# GP4PG: pipeline for modelling and estimating demographic parameters in JAVA language

Jose M Serradell and Oscar Lao

Comments & questions: josemiguel.serradell@upf.edu; oscar.lao@ibe.upf-csic.es

How to cite: Serradell, JM. &  Lao, O. GP4PG pipeline for modelling and estimating demographic parameters in JAVA language. 2024.

# Table of contents

# Introduction

## Genetic Programming (GP)

GP is a specialized form of evolutionary algorithm that draws inspiration from the principles of biological evolution. GP shares similarities with other evolutionary algorithms like genetic algorithms (GA) and evolutionary strategies (ES). These algorithms work with a population of potential solutions, iteratively improving them through mechanisms such as selection, crossover/mutation, and replacement.

GP is a powerful optimization technique designed to mimic natural selection processes, evolving solutions for intricate problems. Initially conceived with the goal of automating the creation of computer programs based on high-level problem descriptions, the versatility of GP is evident in its application across various domains and to address different scopes, such as symbolic regression,  automated code generation, machine learning feature construction, evolutionary art or design optimization in engineering.

The fundamental concept underlying GP involves the evolution of a problem that can be modeled as a tree tailored to solve a specific task. These trees are composed of functions and terminals, and the structures evolve over generations. Functions represent operations or transformations, while terminals represent constants or variables.

GP unfolds within an initial population of randomly generated solutions (trees). Through iterative evolution, more effective solutions are granted the opportunity to 'reproduce.' This reproductive mechanism involves the combination of elements from existing solutions through recombination processes, analogous to genetic crossover in biological reproduction. The evolutionary journey of solutions is further enriched by incremental trial-and-error mutation development, fostering the creation of variations on successful solutions. Over successive generations, the population undergoes refinement as less effective solutions are replaced by more proficient ones. This dynamic process leads to an enhancement in the overall quality of solutions within the population.

Its capacity to adapt and innovate in response to diverse problem statements positions GP as a valuable tool for automated problem-solving and optimization tasks. In essence, genetic programming stands as a testament to the success of leveraging evolutionary principles to generate effective and tailored solutions to complex challenges.

Nevertheless, GP shows some well recognized problems of convergence and getting trapped in local optima. In particular, GP may converge prematurely if the algorithm settles on suboptimal solutions before exploring the entire solution space. Proper hyperparameter tuning, such as population size and mutation rate, is crucial to prevent premature convergence. Similarly, like many meta heuristic optimization algorithms, GP is susceptible to getting trapped in local optima, which are solutions that are better than their immediate neighbors but not necessarily the global optimum. To mitigate this, mechanisms like random mutation and diverse selection strategies are employed to explore a broader range of solutions. Finally, due to the exploratory nature of the whole process, GP may evolve solutions that overfit the training data, leading to poor generalization on unseen data. Balancing exploration and exploitation during evolution helps address this issue. An additional problem related to the implementation of GP tools is the computational cost of evaluating and evolving complex programs can be high.

## Population Genetics

In population genetics, a fundamental question revolves around the automated evolutionary modeling of the genetic variation among individuals, populations and species. This modeling proves particularly valuable when addressing demographic inquiries. It encompasses tasks such as identifying substructure within a population, estimating effective population sizes, determining the divergence times between two populations, establishing the specific demographic relationships between them, among others. The optimization of this modeling process poses a considerable challenge, as finding the optimal solution is categorized as an NP-hard problem. Manually exploring the vast space of potential parameter combinations is impractical due to its sheer complexity. However, leveraging the fact that demographic models are often represented as binary trees offers a promising avenue. Within this framework, evolutionary events such as shifts in effective population size, population extinction or expansion at specific times, and other demographic changes can be conceptualized as nodes within the tree. Herein lies the opportunity to exploit the GP framework. By utilizing GP, we can propose the evolution of demographic models towards configurations that minimize the disparity between observed genetic diversity in our data and the genetic diversity derived from simulating data based on the evolving demographic model.

The GP4PG framework, as proposed, takes into consideration the specific characteristics of demographic modeling and introduces modifications to the classical GP algorithm to accommodate these nuances.

## GP4PG

In GP4PG we implement an adapted version of GP for identifying the parameters and demographic events that produce the observed genetic diversity in a set of samples. Our approach takes advantage of the analogy GP-nodes-demographic-events to explore the demographic space. GP4PG selects from various demographic events, including admixture, addition or reduction of demes, and changes in migration rates. It explores different combinations to generate simulations using a demographic simulator such as fastSimcoal2, which are then tested against observed data. The way GP4PG quantifies the similarity between the observed genetic variation in our data and the one from a simulation is by computing the SFS of all the possible combinations of four populations (4-wise Site Frequency Spectrum (jSFS)).

The way how GP4PG evolves a set of solutions is slightly different from the classical GP framework defined by Koza [Koza, 1994]. In such a framework, recombination of subtrees between two successful tree solutions rules over the mutation, which is less frequently applied. A key issue when applying classical GP to modeling demographic trees is that the time when a demographic event (node) occurs matters. Therefore, recombination can only occur between nodes that occur at the same time scale, otherwise the resulting recombined tree could be incongruent, with demographic events that could be placed between two nodes occurring in time after it. GP4PG solves this problem by creating a hybrid algorithm between GP and invasive weed optimization (IWO) [Mehbarani et al. 2006; Misaghi & Yaghoobi, 2019], also a natural algorithm. The algorithm is inspired by the invasive nature of weeds in ecological systems. Weeds are known for their ability to spread and colonize new areas, which serves as a metaphor for the optimization process. In this algorithm, solution weeds with better fitness (based on the objective function) produce seeds in proportion to how good the fitness is, creating new solutions. Seed solutions are dispersed to new locations in the search space, simulating the invasive nature of weeds. Then, the algorithm evaluates the fitness of the new individuals and competes with the existing population. If a new individual is better, it replaces an existing one. The algorithm involves setting parameters such as population size, reproduction rate, and dispersal range, which influence the exploration and exploitation characteristics of the algorithm.

