SodaPop

User Manual v1.0

Louis Gauthier
louis.gauthier@umontreal.ca

Adrian W.R. Serohijos
adrian.serohijos@umontreal.ca

Département de Biochimie,
Université de Montréal,
Québec, Canada

This manual accompanies the following mansucript:
L Gauthier, R Di Franco, A Serohijos, **SodaPop: A simulation suite for the evolutionary dynamics of asexual populations on protein fitness landscapes** *Submitted to Bionformatics*. Also available in *BioRxiv, 189142 (2017)*

HTTPS://WWW.SEROHIJOSLAB.ORG/SOFTWARE-TOOLS–DATA.HTML

Last revised 13 April 2018, for SodaPop v1.0

Contents

1. Introduction

Overview

Protein evolution is determined by constraints at several levels of biological organization. Random mutations have an immediate effect on the biophysical properties, structure and function of proteins. These same mutations also affect the fitness of the organism. However, the evolutionary fate of these mutations, whether they succeed to fixation or are purged, also depends on population size and dynamics. There is an emerging interest, both theoretically and experimentally, to integrate these two constraints in protein evolution. Although there are several tools available for simulating protein evolution, most of them focus on either the biophysical or the population-level constraints, but not both.

SodaPop is a new computational suite to simulate protein evolution in the context of the population dynamics of asexual populations. SodaPop accepts as input several fitness landscapes based on protein biochemistry or other user-defined fitness function. The user can also provide as input experimental fitness landscapes derived from deep mutational scanning approaches or theoretical landscapes derived from physical force field estimates. SodaPop is designed such that population geneticists can explore the role of protein biochemistry on genetic variation, and that biochemists and biophysicists can explore the role of population size and demography on protein evolution. This highly flexible tool takes as input a fitness landscape based on protein biochemistry and biophysics, experimental deep mutational scans or a phenomenological distribution of fitness effects. SodaPop was designed such that population geneticists can explore the role of protein biochemistry on genetic variation, and that biochemists and biophysicists can explore the role of population size and demography on protein evolution.

The main software is implemented in C++ and supported on Linux, Mac OS/X and Windows. Source code and binaries are freely available at https://github.com/louisgt/SodaPop under the GNU GPLv3 license.

2. Setup and installation

This section will take you through step-by-step installation.

> Note This program and simulation examples were developed for a Unix environment. For use in Windows, we strongly recommend installing a Unix-like environment such as Cygwin or MinGW.

SodaPop is a suite of command-line tools written in the C++ programming language using the C++11 Standard. The source code and compiled binaries can be downloaded from https://github.com/louisgt/SodaPop.

Users unfamiliar with command-line interface are referred to the *Annex: Using the command-line* found at the end of this manual.

2.1 Downloading SodaPop

Directly download SodaPop using the Git command. Alternatively, you can download its zip archive from https://github.com/louisgt/SodaPop.

2.1.1 Cloning via Git

Use the Git command to clone the original SodaPop repository to a local folder with

```
git clone https://github.com/louisgt/SodaPop.git my_folder
```

2.1.2 Zip archive

1. Go to https://github.com/louisgt/SodaPop. Click on the *Clone or download* button and select the *Download ZIP* option.
2. Move the archive to your preferred location and unzip its contents with

```
tar -zxvf [name of the zip file]
```

2.2 Installation

SodaPop comes with compiled binaries (executables). However, you are advised to compile the program on your system.

To compile all components of SodaPop, run

```
make
```

This uses the makefile to build the binaries **sodapop**, **sodasnap** and **sodasumm**, which can be executed from the folder they are located in. If you wish to install the program on your computer, run

```
make install
```

By default, the three binaries above will be appended to `/usr/local/bin`. You can change this option in the makefile by editing the content of the `$INSTALLDIR` variable.

To verify the installation step, run `make` again and the terminal will read

```
make:  Nothing to be done for ‘all’.
```

If the following error arises when compiling

```
error:  unrecognized command line option "-std=c++11"
```

your compiler is likely out-of-date. You can get a new version of gcc/g++ here. SodaPop is compatible with any version starting from gcc 4.7 or higherr.

3. List of command-line options

Command-line flags and parameters are used to toggle features, change program behavior and provide input parameters and files for a simulation. To view the available flags and their description, type

```
sodapop -help
```

Flags can also be added or modified in the *evolve.cpp* source file, using the TCLAP library API.

3.1 Numerical parameters

Number of generations `[-m, --maxgen]`
(Integer) The length of the simulation in number of generations.
(Default value) 10.

Population size cutoff `[-n, --size]`
(Integer) The target population size in each generation or the carrying capacity.
(Default value) 1.

Snapshot interval `[-t, --dt]`
(Integer) The time interval at which a population snapshot is saved. The minimum value is 1 and there is no maximum value. However, if *dt* exceeds the specified length of the simulation, only one snapshot is saved.
(Default value) 1.

3.2 Input files

Gene list file `[-g, --gene-list]`
(File) A text file that lists genes and the files that describe them (see **Section 4.2** and **Fig. 4.4**)

Population description file `[-p, --pop-desc]`
(File) The binary snapshot required to initialize the population in the simulation (see **Section 4.2**).

Path to gene library `[-l, --gene-lib]`
(Integer) The path to the folder containing the gene files.
(Default value) `/files/genes/`.

3.3 Fitness landscape

Input fitness landscape `[-i, --input]`

(File) This file defines effects of mutations for all genes in the simulation (See section 5). For a given gene, there is a matrix $M_{a,k}$ describing the change in fitness for a mutation to amino acid a in residue k. The index a is over all 20 amino acid listed in alphabetical order (from 'A' to 'Y'). In the file, this matrix is described by k rows of 20 values. The mutational matrices of all the genes are ordered in the file according to their gene ID.

By default, the values of in the matrix are assumed to be fitness changes (selection coefficients) from deep mutational scan (DMS) experiments. However, these matrices can contain changes to folding stability ($\Delta\Delta G_{folding} = \Delta G_{mutant} - \Delta G_{wildtype}$) or other biophysical properties.

Fitness function `[-f, --fitness]`

(Integer) Used to select the appropriate fitness function depending on the input type and the biophysical or biochemical property of interest. For a detailed reference of available functions, refer to section Fitness functions.

1 Folding stability.
2 Misfolding toxicity.
3 Combined metabolic output.
4 Multiplicative.
5 Neutral.
6 No mutations.

Simulation type `[--sim-type]`

(String) This flag is used to specify if mutation effects are interpreted directly as selection coefficients, or if they are effects on a particular property. By default, values are interpreted as selection coefficients. Alternatively, the type can be set to thermodynamic stability (ΔG), k_{cat}/K_M or another property implemented by the user.

s Selection coefficients.
stability Thermodynamic stability.
efficiency Protein catalytic efficiency.

Use gamma distribution `[--gamma]`

(Flag) This flag will draw fitness effects from a gamma distribution. The shape and scale parameters are specified with the alpha and beta flags, respectively.

Use normal distribution `[--normal]`

(Flag) This flag will draw fitness effects from a normal distribution. The mean and standard deviation are specified with the alpha and beta flags, respectively.

Alpha parameter `[--alpha]`
(Double) Value of the alpha parameter. If gamma distribution is used, this corresponds to the shape parameter. If normal distribution is used, this corresponds to the mean (μ).

Beta parameter `[--beta]`
(Double) Value of the alpha parameter. If gamma distribution is used, this corresponds to the scale parameter. If normal distribution is used, this corresponds to the standard deviation (σ).

3.4 Analysis and output

Output prefix `[-o, --prefix]`
(String) A prefix string appended to the filename of the output. If the specified directory does not exist, it will be created. If it exists, its current contents are overwritten. This parameter is not required, but strongly recommended since the default setting will create a directory named ?sim/? and could overwrite a previous run. You should always specify a new output prefix if working with multiple runs.

