5/8/21

Creating Infrastructure For Testing Demographic Inference Methods on Simulated Data

Project by: Kai Golan Hashiloni

Instructed by: Dr. Ilan Gronau

Contents

1. Introduction

In [1], the PopSim consortium described a multi-population experiment on simulated genomic data, in which the researchers demonstrated the performance of several demography inference methods. Our goal in this project was to allow one to take a demographic model and run experiments on a series of variants of this model. Since FastSimCoal (fsc26) [4] is a very popular and flexible method, we decided to focus our modifications on experiments that test its performance. The framework we develop here should be easy to extend to additional demographic inference methods.

As a basis for development, we used an analysis directory, named popsimenv [2], which was adapted by Itamar Topol from the original analysis directory [3] describing the experiments in [1]. This directory allows one to run experiments on a customized docker for testing of stdpopsim demography inference methods using the stdpopsim framework. The custom docker and directory contain all necessary files, packages, etc, thus it can be used on any machine and operating system.

To achieve our objectives I explored the mechanism of snakemake and the original snakefile in particular. I understood the structure of an experiment directory to be able to insert changes and later on create a new structure and snakefile. The process of deeply understanding how it was created and meant to be run, included several meetings with Itamar Topol and an online workshop with some of the creators and managers of [3].

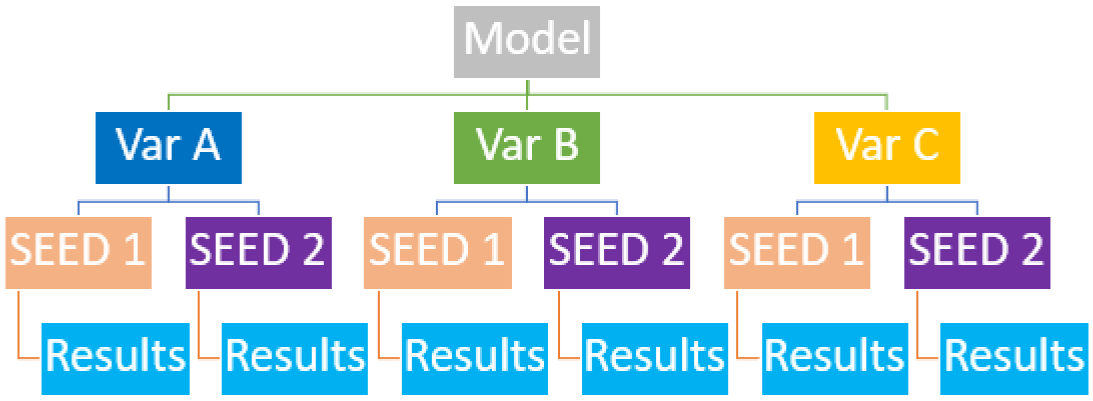
Our conclusions are:(1) with the new infrastructure one can efficiently test the fsc26 method on different model variations, and expanding it to more features and methods is pretty easy. (2) Our experiments found a potential noise in the inference of divergence times of the fsc26 inference method. This observation calls for further examination in follow-up studies that involve more extensive experiments.

1.1. Main Objective

To achieve our goal we wanted to build an infrastructure for running experiments that are as efficient, generalized, and neat as possible. The Original Snakefile used in [2] and [3] was pretty hard-coded because it was meant to generate specific results to demonstrate the tools it uses. To allow the usage of the Snakefile as a black box, it is important to understand its structure deeply and modify it carefully, which was a pretty time-consuming process. The idea which we had in mind while constructing the new infrastructure was to allow one to take a certain standard demographic model, like “OutOfAfrica_3G09” [8], and allow the user to define variants of this model, and then run replicates of each variant. In addition, the procedure should generate summaries of those experiments across the different variants and replicates. The involves the following main tasks:

1. Designing a modular framework to define model variants
2. Modifying the pipeline outline (Snakefile) accordingly
3. Modifying different components of this pipeline / add new ones

Here we provide a schematic explanation of the outline of an experiment, in which there are 3 variants of the model and 2 seeds (replicates) for each of them. In total, we get 6 sub-experiments.



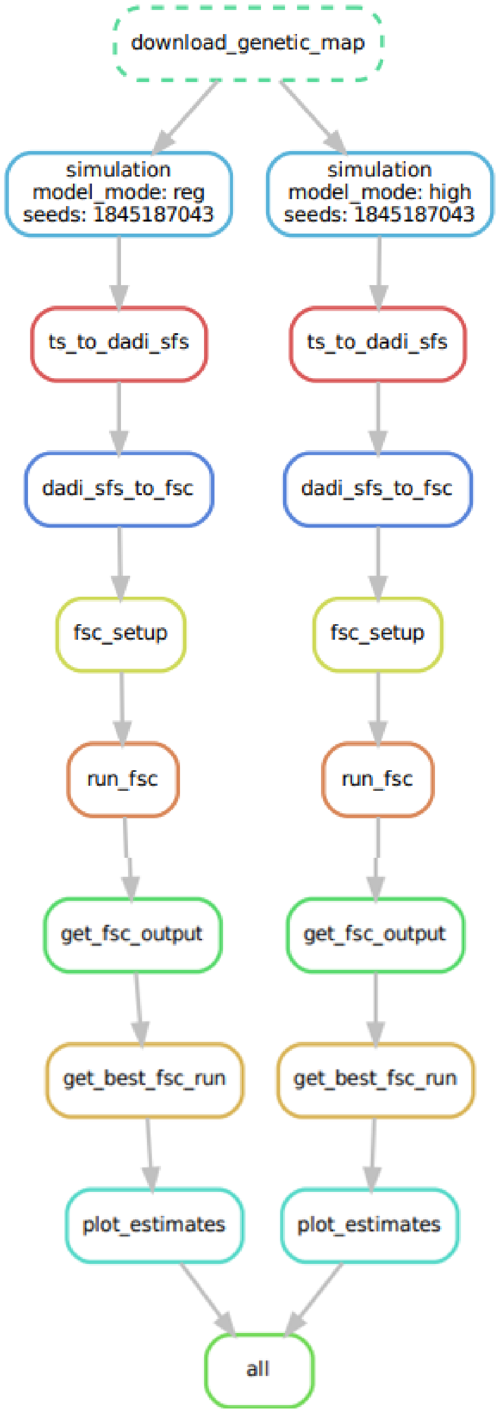
2. The Analysis Pipeline

2.1. Specifying a pipeline using Snakemake

“The Snakemake workflow management system is a tool to create **reproducible and scalable** data analyses” [7]. Snakemake is a python-based mechanism that allows the user to run a pipeline on several variants of some object / different inputs. It is a powerful tool to write pipelines meant to deal with the file system: reading, creating, and modifying files.