In the next section, we explain the characteristics of the demographic nodes.

## Populations, ecodemes and topodemes

Within GP4PG, populations are systematically organized into essential homogeneous groups known as "demes," which, in turn, are nested within broader categories termed "topodemes" and "ecodemes" [GilmourGregor1939]. Gilmour and Gregor conceptualized these terms from an ecological standpoint. An "ecodeme" refers to a collection of topodemes that share a common habitat, while a "topodeme" serves as a grouping for individuals originating from the same locality. The number of topodemes within an ecodeme varies based on the population's heterogeneity. In the context of population genetics and demographics, the term "topodeme" is employed as a unit of population substructure. It signifies distinct population nuclei within a given population, providing insights into the internal diversity observed in a specific region or "ecodeme" [Winsor2000].

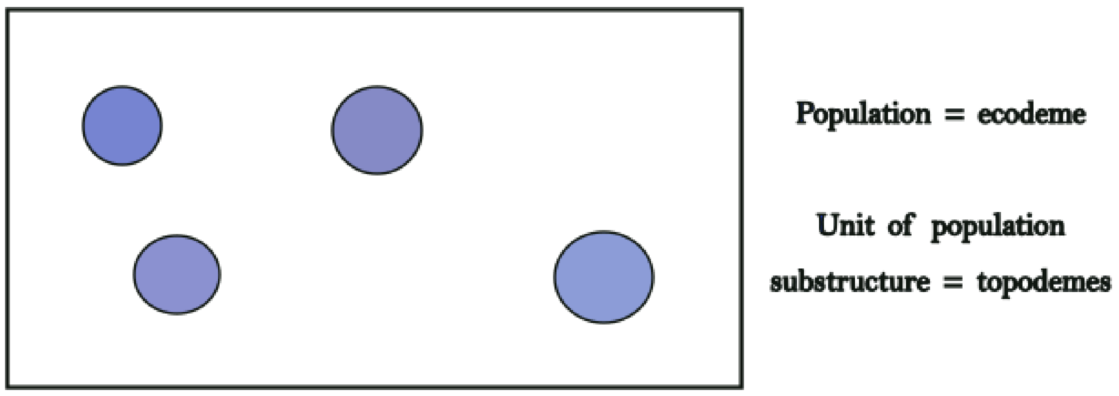

Figure 1: Each blue dot represents a topodeme, that is the basic unit of population substructure and represents a group of individuals originating from the same locality. The rectangle around is the representation of an ecodeme. Ecodemes are collections of topodemes that share a common habitat.

Whenever a new topodeme is created at a certain time, the unit is initialized with an effective population size sampled from a prior distribution, and it is placed at random within the territory that the ecodeme occupies.

## Migration

The GP4PG algorithm incorporates an Isolation by Distance approach [Wright1943] to model the migration rate within each ecodeme and between adjacent ecodemes. According to this approach, populations located at a greater distance from each other are less likely to exchange migrants compared to those in closer proximity.

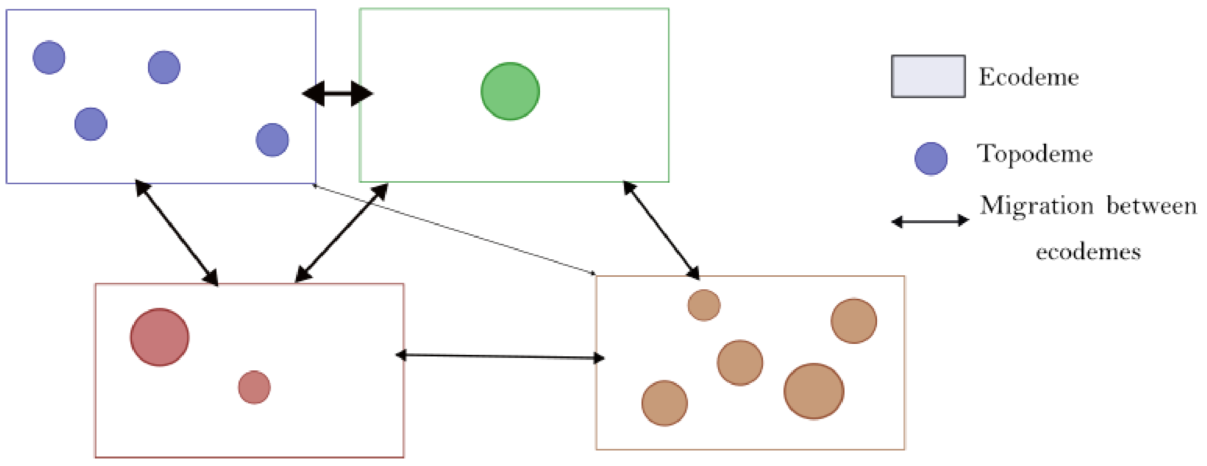

Figure 2: Diagram of the interaction between ecodemes under an Isolation by distance approach. Each ecodeme (colored squares) exchange migrants with their adjacent ecodemes at a different migration rate depending on the distance between ecodemes. The higher the distance, the lesser migration rate between ecodemes (width of the lines between demes refer to the migration rate between ecodemes).

## Ecodeme Splits

Ecodeme splits in GP4PG can be taken in two different ways. Either by setting a genetic barrier between topodemes from the same ecodeme, or by some of the individuals in a topodeme moving to a new location and creating a new ecodeme. Once the split has taken place, the relationship estimated as the exchange of migrants between the two ecodemes can also be modeled. Migration between the two ecodemes can decay over time after the split, suggesting a “soft” split. Setting the migration rate to 0 models a hard split.