Initialize population from single cell `[-c, --create-single]`
(Flag) Using this flag will populate the initial cell vector with copies of a single cell. This can slightly speed up initialization for large-scale populations (10^6 or higher), at the expense of no clonal diversity. If your starting population is polyclonal, you should not use this flag.

Toggle automatic analysis `[-a, --analyze]`
(Flag) Using this flag will call analysis scripts after the simulation algorithm is complete. Refer to the Results and analysis section for detailed information.

Track mutations `[-e, --track-events]`
(Flag) Using this flag will save a log with all arising mutations during the course of the simulation (see **Section 6.4**).

Output format `[-s, --seq-output]`
(Flag) Using this flag you can specify the output format of population snapshots. If you don't need sequence information, use the short format as input/output operations will be lighter. If you'd rather have sequence data as DNA instead of protein, use the long format with DNA. See section 6.5 for details on output formats.
Default Long format.

0 Long format.
1 Short format.
2 Long format with DNA.

4. Quick Start

This chapter introduces users to SodaPop and its applications.

4.1 Program workflow

SodaPop's simulation algorithm is adapted from a Wright-Fisher model with selection (**Fig. 4.1**). Generations are discrete steps in which all N parent cells within the population give birth to a number of daughter cells proportional to their relative fitness. The number of progeny k is drawn from a binomial distribution with N trials and mean w, which is the the fitness of the parent cell over the sum of all cell fitnesses. The offspring become the parents for the next generation.

Initializing a simulation with SodaPop requires a combination of input files with a specific hierarchy (**Fig. 4.2**). The next section will take you through the definition and function of these files.

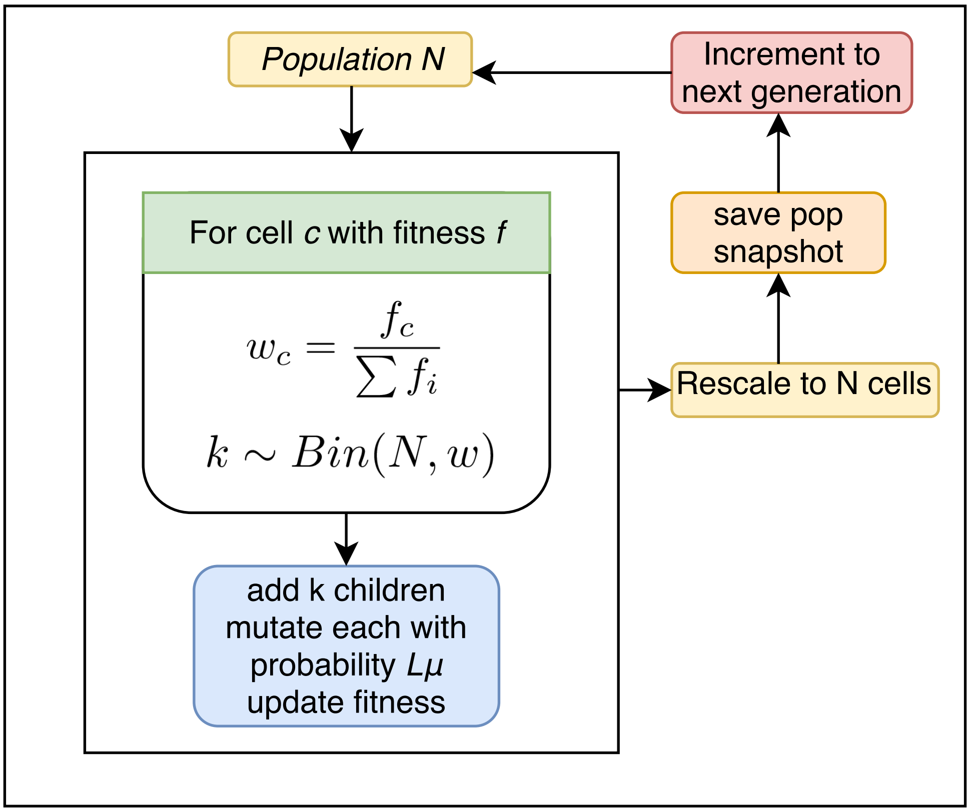

Population N
Increment to next generation
For cell *c* with fitness *f*
$w_c = \frac{f_c}{\sum f_i}$
$k \sim Bin(N, w)$
save pop snapshot
Rescale to N cells
add k children
mutate each with probability $L\mu$
update fitness

Figure 4.1: Illustration of SodaPop's evolutionary algorithm.

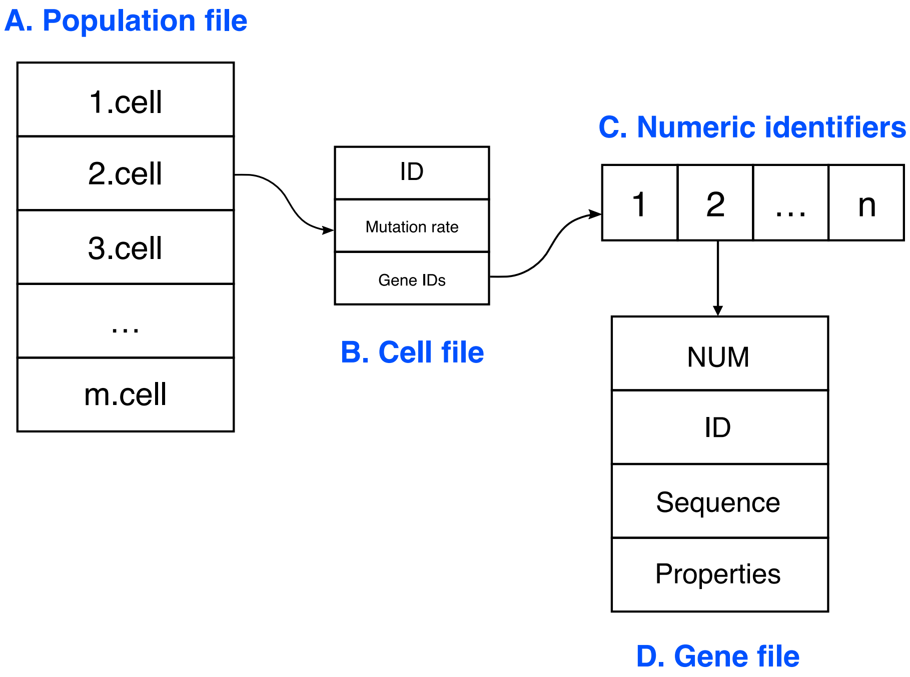


Figure 4.2: Illustration of SodaPop's input file hierarchy. **(A)** The population file indicates the clonal composition of the initial population as the count of different cell types. These cells are defined individually in **(B)** cell files that contain a numeric identifier, a mutation rate and a list of gene identifiers. These numeric identifiers are indexed in a gene list **(C)** which maps to gene files **(D)** that contain gene sequence and information.