A Snakefile’s structure is:

- A block of code part on top, which is executed every time the snakefile is invoked
- Sequence of rules
  - Source(s) / target(s) can be defined for each rule, which implies a dependency relation between rules that can be described using a directed acyclic graph, (DAG)
  - Each rule may be implemented in python or bash
- An optional external configuration file.



2.2. The Snakefile

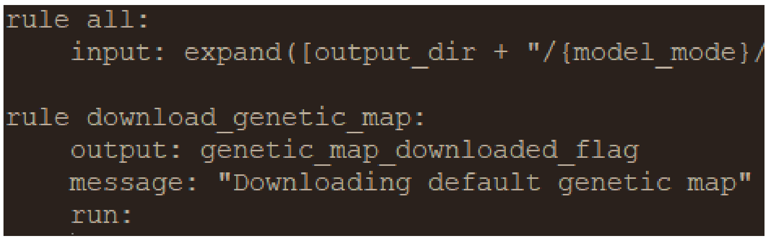
Every experiment requires a running directory of its own, as will be described later, but the heart and brain of the experiment are the Snakefile. It runs a sub-experiment for every variant of the model defined by the user in the config.json file - simulating data, running the inference method, analyzing, and plotting.

As mentioned above, we’ve put much effort into building an infrastructure for running a new experiment that will be much easier to use and efficient for the user. One of our most important milestones was to detach the user from the Snakefile so that they no longer have to change parts of it and can use it as a black box. We achieved this by modifying the Snakefile to make no explicit assumption about the demographic model that it simulates. We’ll now briefly explain the structure of the new Snakefile while referring to important points that were changed from the original structure. We show here a DAG of jobs representing an experiment with two variants of the model, with one replicate each. The figure of this DAG is an additional output of a Snakefile, which helps visualize its workflow and describe the rules’ dependencies.

Pre Rules

This is the part before the rules definition, the first thing it contains is the imports section. We added several packages \ functions here that we needed, some of which are standard libraries, and some are custom libraries that we wrote. In the process of generalization, we separated several scripts from the local experiment folder to a more generic one, so we needed to add their paths to sys.path to import them. After the packages are imported, the JSON configuration is loaded, and then its attributes are used to set various variables, including the random seed. A key variable used by the pipeline is the model_modes, which defines the variants of the model, and is described in detail in the config.json chapter.

Rule all



```
rule all:
    input: expand([output_dir + "/{model_mode}/

rule download_genetic_map:
    output: genetic_map_downloaded_flag
    message: "Downloading default genetic map"
    run:
```

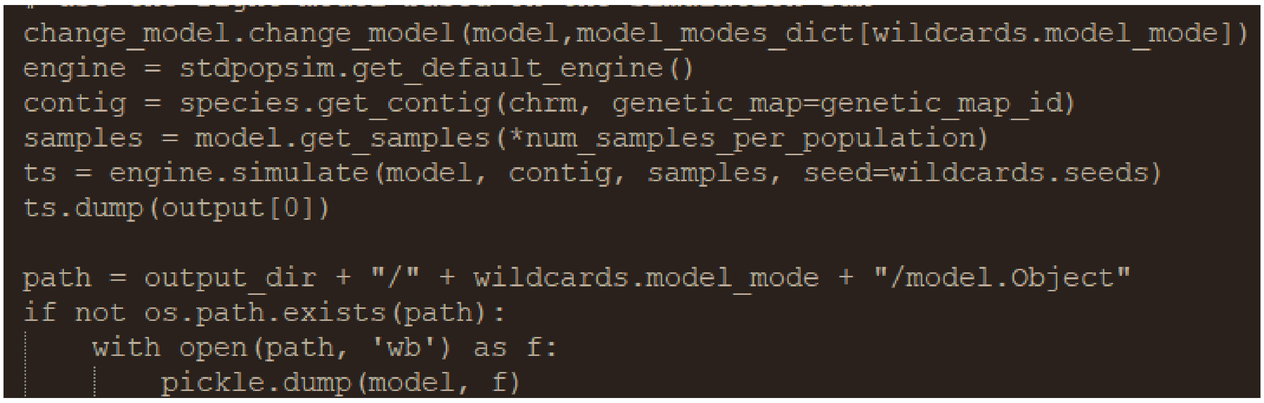
Rule all is the standard root of every Snakemake pipeline. Usually, and as we did here it contains only an input, which is the desired output of the whole Snakefile or the last output to be made. We defined this input as the results graph of all variants and all replicates. Snakemake builds the DAG of jobs following the dependencies defined by the input and output parameters of the rules. We defined this multi-input for the rule all, which again, is the output of the Snakefile, using the expand Snakemake function, which creates a list of elements by a pattern and wildcards. For example, the (part of the) pattern here is `output_dir +"/{model_mode}/…` and the wildcards will all replace {model_mode}, each at a time.

Rule donwnload_genetic_map

This rule happens only once, no matter how many replicates/variants you run in the experiment. Its purpose is to download the genetic map

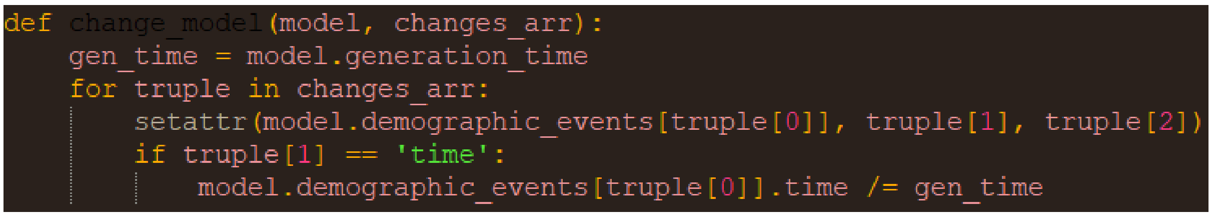
Rule simulation

This rule generates the simulated genome data, based on the variant of the model. The first line in the screenshot contains the usage of the mechanism we built for an efficient changing of the model, based on the user's will. They decide what changes should be made for each variant and specify them in the config.json. The modifying mechanism is implemented by the change_model.py, which takes 2 arguments: the model to modify and the changes to be



```
change_model.change_model(model,model_modes_dict[wildcards.model_mode])
engine = stdpopsim.get_default_engine()
contig = species.get_contig(chrm, genetic_map=genetic_map_id)
samples = model.get_samples(*num_samples_per_population)
ts = engine.simulate(model, contig, samples, seed=wildcards.seeds)
ts.dump(output[0])

path = output_dir + "/" + wildcards.model_mode + "/model.Object"
if not os.path.exists(path):
    with open(path, 'wb') as f:
        pickle.dump(model, f)
```