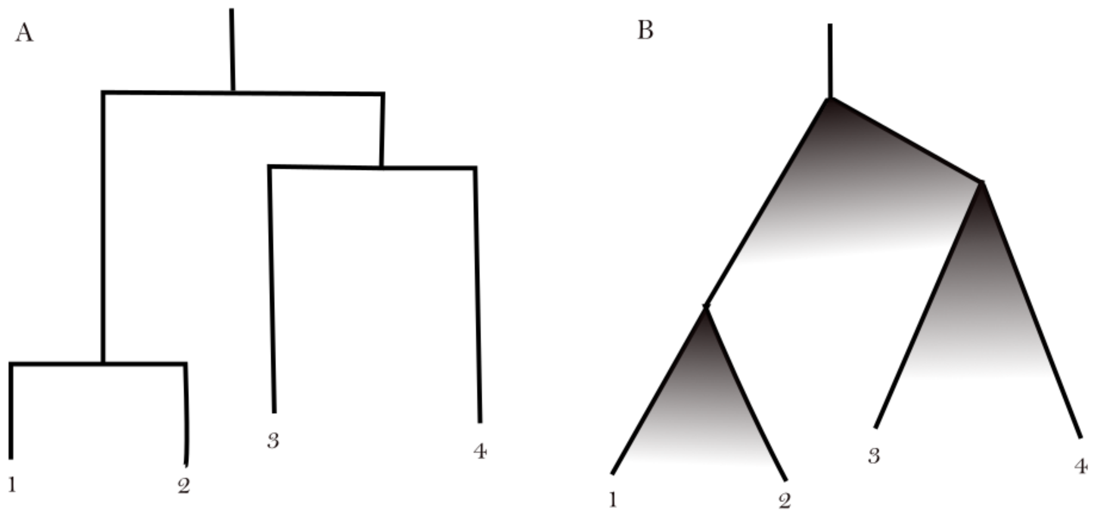

Figure 3: Scenarios that can be implemented in the GP4PG algorithm regarding population splits. In scenario A, where the migration decay rate is set to 0, an ecodeme split into two, producing a hard split where the newly formed ecodemes would stop exchanging migrants. In scenario B, we set a migration decay rate that makes that, after the split of an ecodeme, the new resulting ecodemes would keep having gene flow between them, reducing as time passes until they stop.

When modeling a soft split between Ecodemes i and j, migration rates across generations decay exponentially over time according to the formula:

$$migration_{i,j}(t) \;=\; (migration_{max} - migration_{min})e^{t-T} + migration_{min}$$

Where T is the time in backward of the split between Ecodemes i and j, $migration_{max}$ is the migration at time T, and it is the maximum value of migration between the two Ecodemes, and $migration_{min}$ is the minimum migration that can be achieved between the two Ecodemes as the time of divergence between both goes to infinity.

## Admixture

Single admixture pulses between two ecodemes transform a tree into a more general graph. GP4PG models this situation by embedding the admixture event nodes in the GP framework. This is accomplished by identifying the source EcoDeme in the branch where the admixture event is set, and looking for the sink ecodeme that exists at the time when the admixture event is active.

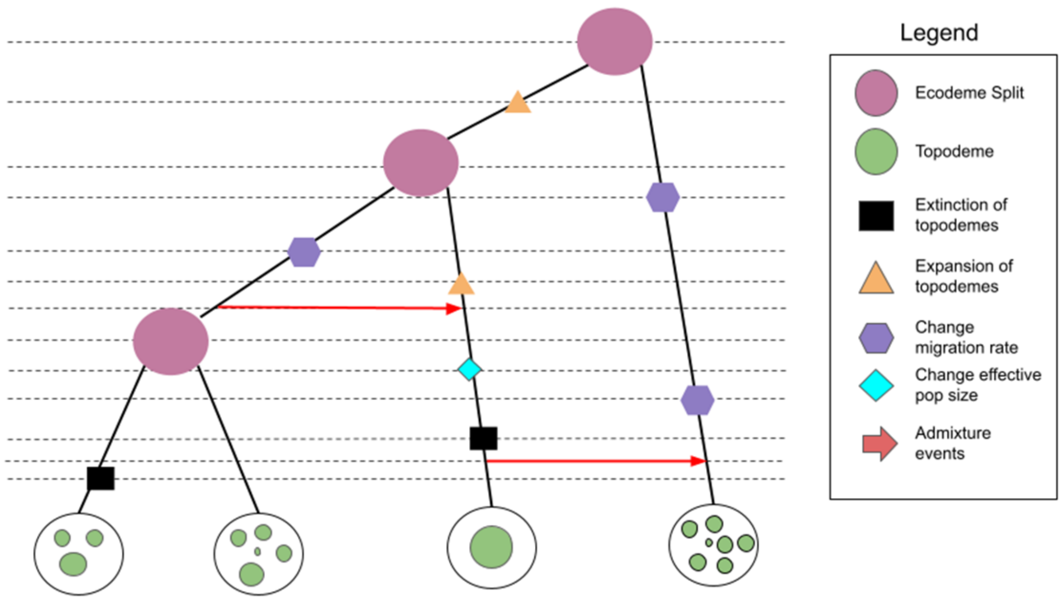

Figure 4: Potential evolutionary parameters that can be part of a tree in the GP4PG algorithm. Admixture events generate punctual gene flow from an ecodeme that works as a source population towards another ecodeme that is the sink population.

# Implementation

## Folder Organization

GP4GP runs all the iterations sequentially although we want to develop multiple independent runs so we parallelized the runs as independent jobs in the cluster. The structure of folders of an GP4PG project is:

1. Main folder
    a. fastSimcoal_0
    b. fastSimcoal_1
    c. …
    d. fastSimcoal_n

In *main_folder* we have all the information required to retrieve the genetic variation of the observed data in Plink binary format (please refer to the https://www.cog-genomics.org/plink2/formats) and identify the callable regions. The observed data must contain at least two individuals from each population and one additional individual with the name "Ancestral" that defines the ancestral allele.

We also have the fastSimcoal folder, each containing the executable of fastSimcoal2. The analysis has been performed with the fastSimcoal2 version fsc26 (http://cmpg.unibe.ch/software/fastsimcoal2/).