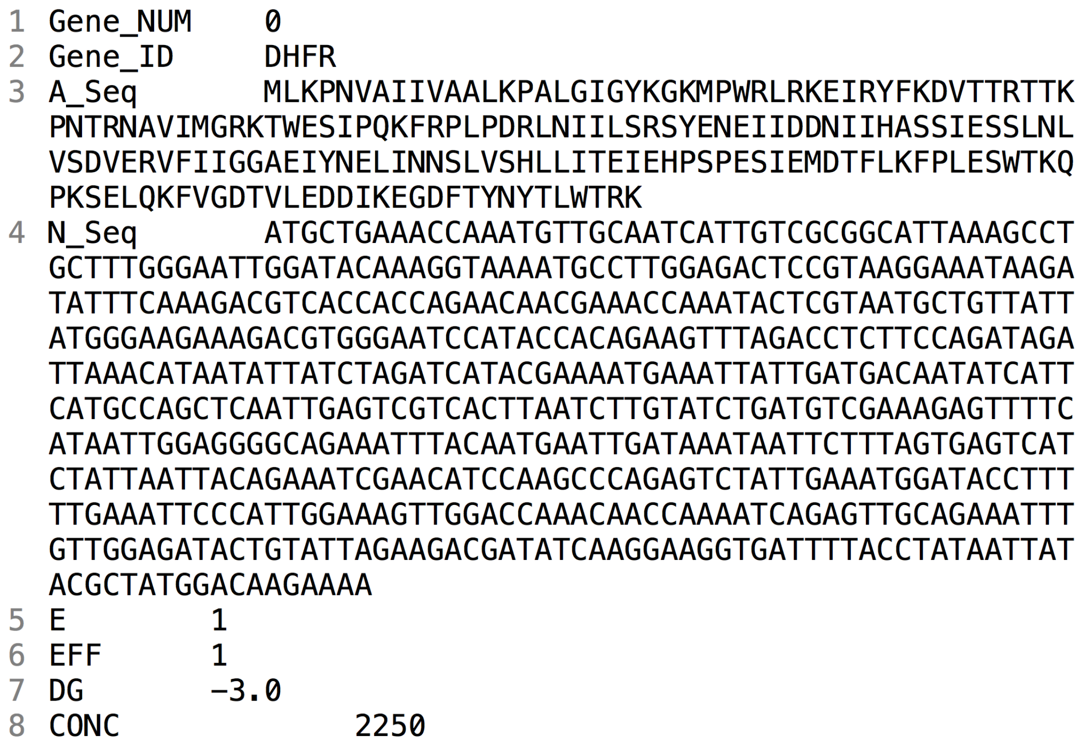
4.2 Creating a starting population snapshot

Prior to running a simulation, the first thing to do is to create your own population snapshot. In SodaPop, a population is defined by a collection of cells, that are in turned defined by their genes. We will thus start by creating a gene file. An example gene file is shown in **Fig. 4.3**.

It has a numeric identifier as well as an alphanumeric ID. The numeric identifier is important for the proper mapping of primordial sequences. Genes are thus saved as [numericID].*gene*. You can keep an index of the alphanumeric ID corresponding to each gene in the gene list (textbfFig. 4.4). Genes have an explicit nucleotide sequence and the corresponding translated amino acid sequence. The nucleotide sequence should have an exact 3 to 1 correspondence with the protein sequence. There should be no stop codons in the sequence. 'E' is the gene's essentiality. 'DG' is the Gibbs folding free energy (ΔG, in $kcal/mol$) or folding stability of the corresponding protein. 'CONC' is the intracellular concentration of the protein. 'F' is the protein's fitness. 'EFF' is the catalytic efficiency of the protein.

The file gene_list.dat keeps an index of all the genes to be used in a simulation. An example gene list is shown in **Fig. 4.4**. You can define as many genes and gene lists as necessary. However, make sure you use the correct list when you run a simulation. If there is a mismatch between the genes of your initial population and the genes in your index list, the program will exit and issue an error.

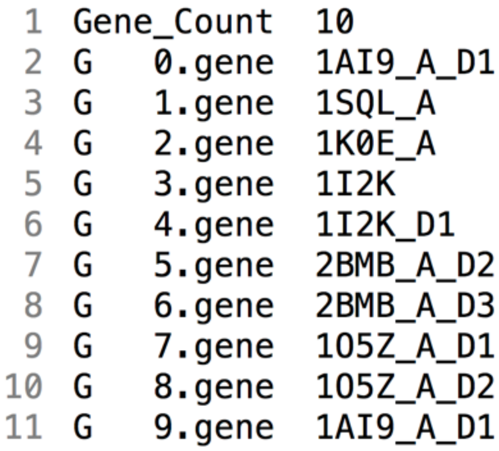
Now that we have created a gene, we need to define a cell and its properties. Cells possess



```
Gene_NUM    0
Gene_ID     DHFR
A_Seq       MLKPNVAIIVAALKPALGIGYKGKMPWRLRKEIRYFKDVTTRTTK
PNTRNAVIMGRKTWESIPQKFRPLPDRLNIILSRSYENEIIDDNIIHASSIESSLNL
VSDVERVFIIGGAEIYNELINNSLVSHLLITEIEHPSPESIEMDTFLKFPLESWTKQ
PKSELQKFVGDTVLEDDIKEGDFTYNYTLWTRK
N_Seq       ATGCTGAAACCAAATGTTGCAATCATTGTCGCGGCATTAAAGCCT
GCTTTGGGAATTGGATACAAAGGTAAAATGCCTTGGAGACTCCGTAAGGAAATAAGA
TATTTCAAAGACGTCACCACCAGAACAACGAAACCAAATACTCGTAATGCTGTTATT
ATGGGAAGAAAGACGTGGGAATCCATACCACAGAAGTTTAGACCTCTTCCAGATAGA
TTAAACATAATATTATCTAGATCATACGAAAATGAAATTATTGATGACAATATCATT
CATGCCAGCTCAATTGAGTCGTCACTTAATCTTGTATCTGATGTCGAAAGAGTTTTC
ATAATTGGAGGGGCAGAAATTTACAATGAATTGATAAATAATTCTTTAGTGAGTCAT
CTATTAATTACAGAAATCGAACATCCAAGCCCAGAGTCTATTGAAATGGATACCTTT
TTGAAATTCCCATTGGAAAGTTGGACCAAACAACCAAAATCAGAGTTGCAGAAATTT
GTTGGAGATACTGTATTAGAAGACGATATCAAGGAAGGTGATTTTACCTATAATTAT
ACGCTATGGACAAGAAAA
E       1
EFF     1
DG      -3.0
CONC            2250
```

Figure 4.3: Example of *.gene* file description.

individual attributes such as a mutation rate. The *.cell* file lists all the identifiers of genes in a cell, preceded by a 'G'. Again, genes are indexed by their numeric identifier. The same goes for cells. An example cell file is shown in **Fig. 4.5**.

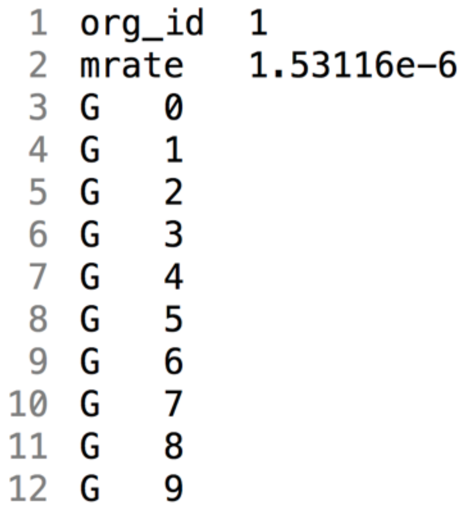


```
Gene_Count  10
G   0.gene  1AI9_A_D1
G   1.gene  1SQL_A
G   2.gene  1K0E_A
G   3.gene  1I2K
G   4.gene  1I2K_D1
G   5.gene  2BMB_A_D2
G   6.gene  2BMB_A_D3
G   7.gene  1O5Z_A_D1
G   8.gene  1O5Z_A_D2
G   9.gene  1AI9_A_D1
```

Figure 4.4: Example of *gene_list* file description.

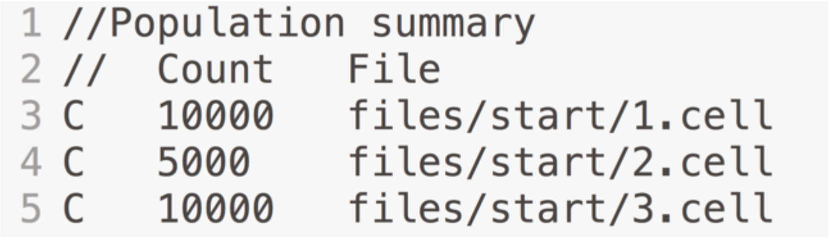
Once a cell has been defined, we can create a starting population snapshot. The last layer in the hierarchy is the population description file (**Fig. 4.6**). It lists the composition of the initial population we wish to create. In our case, we will define a polyclonal population with three different clones. The template shows how to define this population (**Fig. 4.6**). The count is the size of the starting subpopulation for each clone. The comment is optional and is ignored by the program.