made. The program iterates over the changes and inserts each of them into the model. For now, this program contains only a single modification function, which is applicable for the “OutOfAfrica_3G09” model [8], that we used in our experiments. This function was written assuming it will be used on this specific model, thus it **may** suit more models, but it requires a deeper examination. The model’s inner structure, types of demographic events, etc, must be studied to determine if this function can be used on it, otherwise, you should write a new function based on your model.



```
def change_model(model, changes_arr):
    gen_time = model.generation_time
    for truple in changes_arr:
        setattr(model.demographic_events[truple[0]], truple[1], truple[2])
        if truple[1] == 'time':
            model.demographic_events[truple[0]].time /= gen_time
```

Another modification we made, which can be seen in the last part of the snapshot is to store the modified model for every variant as a binary file, using the pickle package. Later on in the Snakefile, we read these object files and pass them to the plotting rule, to be able to compare the inferred values (of fsc26) with the simulated values (“truth”). This is the only required “long-range communication” in the pipeline, for all other data transfers we used the rules input\output mechanism. We could do that because the rules we wanted to communicate are immediate predecessors of one another (directly connected in the DAG). We’ve found the pickle option to be an elegant and efficient solution, which enables us the desired “long-range communication”.

Rule ts_to_dadi_sfs

This rule takes as an input the trees that rule simulation generates, and creates dadi formatted intermediate files, that fsc26 needs later. You can see another way to create dependencies between rules here, which is more explicit. It's being done by one of dadi’s functions.

Rule dadi_sfs_to_fsc

The Snakefile now does another phase of the preprocessing of fsc26, here it uses an fsc26 function.

Rule fsc_setup

Here the Snakefile takes some general structure files of fsc26 and modifies them to suit the current experiment.

Rule run_fsc

This rule runs fsc26 10 times, this is how the original Snakefile was made, to reduce errors. Potentially, this should be done in a loop, but it is manually implemented instead and we decided to keep it as is.

Rule get_fsc_output

This rule extracts the result of the 10 runs of fsc26 and creates a sorted list of them in one text file.

Rule get_best_fsc_run

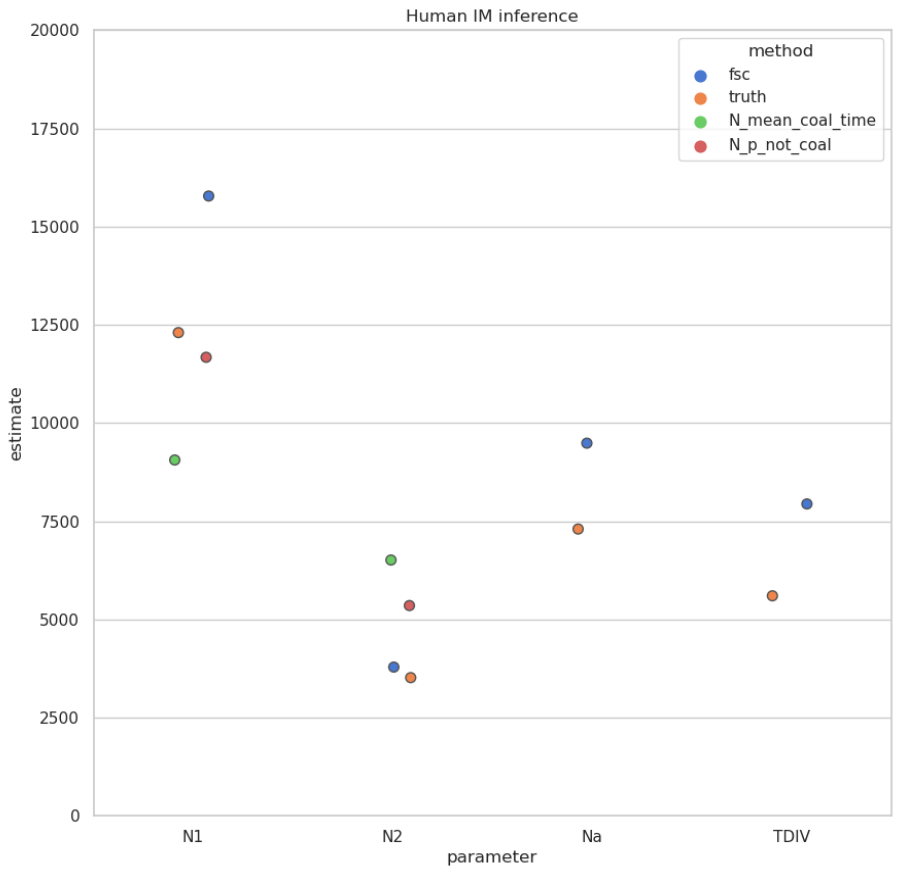
As mentioned above, the Snakefile takes the best estimate out of 10, so it takes the first row of the sorted results txt file.

Rule plot_estimates

```
path = output_dir + "/" + wildcards.model_mode + "/model.Object"
with open(path, 'rb') as f:
    cur_model = pickle.load(f)

params = params_extractor.params_extractor(cur_model)
plots.plot_fsc_results_human_IM_general(input[0], output, simulated_genome_length, params)
```

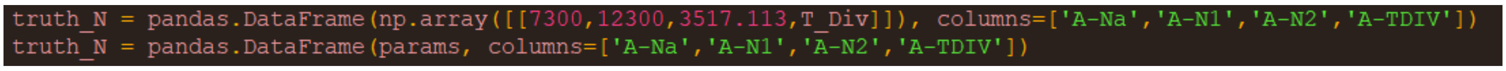
This rule plots 2 graphs for each replicate and of each variant of the model in an experiment separately, that is it will produce the desired graphs (shown below) in every subdirectory of the experiment. As explained in the rule simulation we store the model as a binary file, and here we read it again to the Snakefile. We need it here to pass the simulated model to the params_extractor to get the expected values of the demographic parameters for the graphs.

One graph, which we focused on in our experiment, shows the estimates of 4 values of the model. For each of them, it displays the “truth” values and the prediction of fsc26. For the population sizes (N), the plot also contains two expected values based on different types of averages, since the simulated model contains changing population sizes in time. The Other graph shows an estimate of fsc26 of the migration rates, but we didn’t focus on it this time.



