
            /*See Main Methods section for a detailed description of how genetic similarity is computed */
            double similarity = solve.computeSimilarity(gameteP1, gameteP2);
            if(gameteP1.length == gameteP2.length && similarity >= similarityThreshold
                                    && parent01Chromosome.length == mate02Chromosome.length) {

                        int[][] offspring = joinGametes(gameteP1, gameteP2);
                        int[] offspringPosition = positions.get(parent01);

                        double t = Math.random();
                        if(t < probDispersal) {
                                offspringPosition = newPosition(positions, parent01, dispersalRadius);
                        }

                        newPopulation.put(offspringCount, offspring);
                        newPositions.put(offspringCount, offspringPosition);
                        offspringCount++;
                        break;

             }
        }catch(NullPointerException e) {

        }
    }
    /*Substitute previous population and positions*/
    population = newPopulation;
    positions = newPositions;
    /* The default program counts the number of species at every 500 generations */
     if(g % 500 == 0){
          double[][] adjMatrix = solve.buildAdjMatrix(population);
          int[][] species = sp.countSp(adjMatrix); /*See Main Methods section for a detailed description of sp.countSp()*/
     }

}
```

Pseudo-code and flowchart

SpeciationModel: Main Algorithm

```
1. Let G denote the number of generations
2. Initialize initial population S_g
3. for g = 1 to G do
4.     S_{g+1} ← Initiliaze empty population
5.     while |S_{g+1}| < |S_g| do
6.         for s = 1 to |S_g| do
7.             If individual s is not lost via drift with probability Q
8.                 M ← Find potential mates inside mating radius of individual s
9.                 for m = 1 to |M| do
10.                    p ← select random individual from M
11.                    individuals s and p undergo meiosis and mutation
12.                    if s and m are compatible
13.                        offspring ← joined gametes from s and m
14.                        S_{g+1} ← S_{g+1} + offspring
15.                        break
16.                    end
17.                end
18.            end
19.        end
20.    end
21.    S_g ← S_{g+1}
22. end
23. return gsTotal/p
```

Below, the reader may find the code flowchart, also presented in the main manuscript, highlighting the procedure to build a single generation.



Main Methods and details

EvoMethods class

The EvoMethods has a constructor that initializes the state variables that control the methods built for the class. The variables *meiosisCoeff*, *probPolyploid*, *mutationRate*, *matingRadius* and *reducedFertility* are required for the constructor.

```
public EvoMethods(double meiosisCoeff, double probPolyploid, double mutationRate, int matingRadius, double reducedFertility) {
                        this.meiosisCoeff = meiosisCoeff;
                        this.probPolyploid = probPolyploid;
                        this.mutationRate = mutationRate;
                        this.matingRadius = matingRadius;
                        this.reducedFertility = reducedFertility;
}
```

The class has five methods: *computeSimilarity*(), *meiosis*(), *findMates*(), *mutate*() and *buildAdjMatrix*().

To compute GS (see Main text) for two sets of more than one chromosome each, we apply a simple optimization heuristic (HeuristicGS) for the alignment of chromosomes. Let K^1 and K^2 be two sets

containing the chromosomes of parents (or gametes) 1 and 2, respectively, with the requirement that both parents (or gametes) have the same ploidy level, i.e., $|K^1| = |K^2|$. If K_i denotes the i^{th} element of set K, then the procedure reads as follows:

HeuristicGS(K^1, K^2).

1. Get ploidy level: p
2. Initialize variable to be returned: gsTotal $\leftarrow 0$
3. **for** i $= 1$ to $|K^1|$ **do**
4. $\quad$ L $\leftarrow$ find chromosome in K^2 such that $\mathrm{GS}(K_i^1, K_j^2)$ is maximum for all j
5. $\quad$ gsTotal $\leftarrow$ gsTotal $+ \mathrm{GS}(K_i^1, \mathrm{L})$;
6. $\quad$ K^2.pop(L)
7. **end**
8. **return** gsTotal/p

The method computeSimilarity() uses the HeuristicGS procedure for the alignment of chromosomes and compute similarity between chromosomes while alignment is performed. It takes in two genomes (or gametes) as parameters and returns a real value corresponding to the similarity between genomes (or gametes). The java code is presented below.