To recapitulate, we must first create a gene file (**Fig. 4.2D**). We then define a cell file to include our gene(s) (**Fig. 4.2B**). Finally, we create a population summary defining the initial clonal structure



```
org_id  1
mrate   1.53116e-6
G   0
G   1
G   2
G   3
G   4
G   5
G   6
G   7
G   8
G   9
```

Figure 4.5: Example of *.cell* file description.



```
//Population summary
//  Count   File
C   10000   files/start/1.cell
C   5000    files/start/2.cell
C   10000   files/start/3.cell
```

Figure 4.6: Example of *population* file description.

we want (**Fig. 4.2A**).

We will use the program called **sodasumm** to create a starting population snapshot. If you run the program without any flag or argument, it will display the options:

```
 sodasumm <population summary> [ 0-full | 1-single cell | 2-randomize
barcodes | 3-introduce variation]
```

As input, we need the population summary created above (**Fig. 4.6**), and we must specify whether we want to build the full population with clones sharing the same barcode **(option 0)**, a randomized population from the first cell in the file **(option 1)**, the full population with each cell having a randomized unique barcode **(option 2)**, or the full population with clones sharing the same barcode, with fitness effects randomly introduced in clusters of cells **(option 3)**. After running **sodasumm**, a *.snap* file will be created in the same directory and with the same prefix as the population description file. This is your starting snapshot. You can rename it, but keep the *.snap* extension to distinguish between different input files.

4.3 Running a neutral simulation

Using the created population snapshot, we can run a very simple neutral simulation, that is, without selection. We can do this using the sample files packaged with the program. Try running the following command:

```
sodapop --sim-type stability -f 5 -p files/start/pop1K.snap
  -g files/genes/gene_list.dat -o neutral -n 1000 -m 1000 -t 10
```

The command-line display will read:

```
 Begin ...
Initializing matrix ...
Loading primordial genes file ...
Opening starting population snapshot ...
Creating directory out/neutral/snapshots ...  OK
Constructing population from source files/start/pop1K.snap ...
Saving initial population snapshot ...
Starting evolution ...
Done.
Total number of mutation events:  10767
```

Breaking down this command step by step:

- `sodapop` is the main simulation utility (or program).
- `--sim-type stability` defines the type of input to be used for the simulation. In our case, we chose stability.
- `-f 5` is the choice of fitness function. For more information on fitness functions, see **Chapter 7**. For now, just know that 5 is the mapping for the neutral fitness function.
- `-p files/start/pop1K.snap` is the the starting population snapshot file.
- `-g files/genes/gene_list.dat` is the gene list file.
- `-o neutral` is the prefix for output.
- `-n 1000` is the population size.
- `-m 1000` is the number of generations we want to simulate.
- `-t 10` is the interval at which the program saves a population snapshot.

4.4 Folding stability simulation example

In the `files/` folder of the SodaPop archive, you will find examples and data to help you get started. Here, we will use some of these files to run a simple simulation based on protein folding stability of the dihydrofolate reductase enzyme (DHFR) which is encoded by the *folA* gene in *E. coli*.

The file *DHFR_DDG.matrix* is the fitness landscape of the protein folding stability of *E. coli* DHFR. We will use it as input, along with a starting population snapshot of $N_e = 10^4$ initially monoclonal cells. We will use the first fitness function, based on metabolic flux (see **Chapter 7**). The following command will start a simulation and direct output to the directory `out/foldsim/`:

```
sodapop --sim-type stability -i files/start/DHFR_DDG.matrix
  -f 1 -p files/start/DHFR_start.snap -g files/genes/gene_list.dat
  -o foldsim -t 100 -n 10000 -m 5000 -a
```

Refer to **Chapter 6** for detailed information on simulation output.

4.5 Sampled selection coefficients example

Here, we will use the multiplicative fitness function (**-f 4**) and sample selection coefficients from a gaussian distribution with $\mu = -0.02$ and $\sigma = 0.01$:

```
sodapop --sim-type s --normal --alpha -0.02 --beta 0.01
  -p files/start/population.snap -g files/genes/gene_list.dat
  -o codesim -t 100 -n 10000 -m 1500 -a
```

4.6 Converting binary snapshots to text

sodasumm and **sodapop** both create snapshots in binary format (.snap). To manually convert binary files to text, use **sodasnap**:

```
sodasnap <snap-binary> <out-ascii>
```

Where `<snap-binary>` is the binary snapshot to convert and `<out-ascii>` is the name of the text file to create. Alternatively, adding the `-a` flag to your **sodapop** command will call analysis scripts once the simulation is done. This includes automatically converting your snapshots to text, extracting barcodes and generating basic plots. Refer to **Chapter 6** for more details.

5. Fitness landscapes

When a random mutation occurs in SodaPop, its effect on organismal fitness depends on the fitness landscape chosen by the user. The following sections describe the three models of fitness landscape that can be used in the program.

5.1 Phenomenological distribution

The effects of random mutations on organismal fitness can come from user-defined distributions. In the current version of SodaPop, the DFE can be modelled by a two-parameter distribution of either normal form $N(\mu,\sigma)$ or gamma form $\Gamma(\alpha,\beta)$ using the `--normal` and `--gamma` flags. The `--alpha` and `--beta` flags are used to specify μ and σ or α and β, respectively. These flags are used in tandem with `--sim-type` to determine the nature of the drawn values. For instance, the command

```
sodapop --sim-type s --normal --alpha -0.02 --beta 0.01
  -p files/start/population.snap -g files/genes/gene_list.dat
  -o codesim -t 100 -n 10000 -m 1500 -a
```

will draw selection coefficients from a distribution $N(-0.02,0.01)$.

However, the command

```
sodapop --sim-type stability --normal --alpha 1 --beta 1 -f 1
  -p files/start/population.snap -g files/genes/gene_list.dat
  -o codesim -t 100 -n 10000 -m 1500 -a
```

will draw $\Delta\Delta G$ values from a distribution $N(1,1)$ and calculate fitness effects based on the resulting folding stability, with all other things kept equal. The same distinction is applied to `--gamma`.

5.2 Computational tools in biophysics

SodaPop can take as input one or more substitution matrices that describe the molecular effects of mutations on a particular protein property for all 1-away mutants (**Fig. 5.1**). The entries in these tables can be derived from computational protein engineering tools such as Rosetta [1], Eris [4] or FoldX [2]. Each matrix must be tab-separated and preceded by a header line containing the prefix "Gene_NUM", the gene index matching the corresponding gene file and the protein length in amino acids. Then, each row in the matrix must start by indicating the property (e.g. "DDG"), the residue position (starting at 1), followed by a list of values for each amino acid substitution, including the identity substitution. If you are simulating several proteins at once, each with a different matrix, you must list the matrices one after the other in a single file. To input this file in a simulation, you must use the `-i` flag **Section 3.3**.

```
Gene_NUM 0 192
DDG 1   -0.01400     8.33133    -0.67133
-0.40333     0.34067     1.98400     0.74733
-0.63400    -1.84533    -1.62733     0.00000
0.00733    10.50000    -1.56800     0.72133
-0.48933    -1.76800    -0.54933     2.06733
0.82933
DDG 2   -0.32400     8.24733    -0.92600
-0.17933     1.74867     0.75400     2.02400
0.58200    -0.03667     0.00000     1.28333
0.63067     0.00867     0.47267     0.99067
0.14667    -0.32200    -0.55067     3.50600
1.32000
DDG 3    0.72600     8.69400     0.12467
-0.35133     2.53400     0.92333     0.96400
-0.13800     0.00000     0.62400     1.05600
-0.07867    -0.32000     0.23667     0.70200
0.82933     0.37867    -0.29267     4.00933
1.53867
```

Figure 5.1: Example of substitution matrix for folding stability.