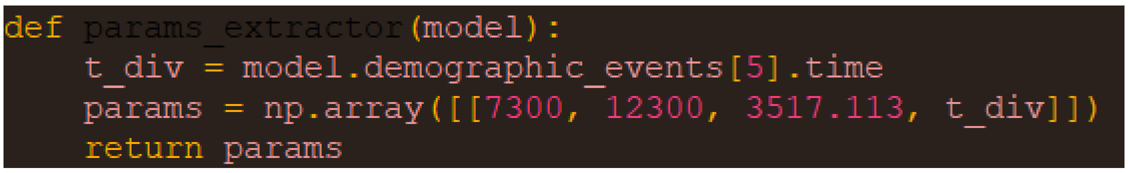
2.3. The program params_extractor.py

The plotter shows the result of each replicate of each variant of the model in a separate plot. The original plotter defined the truth_N variable, which contains the truth values, by writing the numbers hard-coded, as can be seen in the first line of the snapshot below. To make the plotter more generic, we replaced those numbers with a variable, params (2'nd line in the snapshot), which the plotter gets as an argument, enabling plotting if different "truth" values with the same plotter. We pass the params variable to the plotter function through the Snakefile, where we extract these parameters with the local **params_extractor** function, defined in the python script params_extractor.py.



```
truth_N = pandas.DataFrame(np.array([[7300,12300,3517.113,T_Div]]), columns=['A-Na','A-N1','A-N2','A-TDIV'])
truth_N = pandas.DataFrame(params, columns=['A-Na','A-N1','A-N2','A-TDIV'])
```

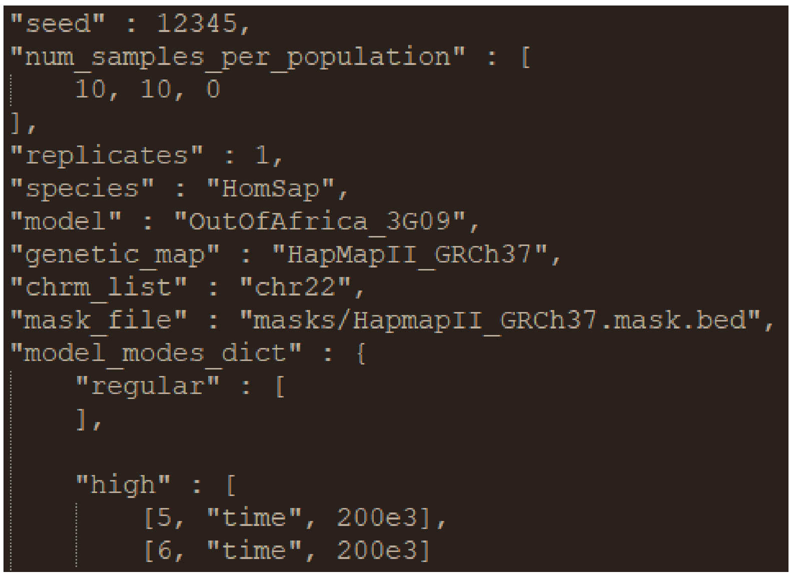
The snapshot below displays the internal structure of the params_extractor function in the params_extractor.py script. The parameter t_div is directly extracted from the model object that is loaded from the stored object file and passed to this function by the snakefile. The other values are still hard-coded, thus, if you want to use (plot) different values, you should modify this script (function). We didn't focus on other values but t_tiv in our experiment so we used the numbers as in the original pipeline.



```
def params_extractor(model):
    t_div = model.demographic_events[5].time
    params = np.array([[7300, 12300, 3517.113, t_div]])
    return params
```

2.4. The JSON configuration file config.json

The config.json file and mechanism existed in the original directory, and it holds the configurable parts of the experiment, while the Snakefile contains the pipeline. The snapshot below shows the config.json file's inner structure in the framework we created, while the original one defines the same parameters but the "model_modes_dict". This modification enables the user to specify what variants should the model run with, where the rest of the configurations are defined.



```
"seed" : 12345,
"num_samples_per_population" : [
    10, 10, 0
],
"replicates" : 1,
"species" : "HomSap",
"model" : "OutOfAfrica_3G09",
"genetic_map" : "HapMapII_GRCh37",
"chrm_list" : "chr22",
"mask_file" : "masks/HapmapII_GRCh37.mask.bed",
"model_modes_dict" : {
    "regular" : [
    ],

    "high" : [
        [5, "time", 200e3],
        [6, "time", 200e3]
```

The parameter model_modes_dict describes the different variants of the model and is later parsed by the snakefile into a python dictionary. It contains the following components:

- The keys are the names of the variants, which are used as the names of the sub-directories of the experiment (displayed below). The user can specify as many variants (modes) as they like.
- The value associated with each key is a list of the model changes applied to this variant (mode). Each change is defined as a list of exactly 3 variables, in that order exactly:
    - <int> index - of the event in the model's demographic_events list, one has to be familiar with this list when defining the variants.
    - <String> attribute - the attributes of the event that needs to be changed (e.g. "time", "size", "population" - each event may have different attributes).
    - <float> value - the new value of the attribute.

3. Running an experiment

In order to run a new experiment using the new infrastructure, one has to do the following:
** all described in the main README.md of the repository on Git-Hub.

First time only:

1. Download Git [5] and Docker [6].
2. Clone the popsimenv directory [2] from git.
3. Inside the popsimenv directory, in the command line run the following command to build the image docker using the Dockerfile:
   `docker build -t popsimenv-image .`

Every time:

4. Open Docker Desktop app.
5. Inside popsimenv directory, run the container interactively with the src folder as shared volume: (this is a one line command)
   ```
   docker run --rm -t -i --name popsimenv --mount
   type=bind,source=%cd%/src,target=/code/src popsimenv-image //bin/bash
   ```

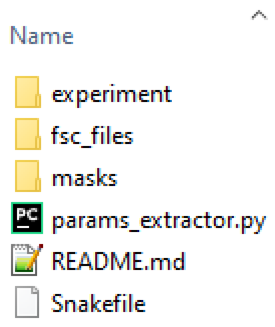
   ** For windows users, Mac has a bit different commanded, written in the README.md

6. Activate the conda environment:
   ```
   conda activate popsim_env_test
   ```

7. The shared volume of the image and your file system is (through the image) /code/src

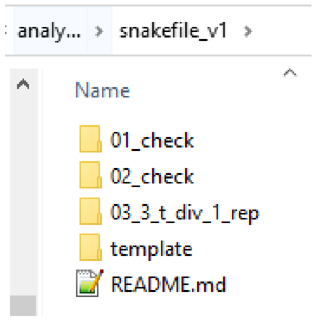
8. Copy the template directory src/analysis/snakefile_v1/template into a working directory inside src/analysis/snakefile_v1/ with an informative name. Example:
   a. Inside "src/analysis/snakefile_v1" run:
   b. `cp -r ./template/ ./myNewExperiment`

9. The contents of your new directory should now look as follows:
10. Modify the config.json file in the experiment subdirectory, based on the desired experiment configuration (see details here).
11. Modify params_extractor.py if needed (see details here).
12. Run in terminal (in the pasted directory) :
    `snakemake -j 1 --config config="experiment"`
    Wait for results, it takes up to several hours.
13. Create a README.md that documents this experiment inside the pasted directory.