The observed data and the genes and CpG island files must be created before we attempt to run any analysis.

## **Creating a new project**

The JAVA pipeline is implemented to input the genomic data in PLINK format (genome.bim) and a set of callable regions of the genome in BED format (file.bed) and create the GP4PG output file (ModelX.xxx) with the error distance in each generation of a run and the parameters of the accepted models, and a fastsimcoal2 arlequin file (ModelX.par) with the parameter values for the model with the least error that can be used to validate the GP4PG model.

The JAVA project is called GP4GP_suite and requires some jar libraries to work. Some of them have been developed by the authors of the paper.

The first step will be to set up the Netbeans GP4PG_suite project in your favorite editor. A simple way is to clone the GP4PG repository, opening it in Netbeans, and setting in Properties the jar libraries from the libraries folder.

Creating a new project will require generating a class where to run the GP4PG algorithm we can named it Population.class that will contain all the information related to the exact location of the main folder, the observed individuals that we are going to use for the training and replication and the populations that we want to use to generate the SFS,as well as the evolutionary parameters that are going to be added to the evolutionary algorithm and the main runner of the GP4PG and the *fastsimcoal* file name. Recall that we are going to use ONE individual per population in the training and ONE individual per population in the replication. In principle, the training individuals should not be used for replication, but in some cases, where only one individual is available the same individual could be used for training and replication. This should not be a big issue, since the training only considers the observed individuals for noise injection and also we have implemented the possibility of random selection of half of the callable regions as the test data so even if we only have one individual for a population we could use a unique set of SNPs for test and replication.

An implementation that uses the ProjectInformation_EA would look like this:

```
public class ProjectInformation_EA {

/**
* Hard coded information of the test project
* @return
* @throws Exception
*/

    public static void main(String[] args) throws Exception {

        // Total number of generations for a GP4PG run (not the same as the generations
of the populations. This is the total number of time the program will try to find a
better solution

       int time_lapses = 200;

        // Working directory. Within this directory you must have the plink bed file
and a folder with fastSimcoal2 (folder must include a masked_regions.txt file)

        String working_folder = "/path/to/GP4PG/directory/";

        //Plink file without extension: SNP data of the tested populations

        String bed_file = "observed.data";

        // Model to execute
        Tested_Model modev = new Tested_Model();

        // Implementation to run the evolutionary algorithm approach

     RunnerEA  runGP4PG  =  new  RunnerEA(working_folder,  Integer.parseInt(args[1])  ,
bed_file, modev);
```

Integer.parseInt(args[1]) = number of the fastSimcoal_n file where the current replica of the GP4PG runs. If you use a cluster and parallelize each run, defining this args[1] in the launcher of the GP4PG could be interesting for easier parallelization.

Next, we have to hardcode the mutation and recombination rates specific to the species we are testing as well as names of the samples we are using in the analysis. All this information is going to be stored in a object called FeaturesOfData that is going to be called later by the executor of the GP4PG

```
       //  Recombination  and  mutation  probabilities.  They  must  extend  the  class
RandomBounded

       RandomTruncatedNormal recomb = new RandomTruncatedNormal(1.8 * Math.pow(10, -
8), 1 * Math.pow(10, -11), Math.pow(10, -11), Math.pow(10, -7));
      //MUTATION RATE in human

       RandomTruncatedNormal mutat = new RandomTruncatedNormal(1.61 * Math.pow(10, -
8), 0.13 * Math.pow(10, -8), 0.6 * Math.pow(10, -10), 0.6 * Math.pow(10, -6));

      // individuals used for the training and replication
       String[]individuals_training= {"Ind_0","Ind_2","Ind_4","Ind_6"};

       String[]individuals_replication={"Ind_1","Ind_3","Ind_5","Ind_7"};

      /* Object with all the features recovered for the data
       *  Includes the recombination and mutation rates which in this case
       *  Also include a value between 0 and 1 that is equivalent to the
       *  number of fragments of the callable region that are going to be
       *  used for the training process
       *  And the training and replication samples
       */
  FeaturesOfData fod = new FeaturesOfData(recomb, mutat, 0.5, individuals_training,
individuals_replication);
```

The following step in the ProjectInformation_EA is to introduce the evolutionary parameters that the GP4PG can use to modify the original models by means of a mutation event.

```
       // Events that can be added or removed from the tree over time
       // Total number of different demographic events that can be added

          Probability_DemographicEventCreator[] probs = new
Probability_DemographicEventCreator[2];

       /* Define each demographic event that can be added to the specific
       *  model.
       *  There are 6 demographic events implemented in the current version
```

```
        *  the GP4PG algorithm: Admixture, Change_of_Migration_In_Backward,
        *  Change_Ne_In_Backward, Extinct_Demes_In_Forward,
        *  Increase_Demes_In_Forward and Stop_Migration_In_Backward
        *  For each demographic event we have a probability of occurrence,
        *  and specific parameters (time at which the parameter can appear
        *  and initial state of the parameter)
        */

   probs[0]       =       new       Probability_DemographicEventCreator(1,       new
DemographicEventCreator_Extinct_Demes_In_Forward(new  RandomUniform(10,  70000),  new
RandomUniformBoundedIntMinIsOne(2)));

   probs[1]       =       new       Probability_DemographicEventCreator(0.3,       new
DemographicEventCreator_Admixture(new       RandomUniform(10,       70000),       new
RandomUniform(0.001,  0.15)));



       //  Parameters  of  the  evolutionary  algorithm.  Number  of  total  generations
evolving, population size of solutions, min and max number of offspring produced by
the worse and best solution of a given generation

        int n_generations = 200;

        int n_pop_EA_size = 100;

        int n_min_offspring = 2;

        int n_max_offspring = 8;

       // Run the GP4PG algorithm with all the previous information
       // name of the version of the fastsimcoal simulation we are using

        runGP4PG.setFastSimcoal2Name("fsc26");

       //results        of        the        GP4PG        are        stored        in
"/path/to/GP4PG/directory/evoAlgorithm_int_replica.err"

       runGP4PG.runEA(fod,   probs,   time_lapses,   n_generations,   n_pop_EA_size,
n_min_offspring, n_max_offspring, 0.7);

       }
}
```