5.3 Experimentally derived fitness effects

SodaPop can take as input a substitution matrix that describes the fitness effects of all 1-away mutants derived from deep mutagenesis scanning experiments (**Fig. 5.2**). Here, we describe the format of these matrices. Each matrix must be tab-separated and preceded by a header line containing the prefix "Gene_NUM", the gene index matching the corresponding gene file and the protein length in amino acids. Then, each row in the matrix must start by indicating "DMS", the residue position (starting at 1), followed by a list of values for each amino acid substitution, including the identity substitution. If you are simulating several proteins at once, each with a different matrix, you must list the matrices one after the other in a single file. To input this file in a simulation, you must use the `-i` flag **Section 3.3**.

```
Gene_NUM    0   211
DMS  1  -0.061012  -0.051554  -0.050201
-0.036683  -0.021648  -0.062298  -0.034156
-0.024325  -0.037651  -0.024325  0  -0.046309
-0.038143  -0.038268  -0.033732  -0.074003
-0.033432  -0.03362  -0.048549  -0.02905
DMS  2  0  -0.041417  -0.045628  -0.045631
-0.049666  -0.032799  -0.046542  -0.049647
-0.051251  -0.051241  -0.044474  -0.045628
-0.022912  -0.045205  -0.051251  -0.035963
-0.031204  -0.033573  -0.057602  -0.049666
DMS  3  -0.034349  -0.054748  -0.046424
-0.049064  -0.057602  0  -0.050634  -0.062654
-0.052264  -0.060104  -0.051251  -0.044941
-0.044677  -0.046542  -0.052264  -0.033109
-0.048768  -0.052454  -0.058593  -0.058576
```

Figure 5.2: Example of substitution matrix derived from DMS data.

6. Results and analysis

The package includes a small bioinformatics pipeline to extract, analyze and plot your data. The scripts are written in Bash and R* and can be used separately from the main program. Those familiar with scripting languages can also use the scripts as templates for other analyses.

By default, SodaPop will always output population snapshots following the format chosen by the user. To toggle on automatic analysis, use the `-a [-analyze]` boolean flag with your command (see List of command-line options).

> **Note** As a requirement for plotting, the R programming language must be installed on your machine. You can download the latest version of R here. The script will automatically install the required R packages for you at runtime.

6.1 Scripts

barcodes.sh : this is a Bash script that uses Unix utilities such as `awk`, `grep`, `uniq`, `sort` and `join`. It executes the following operations:

1. Detect snapshot file format
2. Unzip and convert binary snapshots to text
3. Extract and sort the barcodes for each time point
4. Compute the average population fitness for each time point
5. Count the number of occurrences of a given barcode at each time point
6. Identify the time point corresponding to the fixation of a single barcode
7. Combine all time points in a dataframe
8. Call *polyclonal_structure.R*

This script can executed independently from the main program using the command-line. It requires the following parameters: [name of the simulation directory] [number of generations] [population size] [step size (dt)] [encoding format] [gene count per cell].

As an example, the following would be the command to analyze a long format simulation with n = 10000, m = 10000 and a time step of t = 25 generations:

```
./barcodes.sh test_sim 10000 10000 25 0
```

polyclonal_structure.R : this is a R script that imports the time series data and the fitness table to generate four distinct plots. Examples of these plots are found throughout this chapter.

6.2 Mean fitness plot

The first plot is named *fitness* and shows the evolution of the average population fitness during the course of the simulation.

In the example below (**Fig. 6.1**), we observe a steep drop in fitness in the first few generations. This is a typical case of Muller's ratchet: under a high mutation rate, the population sees its mean

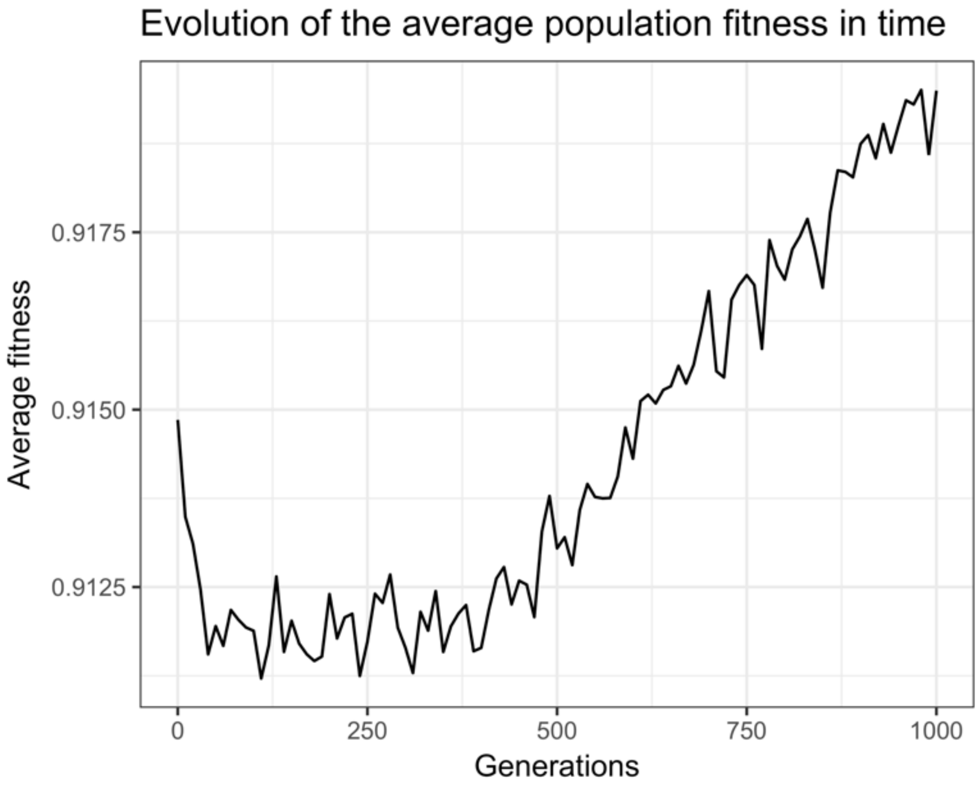


Figure 6.1: Example trajectory showing the evolution of the average population fitness over 1000 generations for $N_e = 10^4$.

fitness decrease as the fittest individuals are hit by marginally deleterious mutations. Over time, marginally beneficial mutations arise, conferring a selective advantage that is driving the mean fitness higher.

6.3 Clonal structure

The second plot is named *clonal_structure*. It represents the share of each unique barcode over the course of the simulation, up to the point of fixation of a single lineage. In **Fig. 6.2**, the vertical axis represents the density of each lineage. The sum of all lineages is always equal to the size of the population (in this case, 10^4). The slight slope in the top left area of the plot is a result of discarding all lineages that are lost immediately after the first time point. The reason is that as the ratio of unique barcodes to the population size approaches 1, a majority of barcodes will be naturally lost to stochastic drift. Removing these lineages speeds up plotting and makes for a sharper image.

6.4 Clonal trajectories

The two remaining plots are named *clonal_trajectories* and *log_clonal_trajectories*. They are temporal representations of the structure of the population in terms of each segregating lineage – that is, cells sharing a common barcode.