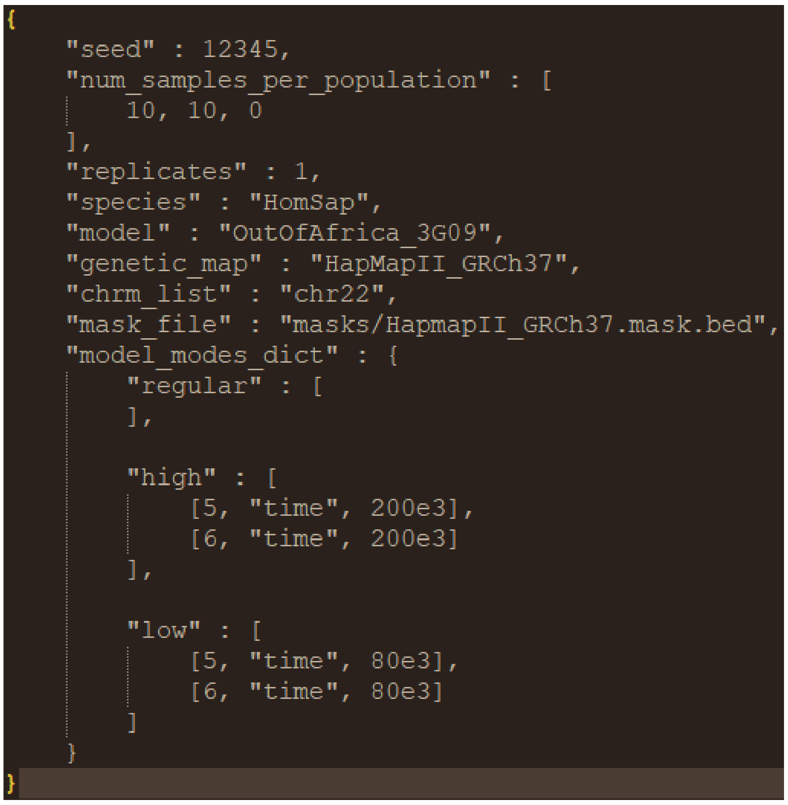

Name
experiment
fsc_files
masks
params_extractor.py
README.md
Snakefile

4. Example Experiment

An example of an experiment, as explained in Running an experiment, in our new framework, using Snakefile_v1. First, we copied the template directory into a working directory named “03_3_t_tiv_1_rep” - because it will run 1 replicate of 3 variants, each with a different T_DIV.

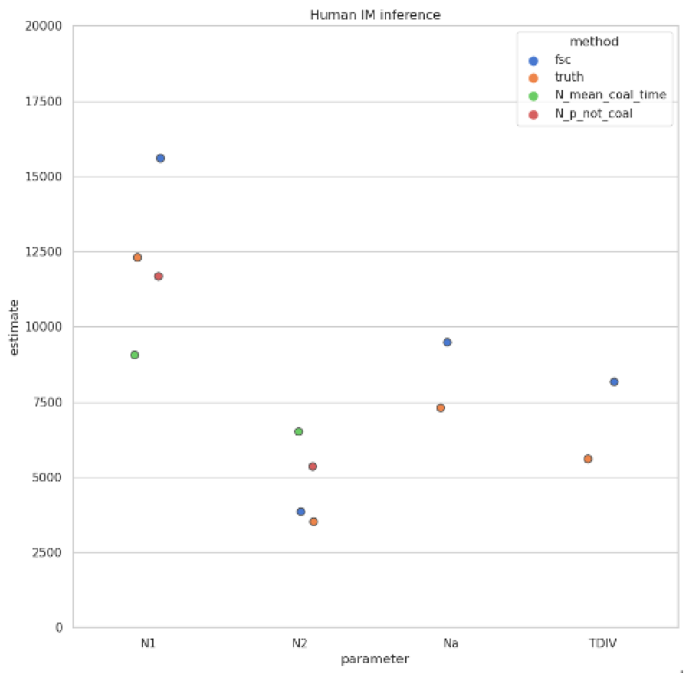


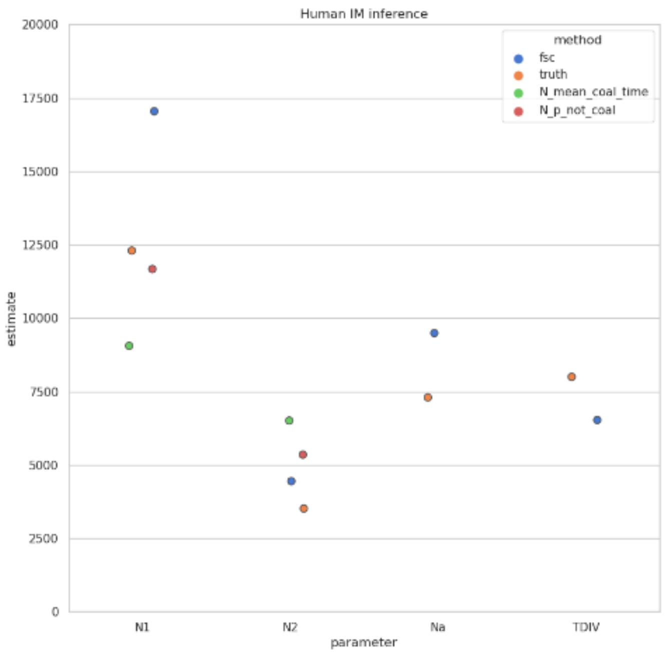
There is no need to modify the program params_extractor.py for this experiment, so we only need to modify the configuration file config.json inside the “experiment” directory. In the snapshot below you can see the config file we use, and exactly how we define this experiment. We specify the species to work with, the model (this model assumes there are two populations), what chromosome to simulate, etc. note that we create one replicate for each of the three variants of the model - in each with we change **only** the time the two populations divergence (“T_Div”). We have a regular variant with no modifications, one with a higher T_Div and one with a lower T_div