It indicates that all the information of the project is in `/path/to/GP4PG/directory`, that we will use the executable fsc26 which is stored in a folder  fastSimcoalx, 200 generations of the Evolutionary Algorithm will be simulated calculating the error against the simulated data. At each generation a 100 maximum number of solutions would be produced using as training for noise injection the training samples half of the random set of callable regions and  the replication samples for the parameter/model validation, using all the populations to retrieve. In this case we set that the solutions with a lower error would produce a 8 offspring while this would decrease until we reach the solutions with a higher error that are going to produce 2 new offspring (not making the minimum offspring equal to 0 allow a faster search of the solution space as the non-optima solution will still participate in the search and won't be discarded)

In the following step we are going to learn how to build an initial model in the GP4PG framework.

## Generate the unspecific parameters for the comparing topologies *GP4PG_Model.java*

In order to initialize the GP4PG algorithm we first need to define some parameters that are common for all the topologies we want to compare. These parameters are implemented in the public class GP4PG_Model that is extended from the ModelToRun abstract class. Specifically we have to define the geographical space of each of the populations tested. Each of the populations is defined by an ecodeme and one or more topodemes that are part of this ecodeme. These topodemes are the local units of populations and the interaction of the different topodemes that form an ecodeme is what we identify as population substructure. The topodemes interact with other topodemes by interchanging genetic material following an Isolation by distance pattern that gives an increased migration rate to those demes that are closer together and a smaller migration rate to demes that are further away. In order to situate the topodemes in a space we have to define a geospatial space for each of the ecodemes (populations) that are tested in the analysis.

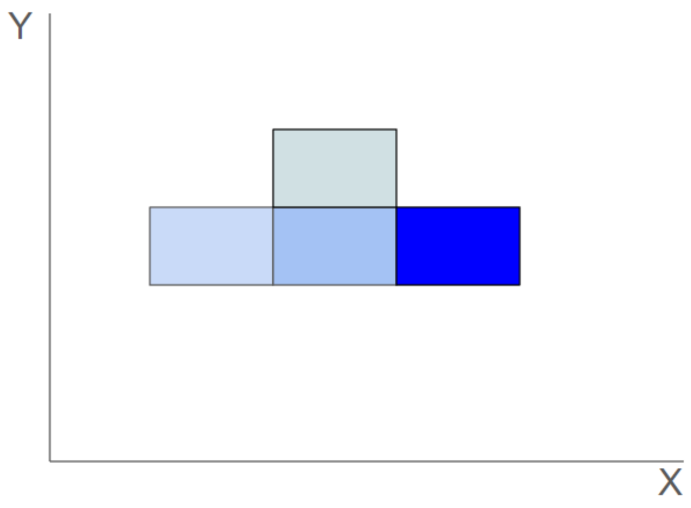

Figure 5: Geospatial distribution of ecodemes of equal size. Each colored square represents an ecodeme that will have a discrete number of topodemes.

```
public class NA_Model extends ModelToRun {

    protected NodeEvent_Sample_EcoDeme e_0, e_1, e_2, e_3;
    protected NA_Model ma_ascertained;
    protected double [] probability_colonization = {0.5, 0.1, 0.1, 0.3};

    protected void initialize_pops() throws Exception_NodeEvent {
        //Influence area for each population (Pop0)
        Coordinate upper_left_0 = new Coordinate(-30, 10);
        Coordinate lower_right_0 = new Coordinate(-10, -10);
        Rectangular_Region    amazigh    =    new    Rectangular_Region(lower_right_0,
upper_left_0);
        //Pop1
        Coordinate upper_left_1 = new Coordinate(-10, 10);
        Coordinate lower_right_1 = new Coordinate(10, -10);
        Rectangular_Region    na_arab    =    new    Rectangular_Region(lower_right_1,
upper_left_1);
        //Pop2
        Coordinate upper_left_2 = new Coordinate(30, 10);
        Coordinate lower_right_2 = new Coordinate(50, -10);
        Rectangular_Region   middleEast   =   new   Rectangular_Region(lower_right_2,
upper_left_2);
        //Pop3
        Coordinate upper_left_3 = new Coordinate(-10, 30);
        Coordinate lower_right_3 = new Coordinate(10, 10);
        Rectangular_Region    europe    =    new    Rectangular_Region(lower_right_3,
upper_left_3);
```

Once we have defined the space where the ecodemes inhabit next we have to initialize each of the ecodemes. In order to initialize an ecodeme we have to define 6 parameters. The number ID of the ecodeme, the total number of samples (observed data) that form this ecodeme, the initial distribution of topodemes possible, the migration rate within the demes and the name of the population.

```
int[] samples_0 = {2};

        /**
         * Create Ecodemes for all poppulations
         * @id, @samples,
         * @topodemes, @Ne, @migrationRateWithinDeme, @population
         */

    CreateEcoDeme deme_0 = new CreateEcoDeme(0, samples_0, new RandomUniform(1, 5),
new RandomTruncatedNormal(20000, 10000, 1000, 50000), new RandomUniform(Math.pow(10,
-6), Math.pow(10, -4)), Pop0);
        e_0 = new NodeEvent_Sample_EcoDeme(deme_0);

        CreateEcoDeme deme_1 = new CreateEcoDeme(1, samples_0, new RandomUniform(1,
5),     new     RandomTruncatedNormal(20000,     10000,     500,     40000),     new
RandomUniform(Math.pow(10, -6), Math.pow(10, -4)), Pop1);
        e_1 = new NodeEvent_Sample_EcoDeme(deme_1);

        CreateEcoDeme deme_2 = new CreateEcoDeme(2, samples_0, new RandomUniform(1,
5),     new     RandomTruncatedNormal(20000,     10000,     1000,     50000),     new
RandomUniform(Math.pow(10, -6), Math.pow(10, -4)), Pop2);
        e_2 = new NodeEvent_Sample_EcoDeme(deme_2);
        CreateEcoDeme deme_3 = new CreateEcoDeme(3, samples_0, new RandomUniform(1,
5),     new     RandomTruncatedNormal(20000,     10000,     1000,     50000),     new
RandomUniform(Math.pow(10, -6), Math.pow(10, -4)), Pop3);
        e_3 = new NodeEvent_Sample_EcoDeme(deme_3);
       }
```

The values for each parameter of the ecodeme are just some indicative data to define the populations but the final model could have very different values as the GP4PG evolves to find the most suitable group of parameters to fit the data.