In **Fig. 6.3**, the vertical axis represents the total count of each lineage as it segregates. Again, this plot will show lineages up to the point of fixation, if such a point exists. This plot is useful to quickly identify selective sweeps. **Fig. 6.4** shows the same plot with the vertical axis in log10-scale. This is useful to visualize the segregation of low-frequency lineages and estimate the selection

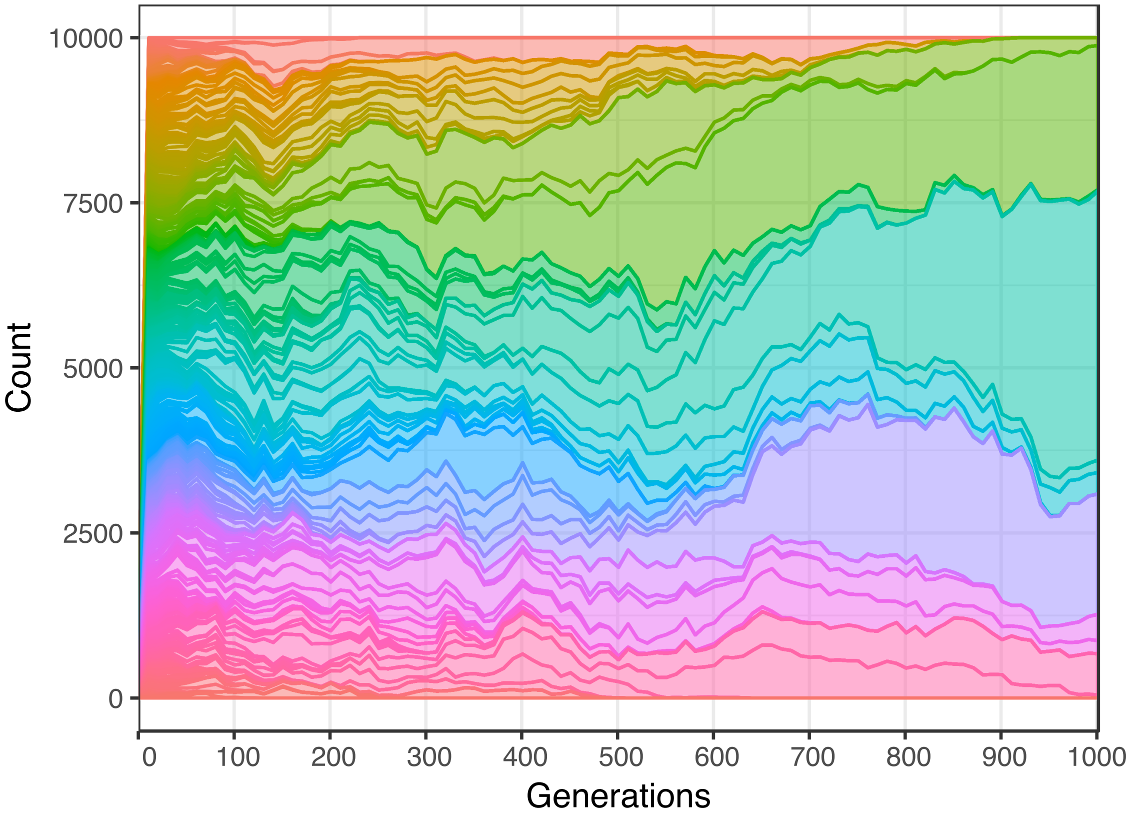


Figure 6.2: Example run of $N_e = 10^4$ cells showing the clonal structure spanning 1000 generations.

coefficient and establishment time of beneficial mutation by extrapolating the exponential growth regime of a lineage. These estimations can then be correlated with the exact numbers found in the mutation log (see section below).

6.5 Mutation log

The file *MUTATION_LOG* contains the information on arising mutations in a simulation run. It can be toggled on with the `-e [-track-events]` flag.

The tab-separated file lists mutations line-by-line:

```
AGACTCAAGTGTGAC 6 K 272 E 0.001078 1
TGCAAGCAAACGGGC 5 D 160 H -0.363828 9
CGAGACTGTGGGAGT 4 T 48 R -0.198414 21
...
```

The columns respectively correspond to the barcode, the gene ID, the previous residue identity, its position in the sequence, the new residue identity (the mutation), its selection coefficient and the generation at which it occurred.

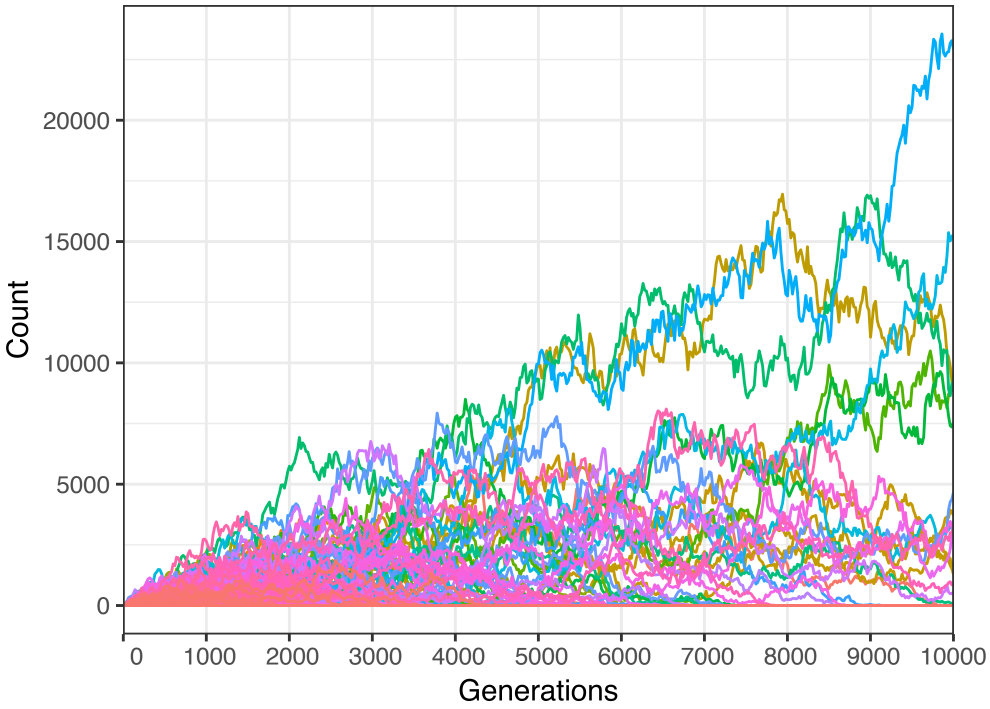


Figure 6.3: Example run of $N_e = 10^5$ cells showing the clonal trajectories spanning 10000 generations.

6.6 Command log

The *command.log* file is created by default for each simulation run and contains the full command and its corresponding standard output.

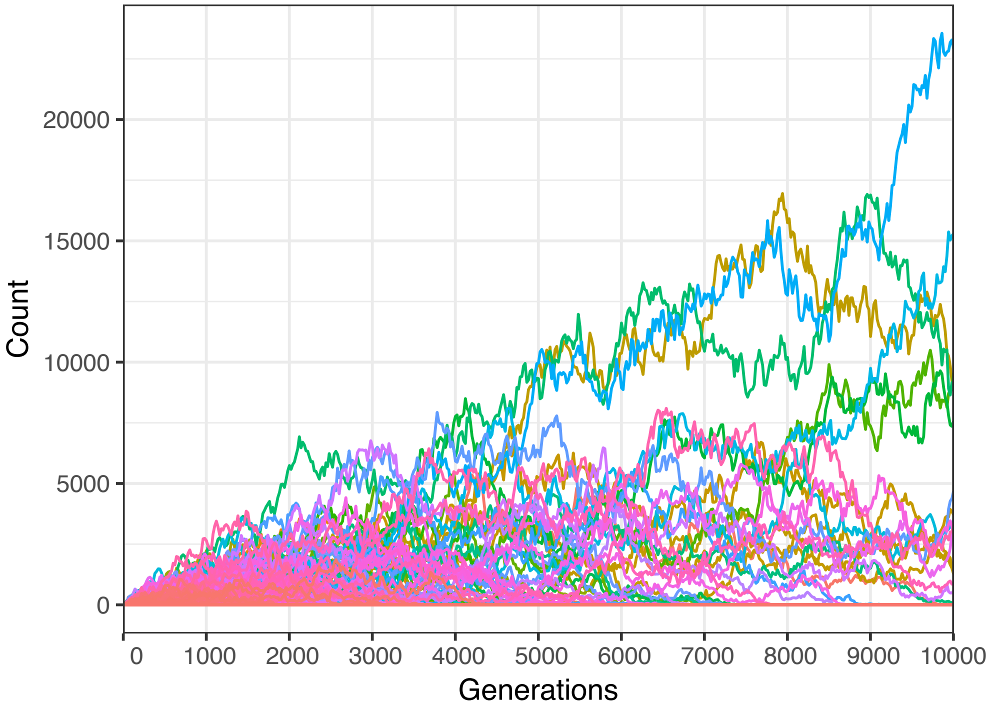


Figure 6.4: Example run of $N_e = 10^5$ cells showing the clonal trajectories spanning 10000 generations.