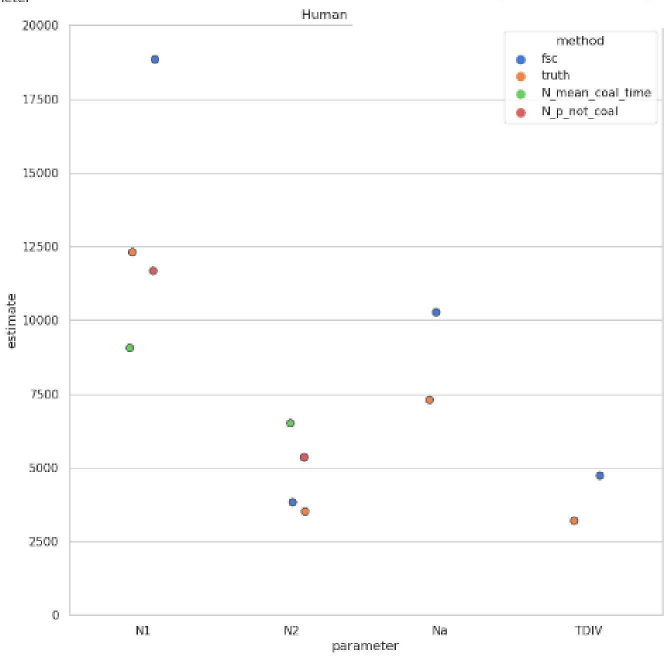


```
{
    "seed" : 12345,
    "num_samples_per_population" : [
        10, 10, 0
    ],
    "replicates" : 1,
    "species" : "HomSap",
    "model" : "OutOfAfrica_3G09",
    "genetic_map" : "HapMapII_GRCh37",
    "chrm_list" : "chr22",
    "mask_file" : "masks/HapmapII_GRCh37.mask.bed",
    "model_modes_dict" : {
        "regular" : [
        ],

        "high" : [
            [5, "time", 200e3],
            [6, "time", 200e3]
        ],

        "low" : [
            [5, "time", 80e3],
            [6, "time", 80e3]
        ]
    }
}
```

With this configuration, we run the Snakemake and get the results, which are two graphs for each replicate and variant, as described in the plot_estimates rule. We focused on the graph that visualizes the T_Div (on the scale of the number of generations ago) and the populations' sizes estimation, named "estimates_N_tdiv_fsc". In particular on the values of T_Div, which are given by two points on each graph: the orange one is the "truth" value, the value defined in the model's variant, and the blue is the estimation of fsc26. On the top-left shown the resulted graph of the "regular" variant, on the top-right, results for the "high" variant, and in the bottom - results for the "low" variant.

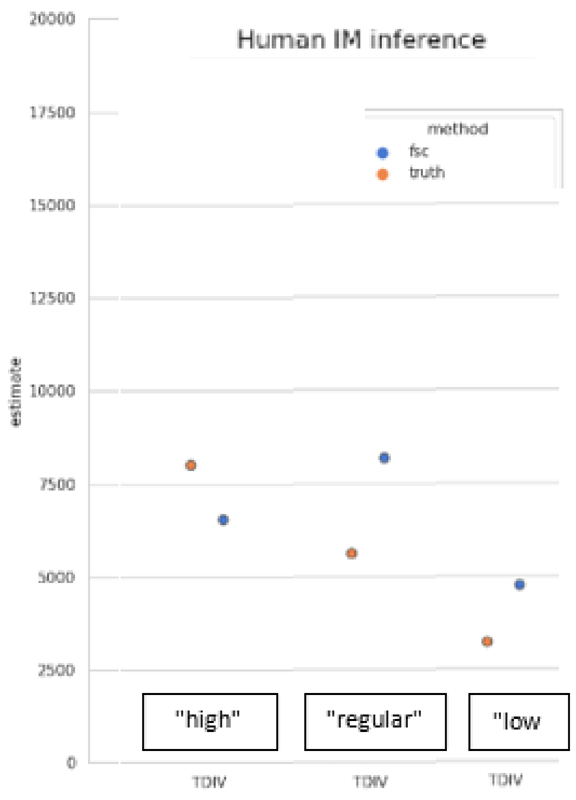






As we focused on fsc26 estimation of T_Div we show below a concentration of the 3 different variants' results of this parameter. Note that the estimate for the "regular" variant was higher than the one for the "high" variant. On the other hand, the estimate for the "low" variant was lower than the two others. Another interesting point is the influence of changes of T_Div on the other parameters. For example, N1 becomes larger as T_Div goes down, despite not being changed, so it's possible that they are correlated.

What we observe here is that fsc26, in this experiment, is not completely monotone with changes of T_Div, as we'd expect it to be. It raises a question about the accuracy of fsc26, we suggest this optional examination for this topic.



Note that one can run a simple variant of this experiment by creating another experiment directory (e.g. experiment2), in the current working directory, that only has a config.json file. In the config.json apply the desired change, and re-run with:

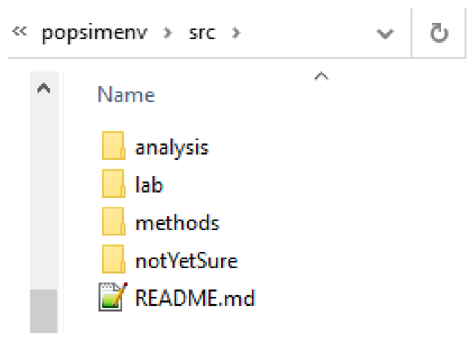
```
snakemake -j 1 --config config="experiment2"
```

There is no need to re-copy the entire directory (with fsc_files, etc).

5. Structure of src directory

As mentioned above, the popsimenv working directory is designed to run a Docker container on the user's machine. The container contains all necessary packages and environment settings for running the experiments. The piping mechanism between the container's filesystem and host OS filesystem is through a shared volume, the src directory. It means that to pass files to and from the container, the user must do it through this folder, which is thus practically the working directory for the analysis. All other files in the popsimenv directory (besides the src directory) are for the container set-up and don't need to be touched.