The final step in this class includes a probability to select each model when to be used for the GP4PG. By this probability the GP4PG ascertains which model is going to test in each of the possible solutions in a generation.

```
/**
     *
     * @param gto
     * @param time_lapse
     * @return
     * @throws IOException
     * @throws Exception_NodeEvent
     */
    @Override
    public Evolution_FastSimcoal2 get_evolutionary_model(Genome_To_Simulate gto, int
time_lapse) throws IOException, Exception_NodeEvent {

       double[] prob_model = {0.25,0.25,0.25,0.25};
      RandomMultinomial rmul = new RandomMultinomial(prob_model);
      int model_i = rmul.sample();
      switch (model_i) {
            case 0 -> {
             ma_ascertained = new GP4PG_ModelA();
             return ma_ascertained.get_evolutionary_model(gto, time_lapse);
            }
            case 1 -> {
             ma_ascertained = GP4PG_ModelB();
             return ma_ascertained.get_evolutionary_model(gto, time_lapse);
            }
            case 2 -> {
             ma_ascertained = new GP4PG_ModelC();
             return ma_ascertained.get_evolutionary_model(gto, time_lapse);
            }
            default -> {
             ma_ascertained = new GP4PG_ModelD();
             return ma_ascertained.get_evolutionary_model(gto, time_lapse);
            }
       }
       @Override
             public String model_name() {
              return ma_ascertained.getClass().getCanonicalName();
             }

       @Override
             public int number_of_sampled_populations() {
              return 4;
             }
}
```

In this case we want to test four different models with four different topologies to tests and indicate that the GP4PG can select either of the models with the same probability (0.25). The next step in the pipeline is to build each model.

## Generating a model in the GP4PG JAVA framework *GP4PG_ModelA.java*

Let’s take as an example the following demographic topology that we want to test.

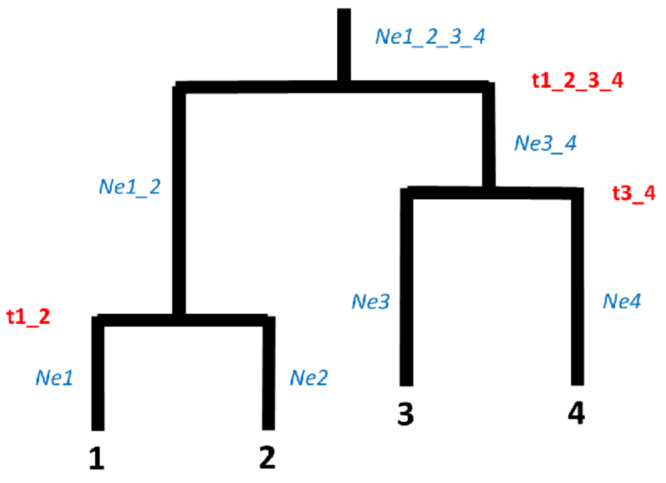

Figure 6: Model A. Ne = effective population size of a population x, tx_y = time of split between population x and y

We now have defined the initial effective population size and the geospatial space of each ecodeme as well as for the evolutionary parameters of interest  that have been described in the previous sections. So now, the only things left to hard code in the analysis are the basic topology of the different tested models, the initial split times and the migration (presence or absence of current day migration) and the migration decay, which relates to the reduction of gene flow between recently separated demes in time. To do that we extend the GP4PG_model class which is an extension of the abstract class ModelToRun. The GP4PG_model class will be explained in the next section as it includes the basic parameters of the initial populations in the models.

```
public class GP4PG_ModelA extends GP4PG_Model{

    @Override
    public Evolution_FastSimcoal2 get_evolutionary_model(Genome_To_Simulate gto,
int time_lapse) throws IOException, Exception_NodeEvent {

       NodeEvent_Split model = build_model();
      return new Evolution_FastSimcoal2("NorthAfrica_ModelD", model, gto,
time_lapse);

    }

    public NodeEvent_Split build_model() throws Exception_NodeEvent {
        super.initialize_pops();

         /**
         * Splits and migratory decay events
         * Have to set up one decay everytime we have a split (range _ min _ max
migration)
         * In human population most cases of Split follow colonization pattern
rather than genetic barrier but we can set both in case of doubt
         */

//Event 1 : e0 <> e1 (Pop1 with Pop2)

        Range rg1 = new Range(0.0, Math.pow(10, -2));

        double min_v1 = new RandomUniform(0.0, Math.pow(10, -6)).sample();
        double max_v1 = new RandomUniform(min_v1, Math.pow(10, -2)).sample();

        Function_Migration_Decays_Exponentially fc1 = new
Function_Migration_Decays_Exponentially(min_v1, max_v1, 2, Math.pow(10, -4), rg1);
```

The migration decay function includes a minimum value of migration between populations, a maximum value, the scale at which it decay, the standard deviation and the range of migration.

```
        NodeEvent_Splits_By_Colonization_Binary e_0_1 = new
NodeEvent_Splits_By_Colonization_Binary(new RandomUniform(5, 500), new
RandomUniform(Math.pow(10, -5), Math.pow(10, -2)), fc1,probability_colonization);
```

Next we implement the split event between the 2 populations, for it we must set a time split parameter distribution for the event. Bear in mind these values are only to be able to initialize the GP4PG but the resulting values could be outside this range as the algorithm evolves and tests parameter values not implemented by the user to find the most optimum set of values that explain the data. It includes a random Uniform distribution of the time of split, a probability that migration within the ecodeme occurs, the migration decay function for this split and a probability value that indicates which deme is the source and which is the sink when the populations coalesce backwards.