```
public double computeSimilarity(int[][] genome01, int[][] genome02) {

        double similarity = 0.0; /* similarity to be returned by the end of method execution */
        ArrayList<Integer> queue = new ArrayList<>(); /* start a queue to store chromosomes already compared */

        /* if a chromosome (genome02) is the most similar to one from the queue (genome01)
        *  then it must not match again another genome from the queue (genome01) */
        ArrayList<Integer> matched = new ArrayList<>();
        for(int i = 0; i < genome01.length; i++) {
          /*add all chromosomes from genome01 to queue. The choice of each genome is picked first is irrelevant*/
            queue.add(i);
        }

        while(!queue.isEmpty()) { */while the queue is not empty compared chromosomes*/

            int chromosome = queue.get(0);
            queue.remove(0);
            int compare = -Integer.MAX_VALUE;
            int matchedIndex = -1;
        /* look for best match from genome02 with chromosome from the queue*/
            for(int i = 0; i < genome02.length; i++) {
                boolean notMatched = true;
        /* See whether this chromosome has already a correspondence from previous elements from the queue */
                for(int j = 0; j < matched.size(); j++) {
                    if(i == matched.get(j)) {
                        notMatched = false;
                        break;
                    }
                }

                if(notMatched) {
        /*Then count similar alleles. The algorithm will find the best match for the chromosome in the queue*/
                    int similarAlleles = 0;
                    for(int k = 0; k < genome02[i].length; k++) {
                        if(genome01[chromosome][k] == genome02[i][k]) {
                            similarAlleles++;
                        }
                    }

                    if(similarAlleles > compare) {
                        compare = similarAlleles;
                        matchedIndex = i;
                    }
                }
            }

            if(matchedIndex != -1) {
                matched.add(matchedIndex);
                similarity += Double.valueOf(compare);
            }
        }

        return similarity/(Double.valueOf(genome01[0].length)*genome01.length);
}
```

Next, we have a $meiosis()$ method, which takes in as parameter a genome and returns a gamete. The procedure is thoroughly described in the main text. As far as the Java implementation is concerned, we implement a switch statement that distinguishes between diploid and tetraploid genomes. After gametes are formed, these are subject to mutations, with a built-in method called $mutate()$. The class also contains two simples methods, $findMates()$ amd $buildAdjMatrix()$. The first method simply scans all individuals inside a given mating radius around a focal parent, and returns a list of potential mates. The second performs pairwise genetic similarity computations for all individuals, using the previously described method $computeSimilarity()$, and returns an adjacency matrix, used for retrieving species identities (see below).

SpTracker and Population classes

The SpTracker class contains only two methods. One for counting the number of species, $spCount()$, and an auxiliary method, $removeId()$, to remove rows of a matrix returned by the previous method. The method $spCount()$ is a straight forward greedy clustering algorithm, that clusters individuals according to their genetic similarities. The method takes in as parameter the adjacency matrix referred to in the last section, and outputs a matrix, with the number of rows equal to the number of species, and the number of columns equal to the population size, representing individual identities. At each species (or row), all columns that have a 1, represent an individual of the population that is assigned to that species, and 0 otherwise. Let *Pop* be the current population of individuals and *AdjMatrix* the adjacency matrix representing genetic similarities *G* between all pairs of individuals, then the pseudo-code of the algorithm reads as follows:

countSp$(Pop, AdjMatrix)$.

1. $sp_{ij} \leftarrow initialize\ empty\ list\ to\ store\ individual\ j\ in\ species\ i$
2. $sp_{11} \leftarrow first\ individual\ of\ population\ is\ assigned\ to\ species\ 1$
3. $k \leftarrow 1$ /* dummy variable to control species identity */
4. **do**
5. | **for** j $= 1$ to Pop. Size **do**
6. | | **if** $AdjMatrix_{1j} \geq G$ **do**
7. | | | $sp_{kj} \leftarrow 1$ /* Assign individual j to species k */
8. | | **end**
9. | **end**
10. **while** there is no individual j $\in sp_{kj}$ such that $AdjMatrix_{ji} \geq G, \forall\ i \in Pop$
11. **If** Pop. size > 0 **do**
12. | $k \leftarrow k + 1$
13. | $sp_{k1} \leftarrow assign\ free\ individual\ from\ the\ population$
14. | Repeat lines $4 - 10$
15. **end**
16. ***return*** sp_{ij}