7. Fitness functions

SodaPop allows you to specify how organismal fitness is calculated by choosing a fitness function.

There are currently five native functions in this version. However, it is easy to implement a custom fitness function (for details, see **Section 8.1**). Here we introduce the native fitness functions of the program.

7.1 Metabolic flux

This function expresses organismal fitness as the metabolic output of the cell, proportional to the number of correctly folded proteins. Assuming a two-state model of protein folding [3], the fraction of proteins in the native state P_{nat} is given by

$$P_{(nat,i)} = \frac{1}{1+e^{\beta \Delta G_i}} \tag{7.1}$$

where $\beta = \frac{1}{k_B T}$ $(k_B T = 0.593 kcal/mol)$ and ΔG_i is the folding free energy of the protein. Summing over all genes and accounting for gene expression level, we obtain

$$fitness = \frac{a_0}{\sum_i (A_i \cdot P_{(nat,i)})^{-1}} \tag{7.2}$$

where a_0 is a normalizing factor, i is the index for each gene and A_i is the corresponding cellular abundance of the protein. By requiring that fitness be optimally 1 at very stable regimes ($\Delta G \rightarrow \infty$), we get

$$a_0 = \frac{1}{\sum_{i=1}^{N} (A_i)^{-1}} \tag{7.3}$$

7.2 Toxicity

This function describes fitness as the fitness cost due to misfolded proteins in the cell, given by

$$fitness = c \sum_i A_i (1 - P_{(nat,i)}) \tag{7.4}$$

where c is a constant modulating the fitness cost per misfolded protein.

7.3 Combined metabolic output

This function describes fitness as a combination of the two previous functions, namely metabolic flux and toxicity due to misfolding. It is expressed as the difference

$$fitness = f_{flux} - f_{toxicity} \tag{7.5}$$

If $f_{toxicity} > f_{flux}$, the cell is not viable and the fitness is set to 0.

7.4 Neutral

In this scenario, the fitness effect of all arising mutations is null. That is, the selection coefficient of a mutation is 0. Thus, organismal fitness is given by

$$f_{neutral} = 1 \tag{7.6}$$

7.5 Multiplicative

Consider a gene with wild-type fitness w_f that acquires n mutations with selection coefficient s_i. The multiplicative fitness of this gene is given by

$$F_n = w_f \prod_{i}^{n} (1 + s_i) \tag{7.7}$$

By design, only a single mutation can arise in a gene at once, so this computation is straightforward and only requires the previous fitness state of the sequence. In this case, if multiple genes are present, the organismal fitness is the arithmetic mean of all sequences' fitnesses.

7.6 No mutation

Mutations can be turned off completely by using this dummy fitness function. This is useful to simulate the effects of standing genetic variation on population dynamics.

8. Modifying the code

SodaPop is written in an object-oriented programming (OOP) framework, making it easily hackable by anyone with elementary knowledge of OOP and/or C++.

8.1 Implementing a fitness function

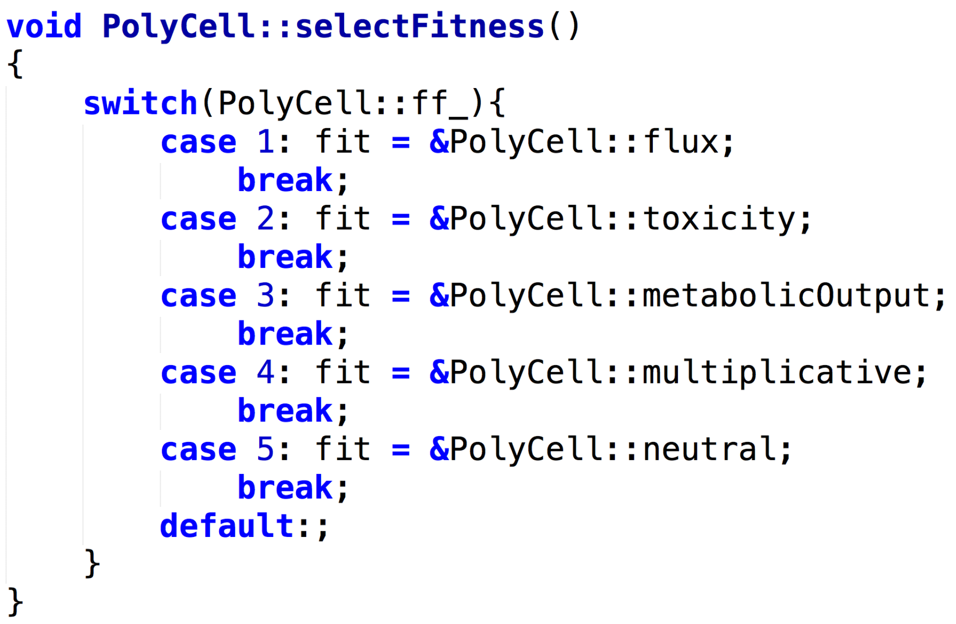
```
// METABOLIC FLUX FITNESS FUNCTION
double PolyCell::metabolicOutput()
{
    double flux = 0;
    double toxicity = 0;
    for (auto& it : Gene_arr_) {
        flux += 1 / it.functional();
        toxicity += it.misfolded();
    }

    flux = A_FACTOR / flux;
    toxicity = COST * toxicity;

    double fitness = flux - toxicity;
    return (fitness < 0) ? 0 : fitness;
}
```

Figure 8.1: Example of fitness function. The function returns a double.

Fitness functions are located in the *PolyCell.h* source file. All functions have the same return type (double) and are indexed by increasing integers starting from 1. A static function pointer selects the correct fitness function to use during simulations (**Fig. 7.2**). Users can implement a new fitness function by following this signature and adding their new function to the selectFitness() switch case statement.