Last, we have to define which populations are going to be part of the colonization event.

```
//set event > set event 0 = event before // set event 1= event after
        e_0_1.set_event(0, e_0);
        e_0_1.set_event(1, e_1);
```

Repeat the process for each split in the topology

```
//Event 2 : e0_1 <> e2
         Range rg2 = new Range(0.0, Math.pow(10, -3));

        double min_v2 = new RandomUniform(0.0, Math.pow(10, -6)).sample();
        double max_v2 = new RandomUniform(min_v1, Math.pow(10, -3)).sample();


        Function_Migration_Decays_Exponentially fc2 = new
Function_Migration_Decays_Exponentially(min_v2, max_v2, 2, Math.pow(10, -4), rg2);

        NodeEvent_Splits_By_Colonization_Binary e_0_2_3 = new
NodeEvent_Splits_By_Colonization_Binary(new RandomUniform(80, 1200), new
RandomUniform(Math.pow(10, -5), Math.pow(10, -2)), fc2,probability_colonization);

        e_0_1_2.set_event(0, e_2);

        e_0_1_2.set_event(1, e_0_1);

//Event 3 : e0_1_2 <> e3
        Range rg3 = new Range(0.0, Math.pow(10, -3));

        double min_v3 = new RandomUniform(0.0, Math.pow(10, -6)).sample();
        double max_v3 = new RandomUniform(min_v1, Math.pow(10, -3)).sample();

        Function_Migration_Decays_Exponentially fc3 = new
Function_Migration_Decays_Exponentially(min_v3, max_v3, 2, Math.pow(10, -4), rg3);

        NodeEvent_Splits_By_Colonization_Binary e0_1_2_3 = new
NodeEvent_Splits_By_Colonization_Binary(new RandomUniform(100, 1300), new
RandomUniform(Math.pow(10, -5), Math.pow(10, -2)), fc3,probability_colonization);

        e0_1_2_3.set_event(0, e_3);
        e0_1_2_3.set_event(1, e_0_1_2);

         e0_1_2_3.specify_times();

        return e0_1_2_3;
      }

  @Override
    public String model_name() {
        return this.getClass().getCanonicalName();
    }

}
```

Return the parameter of the selected model and the name of the model

## Running simulations

Simulations are run in the folders fastSimcoal_n, where n ranges between 0 and the total number of parallel independent runs of GP4PG we want to test.

In order to run the simulation, we will need to specify which are the fragments we want to simulate in the file “masked_regions.txt” within the working folder, which must be generated BEFORE running any simulation.

The format of the fragments that are we are going to simulate is:

*1 1 10000000 : 1 10000000*

*2 1 10000000 : 1 10000000*

In this example, we are going to simulate two fragments, each of 10Mb (notice that a REAL project will use the whole genome). Within each fragment, we are going to consider all the regions as callable. If that was not the case, we could, for example, specify which are the callable fragments within each region:

*1 1 10000000 : 1 5000000, 6000000 9000000*
*2 1 10000000 : 1 10000000*

This means that fragment one, which comprises 10Mb, has two callable regions. One starting at position 1 until 5Mb and another starting at 6Mb until 9Mb. We recommend to mask the sites that are gens or known CpG islands to keep the most neutral set of SNPs for the demographic analysis.

The typical run of a GP4PG algorithm would give an output that will look like:

Information of the paths to the data and the main parameters of the evolutionary algorithm

```
time_lapses 200 n_generations 200 popEASize 100 min_offspring 2 max_offspring 8
working_folder /path/to/working/folder
Using the PLINK FILE in /path/to/observed.data
```

Fragments used in the training phase of the GP4PG, random sampling of the whole set of callable regions are used in each run

```
TOTAL NUMBER OF FRAGMENTS 5375 FRAGMENTS IN TRAINING 2687
FRAGMENT   15   68842239   68852239:     68842239   68844718,   68845041   68852239
PERCENTAGE_CALLABLE: 0.9677 HAS_SNPS 16
```

```
FRAGMENT  4  112013145  112023145:   112013145  112023145  PERCENTAGE_CALLABLE:  1.0
HAS_SNPS 26
FRAGMENT 4 56770645 56780645:  56771088 56780460 PERCENTAGE_CALLABLE: 0.9372 HAS_SNPS
32
…
FRAGMENT 7 38151420 38161420:  38151420 38161420 PERCENTAGE_CALLABLE: 1.0 HAS_SNPS
44
```

Initial joint 4-Site Frequency Spectrum of the observed data

```
REPLICATION FOUR_SFS [26904.0, 3898.0, 625.0, 3672.0, 916.0, 287.0, 357.0, 160.0,
97.0, 3787.0, 893.0, 295.0, 1127.0, 647.0, 336.0, 264.0, 259.0, 221.0, 346.0, 138.0,
114.0, 220.0, 255.0, 136.0, 108.0, 117.0, 220.0, 4099.0, 785.0, 279.0, 1022.0, 600.0,
346.0,…]
ROWS 2687 COLUMNS 6559
```

After this there are all the simulations and the distance of each simulation to the observed data. In this case we have 200 generations for the GP4PG algorithm and the error (scaled distance between observed data and simulations goes from 30.37 to 2.5)

```
GENERATION -1 30.377990862265534
GENERATION 0 26.28441250886905
GENERATION 1 19.414009502819454
GENERATION 2 17.980956746373117
GENERATION 3 9.801126833694148
GENERATION 4 9.847056173386804
GENERATION 5 8.098459124984105
GENERATION 6 8.156290953139738
…
GENERATION 195 2.525790392851644
GENERATION 196 2.695685498244917
GENERATION 197 2.633325198242919
GENERATION 198 2.658668407046167
GENERATION 199 2.559485974146053
```