The structure of the original directory was great to begin with, but it was not scalable and parts of the pipeline worked with the assumption that some files\scripts are located in the Snakefile directory. To make the entire pipeline more scalable and easy to work with, we decided to reconstruct the entire working directory, src, to meet our needs, and also to help future potential users. The new structure is described here and is also documented in the directory.



analysis
lab
methods
notYetSure
README.md

5.1. analysis

This directory contains files that specify the specific experiment one wishes to execute. It is also the base directory where the analysis takes place. The original experiment directory is there, and so is "Snakefile_v1" which contains previous runs of experiments made by us, with the most updated Snakefile.

5.2. methods

Contains directories for packages, methods, and tools that can be used in the experiments generically. The sub-directories are:

1. Inference methods directories: dadi, fsc26, G-PhoCS, smc++
2. Stdpopsim - contains some functions related to this package.
3. Plot - contains plotting functions, if one wants a specific plot, this is the place to define a function for it.

5.3. notYetSure

Contains files and scripts that appeared in the original analysis directory, but I couldn’t find any use for them in the experiment we designed here.

1. I think those can be removed:
   a. requirements.txt file
   b. extern directory
2. The p_not_coalesce.py contains an example for printing a model for debugging and it might come useful at some point.
3. The n_t directory is a complete experiment that may be explored and used for expansion later.

6. Further Research

This chapter describes several options for upgrading the infrastructure or further research. We didn't implement those ideas due to the time and scope limitations of the project.

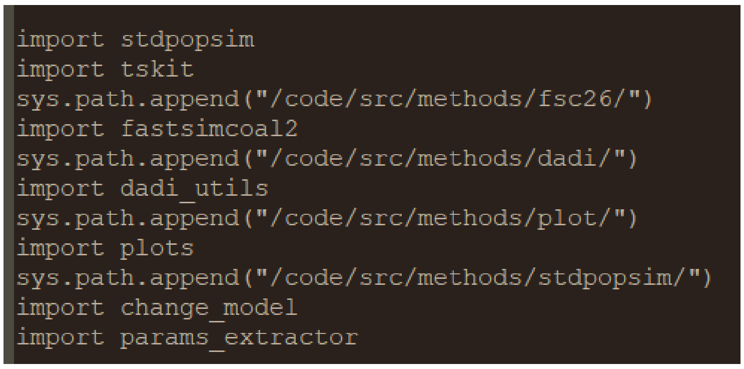
6.1. Snakefile_v2

One of our goals was to make the process of running an experiment more neat and generalized. Here we present a way of making it better.
The current snakefile (v1) doesn't need to be opened by the user, as mentioned above. Thus, it's possible and makes sense not to have it in every experiment's directory, but to have only 1 snakefile, which will be run from different locations and for different purposes. In particular, it should be located in the "methods" directory, described above.
But, the snakefile still contains code parts that assume it is local, which need to be treated, such as:

imports

Imports - 'import params_extractor' is a local import, and thus will have to be changed, for example by using the "output_dir" variable, or passing the location as a parameter from the command line \ config.json file.

1. It is possible that the whole imports section can be improved, instead of using the clumsy mechanism of sys.
2. It is possible that there isn't really a need for a local params_extractor.py file, and that can be generalized higher in the directories hierarchy, like the plotter.

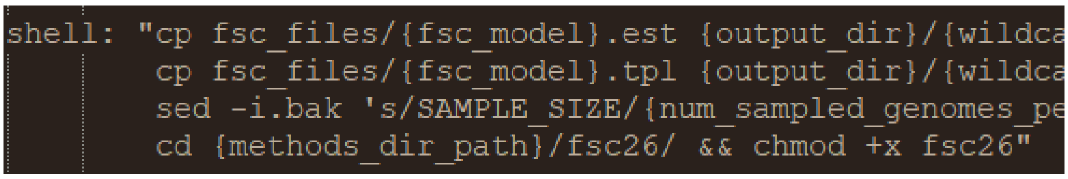


```
import stdpopsim
import tskit
sys.path.append("/code/src/methods/fsc26/")
import fastsimcoal2
sys.path.append("/code/src/methods/dadi/")
import dadi_utils
sys.path.append("/code/src/methods/plot/")
import plots
sys.path.append("/code/src/methods/stdpopsim/")
import change_model
import params_extractor
```

Masks & fsc_files

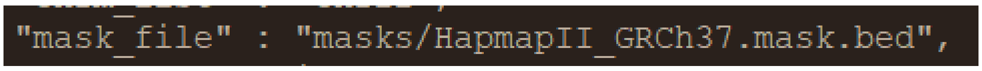
Those are files the snakefile uses at several points, and currently, they are a part of the running directory, although they are general and there is no reason to copy them over and over. Here are some of these code parts, which require adaptations:

1. Those bash commands contain relative paths, that assume those files are in the calling directory.



```
shell: "cp fsc_files/{fsc_model}.est {output_dir}/{wildca
        cp fsc_files/{fsc_model}.tpl {output_dir}/{wildca
        sed -i.bak 's/SAMPLE_SIZE/{num_sampled_genomes_pe
        cd {methods_dir_path}/fsc26/ && chmod +x fsc26"
```

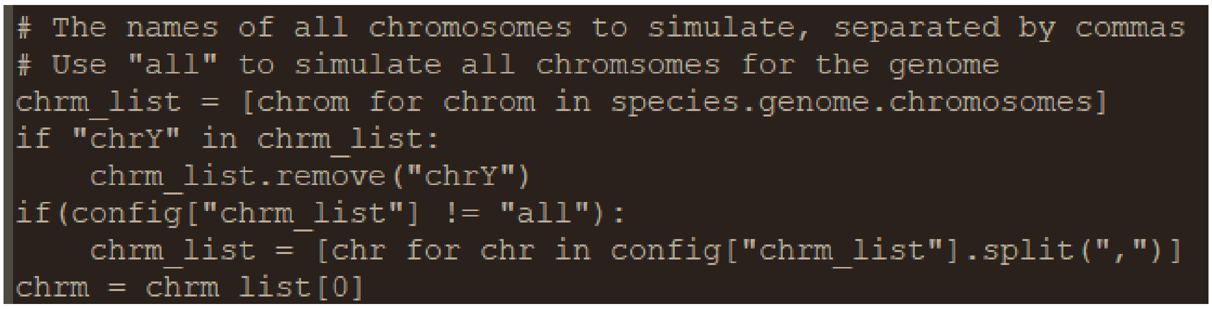
2. Here we read the mask file in the config.json file, it can easily be changed to something more general, using absolute paths.



```
"mask_file" : "masks/HapmapII_GRCh37.mask.bed",
```

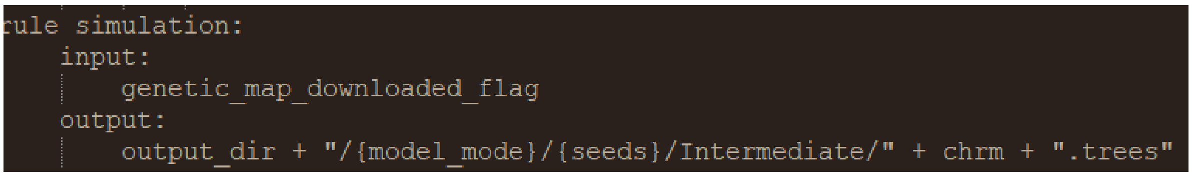
6.2. Chromosomes

For now, the mechanism allows running with a single chromosome, as shown here. We chose this approach because it made things considerably simpler, and we didn't want to put too much effort into this issue in the initial stage of development. The original snakefile had the ability to run with more than one chromosome, but we had problems with passing the "chrms_list" as a wildcard in the Snakefile, in a way that won't cause trouble, such as models slipping to other branches of the snakefile of different models.



```
# The names of all chromosomes to simulate, separated by commas
# Use "all" to simulate all chromsomes for the genome
chrm_list = [chrom for chrom in species.genome.chromosomes]
if "chrY" in chrm_list:
    chrm_list.remove("chrY")
if(config["chrm_list"] != "all"):
    chrm_list = [chr for chr in config["chrm_list"].split(",")]
chrm = chrm_list[0]
```

If one wants to go back to the "chrm_list" mechanism, note that it is needed to look for all the places it takes place in, for example:



```
rule simulation:
    input:
        genetic_map_downloaded_flag
    output:
        output_dir + "/{model_mode}/{seeds}/Intermediate/" + chrm + ".trees"
```

6.3. Summary Plot

Another idea is to create a single plot that contains information about all the experiments of each variant (mode). This plot will display the results of all replicates ("seeds") of the variant in a single graph. For example, showing all "T_Div"s, this graph can help with a closer examination of an inference method.

6.4. dadi & smc++ & G-PhoCS

More inference methods in the same snakefile. This is another thing we omitted to make things easier when changing the snakefile frequently but appeared in the original snakefile. Adding them back shouldn't be very problematic, as long as you follow the new snakefile structure and write the code (modify the existing one) accordingly.

6.5. A closer examination of fsc26 accuracy

We noticed that fsc26 might have some noise and it doesn’t accurately estimate some parameters of the model when those are changed. Arguing such a claim on an inference method (like fsc26) required proof, which can be achieved by running a large number of experiments and analyzing the average behavior of the method. In our case, one should define an experiment with several variants of the model (changing only the attributes needed for this examination) and set a large number of replicates. For each variant, calculate the average results and see if the method was able to estimate the values that were changed accurately. This comes up as a solution to the fact that the methods may have quite a large standard deviation and contain noises.

7. What I’ve learned

1. Code project directory management - I leapedJSON in the way I use and manage a project directory. Before, I had experience working on a single \ few scripts in a Sandbox simple directory where I’m the only user. This project required a deep understanding of many scripts, files, and methods of different types, and I had to create a clear hierarchy to maintain order while taking care of documentation for later users.
2. Research - As mentioned in the last point, until now all of my experience in programming was in “safe zones”, such as HW, and the instructions were clear, plus I knew what the goals and I had colleagues to discuss with and teachers for explaining subject \ ambiguous parts. However, in this project, I had only Ilan to talk to, but the main difference is that neither of us had answers \ clear references to compare our “result” to, thus I had to explore topics on my own and be very thorough. We had to consistently ask ourselves what should be the next step, what's more interesting, etc’.
3. Snakemake mechanism - it was a new concept for me, and now I have a great understanding of how it works and its capabilities, I can use it in other projects I’ll work on.
4. Docker - Although I still don’t fully understand how to build a Docker, I’m much more familiar with it now. This is a very important tool \ trend these days instead of the clumsier Virtual Machines.
5. Github - Another new tool for me, which I’m already using in other projects I’m working on, as a collaborator or menager.
6. Stdpopsim & Msprime genome simulating package - I discovered a whole new region of research and work in Computational Biology, and had the chance to learn about it hands-on with personal guidance. I have much more to learn on this topic, but I clearly have a basic understanding of the fundamentals in it.
7. JSON configuration files - I didn’t know about the existence of this format, although it is widely used and an important feature in programming.
8. Pickle package.
9. Python & Linux - I practiced those and learned more about them.

8. Bibliography

[1] Adrion, J. R., Cole, C. B., Dukler, N., Galloway, J. G., Gladstein, A. L., Gower, G., Kyriazis, C. C., Ragsdale, A. P., Tsambos, G., Baumdicker, F., Carlson, J., Cartwright, R. A., Durvasula, A., Gronau, I., Kim, B. Y., McKenzie, P., Messer, P. W., Noskova, E., Ortega-Del Vecchyo, D., Racimo, F., … Kern, A. D. (2020). A community-maintained standard library of population genetic models. *eLife*, *9*, e54967. https://doi.org/10.7554/eLife.54967

[2] https://github.com/itatop/popsimenv

[3] https://github.com/popsim-consortium/stdpopsim

[4] Download: http://cmpg.unibe.ch/software/fastsimcoal2/
Manual: http://cmpg.unibe.ch/software/fastsimcoal2/man/fastsimcoal26.pdf
Citation:

Laurent Excoffier, Matthieu Foll, fastsimcoal: a continuous-time coalescent simulator of genomic diversity under arbitrarily complex evolutionary scenarios, *Bioinformatics*, Volume 27, Issue 9, 1 May 2011, Pages 1332–1334, https://doi.org/10.1093/bioinformatics/btr124

Excoffier, L. (2013, October 24). *Robust Demographic Inference from Genomic and SNP Data*. PLOS GENETICS. https://journals.plos.org/plosgenetics/article?id=10.1371/journal.pgen.1003905

[5] https://git-scm.com/downloads

[6] https://www.docker.com/products/docker-desktop

[7] https://snakemake.readthedocs.io/en/stable/

[8] Gutenkunst RN, Hernandez RD, Williamson SH, Bustamante CD (2009) Inferring the Joint Demographic History of Multiple Populations from Multidimensional SNP Frequency Data. PLoS Genet 5(10): e1000695. https://doi.org/10.1371/journal.pgen.1000695