Once the program has finished all the simulations it will output the 4-jSFS of the best solution and the parameters of such solution in the same output file, as well as producing a ModelX.par file with the best model of the run.

```
[26620.0, 4083.0, 594.0, 3560.0, 917.0, 304.0, 430.0, 181.0, 83.0, 3677.0, 1023.0,
313.0, 1131.0, 682.0,...]
BEST SOLUTION 0 with_fitness 2.559485974146053

Demes structure
NodeEvent_Sample_EcoDeme -> [[6] -12.457947200081678;-21.526187310072157:49741.0]
…
NodeEvent_Sample_EcoDeme   ->   [[0]   -6.743717220523688;1.182742330435719:14852.0]
migration_within 1.705580775406624E-5

Demographic events from the most recent to the oldest

NodeEvent_Splits_By_Colonization_Binary
NodeEvent_Splits_By_Colonization_Binary  56  [0]  -6.74;1.18:14852.0  merges  backward
with [2] 43.07;-4.70:22620.0
```

```
migration decays in forward with min migration = 8.89E-7 max_migration = 0.008

…

NodeEvent_Change_Ne_In_Backward 208 [[1] -19.13;-7.93:997.0] migration_within 9.89E-
5       sends_mig_backward_to_ecodeme_with_prob             [6,       =       7.63E-
7]sends_mig_backward_to_ecodeme_with_prob              [0,2,       =       7.84E-4]
sends_mig_backward_to_ecodeme_with_prob           [4,           =           4.82E-5]
sends_mig_backward_to_ecodeme_with_prob                [3,         =         4.52E-4]
sends_mig_backward_to_ecodeme_with_prob                [7,         =         7.74E-7]
sends_mig_backward_to_ecodeme_with_prob [5, = 3.52E-6]  changes Ne from 9453 to 997
…

NodeEvent_Admixture 3382 [[6] -10.50;-16.04:1165.0;[6] -21.45;-15.85:33156.0;[6] -
5.27;-12.28:48079.0;[6]         -19.54;-18.84:907.0]migration_within         4.97E-5
sends_mig_backward_to_ecodeme_with_prob           [0,5,3,2,4,1,      =      4.14E-6]
sends_mig_backward_to_ecodeme_with_prob [7, = 4.48E-6] sends migrant in forward to
[3] 0.91;28.62:19479.0 with migration rate 0.097
…

NodeEvent_Incease_Demes_In_Forward
NodeEvent_Incease_Demes_In_Forward 3423 [6] -19.54;-18.84:907.0 appeared from [6] -
21.45;-15.85:33156.0

NodeEvent_Splits_By_Colonization_Binary
NodeEvent_Splits_By_Colonization_Binary 5487 [4] 55.32;22.27:8104.0 merges backward
with [7] -3.44;-42.21:5842.0
migration decays in forward with min migration = 1.13E-7 max_migration = 6.23E-4
```

## Validate Models

Once we have obtained a model from the GP4PG simulations we then proceed to validate the model by comparing the j-4SFS of the simulations obtained from this model with the observed data. To do so, we use the Validate.java class implemented in the GP4PG_suite. In order to execute the Validate.java, first we have to make it the principal main class in the Java project (https://stackoverflow.com/questions/1635136/how-to-set-default-main-class-in-java) and then build the following folders in the *main_folder* :

1. main_folder
    a. fastSimcoal_0
    b. fastSimcoal_1
    c. …
    d. fastSimcoal_n
    e. fastSimcoal_Validation
    f. model_names.txt

In the *main_folder* we will have the arlequin ModelX.par files for each of the best models in any independent run of the GP4PG. These par files are the ones that the Validate.java will read to compute the validate simulations and calculate the 4-jSFS with the following code:

```
public class Validate {/*
     */
    public static void main(String[] args) throws Exception {

        // PARAMETERS TO FILL!
       String folder_fastSimcoal2 = "/path/to/fastSimcoal2_Validation/";
        String name_fastSimcoal2 = "fsc26";

        int number_of_sampled_populations = 4 ;

        // END PARAMETERS TO FILL
        // File with the "best(s)" models,

        String par = "name_of_file.par";

        // For each par, do n simulations
        RunFastSimcoal2 run = new RunFastSimcoal2();
       ComputeSFSFourPopsByWindow                cms                =                new
ComputeSFSFourPopsByWindow(number_of_sampled_populations);

       String model_name = par.substring(0, par.lastIndexOf("."));
     WriteFile  output  =  new  WriteFile("/path/to/output"  +  "sfs_"  +  model_name  +
 "_.txt");
        for (int rep = 0; rep < 1000; rep++) {
               run.run(folder_fastSimcoal2, name_fastSimcoal2, par, 60000);

               double[]      sfs      =      RetrieveSFSFromFastSimcoal2File.get_SFS
 (folder_fastSimcoal2, model_name);

               sfs = cms.sfs_by_four_pops(sfs);

               output.print(model_name);

               for (double s : sfs) {

                      output.print(" " + s);

                }

               output.println("");
       }
   }
}
```

Once we have obtained the 1000 4-jSFS from the 1000 simulations of the best model from the GP4PG_suite we then can compare it with observed data by means of a Principal Component Analysis.

## References

Koza JR. Genetic programming as a means for programming computers by natural selection. Stat Comput. 1994;4:87–112

Mehrabian AR, Lucas C. A novel numerical optimization algorithm inspired from weed colonization. Ecol Inform. 2006;1:355–66.

Misaghi M, Yaghoobi M. Improved invasive weed optimization algorithm (IWO) based on chaos theory for optimal design of PID controller. J Comput Des Eng. 2019;6:284–95.