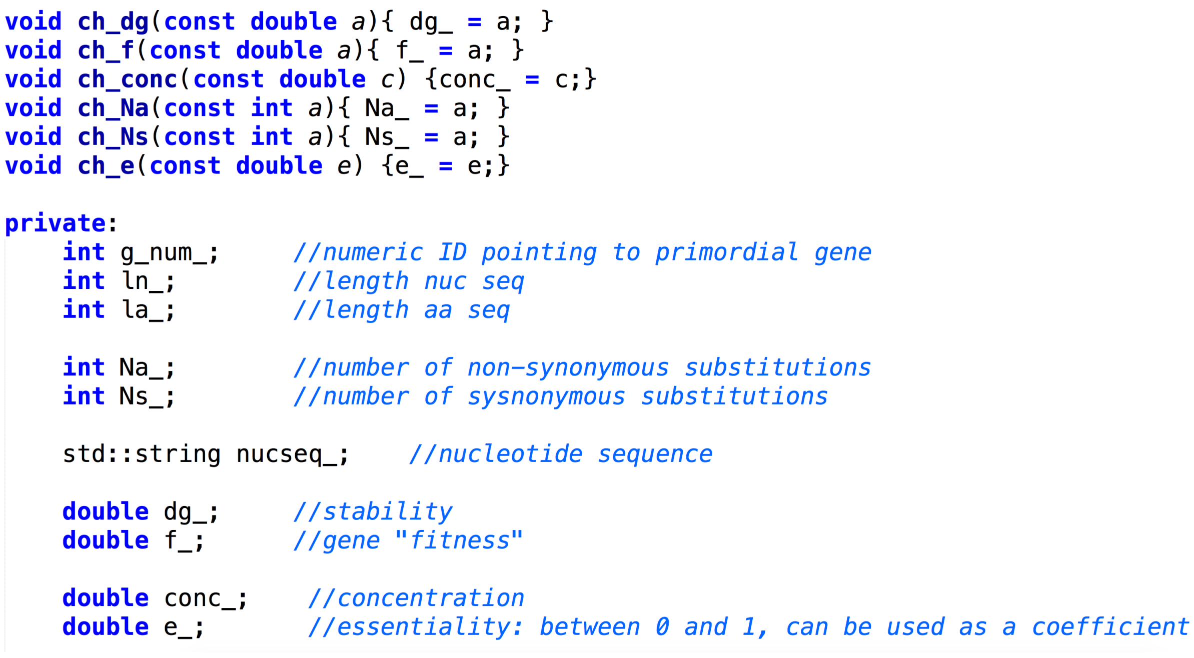
8.2 Adding gene properties

The present version implements folding stability (ΔG) and enzymatic efficiency (K_{cat}/K_m) as the fitness defining properties for each gene product. Other biochemical properties can be implemented here, such as or binding affinity (K_d). Additionally, other properties like protein concentration and essentiality can be used to model specific genotype-to-phenotype relationships (fitness functions).



```cpp
void PolyCell::selectFitness()
{
    switch(PolyCell::ff_){
        case 1: fit = &PolyCell::flux;
            break;
        case 2: fit = &PolyCell::toxicity;
            break;
        case 3: fit = &PolyCell::metabolicOutput;
            break;
        case 4: fit = &PolyCell::multiplicative;
            break;
        case 5: fit = &PolyCell::neutral;
            break;
        default:;
    }
}
```

Figure 8.2: The control structure used to select the proper fitness function. New functions can be added to the switch statement with a function pointer.



```cpp
void ch_dg(const double a){ dg_ = a; }
void ch_f(const double a){ f_ = a; }
void ch_conc(const double c) {conc_ = c;}
void ch_Na(const int a){ Na_ = a; }
void ch_Ns(const int a){ Ns_ = a; }
void ch_e(const double e) {e_ = e;}

private:
    int g_num_;     //numeric ID pointing to primordial gene
    int ln_;        //length nuc seq
    int la_;        //length aa seq

    int Na_;        //number of non-synonymous substitutions
    int Ns_;        //number of sysnonymous substitutions

    std::string nucseq_;    //nucleotide sequence

    double dg_;     //stability
    double f_;      //gene "fitness"

    double conc_;    //concentration
    double e_;       //essentiality: between 0 and 1, can be used as a coefficient
```

Figure 8.3: Part of the *gene.h* class definition. New private properties can be added here, with the corresponding accessory functions.

9. Implementation and performance

SodaPop is built on streamlined data structures and a fast algorithm to achieve high computational efficiency and to minimize the general trade-off between flexibility and runtime. Here we briefly describe this software's design decisions.

9.1 Data structures and complexity

The population is implemented as a vector (*std::vector* container) of cells. This is because vectors store elements contiguously in physical memory, making elements accessible sequentially through iterators and pointer arithmetic, and providing spatial locality of reference. The most common operation is appending a cell at the end of the vector, which is done in amortized constant time [$O(1)$]. This operation is linear $O(k)$ when appending k cells at once. Random iterator access is also done in constant time, making the random rescaling of the population an efficient operation that is linear in the number of cells to be added/removed. Vectors also confer efficient memory handling since memory can be reserved ahead of time, provided a good estimate of capacity is known. Because we use a constant population size between generations, and because generation overhead never exceeds $2N_e$, the upper bound on population size is a known variable, preventing costly $O(n)$ vector doubling reallocations when the container reaches full capacity.

Whenever it is possible, we use safe C++ standard library operators such as *std::shuffle* and *std::swap*. The first operator is used when the new generation exceeds the intended size (as a result of binomial drawings). It performs a fast random shuffle of the population vector indexes. This is followed by a standard resizing of the vector whose complexity is linear on the number of elements erased. Essentially, instead of picking k cells to erase at random and downshifting the vector every time, we shuffle the vector and then trim off the last k cells. The second operator is used to assign the new vector of cells to the population vector, overwriting the previous generation. Because cells are large objects containing a lot of information distributed over different containers, overwriting the population one by one would not only be inefficient, but potentially pose the risk of losing information by shallow copies of certain object members. The *swap* operator uses move semantics to swap the two cell vectors efficiently. We can then reset the old vector for the new generation.

The program reads and writes binary files because they are smaller in size and can be handled with buffers reading discrete chunks of data. Since the memory overhead for I/O is significantly higher than that of type conversion, the computational cost incurred by converting types to binary is negligible. Moreover, while using binary over raw text might not be significantly advantageous for small-scale simulations, it can trim down the size of snapshots in larger runs where each population can reach several hundred megabytes in size.

9.2 Pseudo-random number generation

We are working with the assumption that arising mutations are uniformly distributed along the genome, and a single simulation run can easily produce millions of mutations. Hence, we need a reliable pseudo-random number generator (PRNG) that lacks bias. Additionally, we want our

PRNG to be relatively fast as it is being used heavily in the algorithm. Finally, the PRNG should be lightweight and compatible with C/C++. Following these criteria, we opted to use the PCG family of random generators by Melissa E. O'Neill, as a substitute to the widely used Mersenne Twister with a native implementation in C++11. The latter has a large state space and is known to be uneven in its output, while PCG is compact, relatively fast, and uniform in its output, which makes it a good candidate for simulation.

9.3 Runtime

SodaPop is the first publicly available tool that can effectively simulate multi-scale molecular evolution and polyclonal population dynamics. We benchmarked SodaPop for multiple population sizes and number of generations. All simulations were run on a standard iMac desktop with a 3.2GHz Intel Core i5 processor and 16GB memory. **Figure 8.1** shows that runtime is quasi-monomial with respect to population size. Simulating up to a million cells for long time periods is entirely tractable using standard desktop computers. We limited our desktop benchmarking to $N_e = 10^6$ cells, as higher orders of magnitude induce a shift in performance due to a RAM bottleneck. The simulation of populations with higher orders of magnitude requires a larger amount of memory than the current standard in commercial desktop computers.However, larger population sizes can be simulated on high-performance computing clusters where memory allocation is not as limiting.

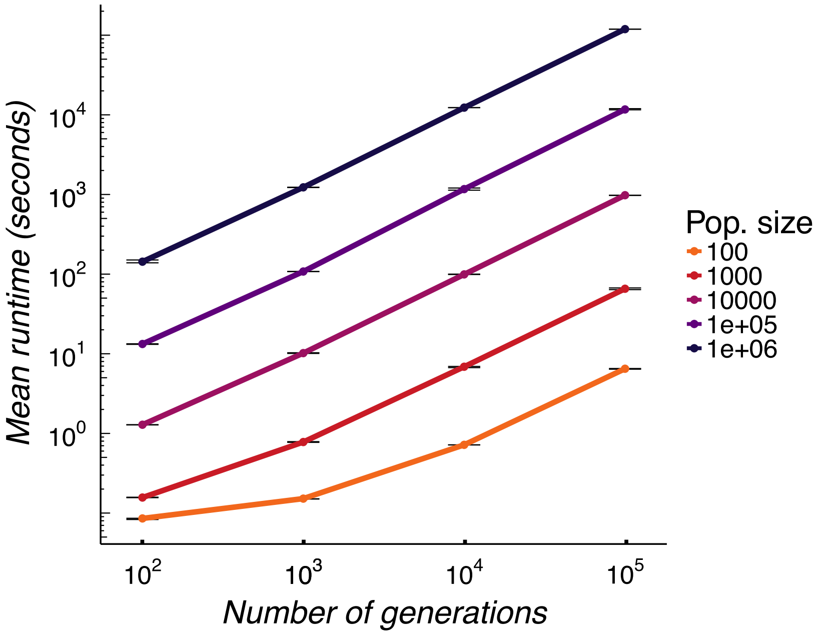
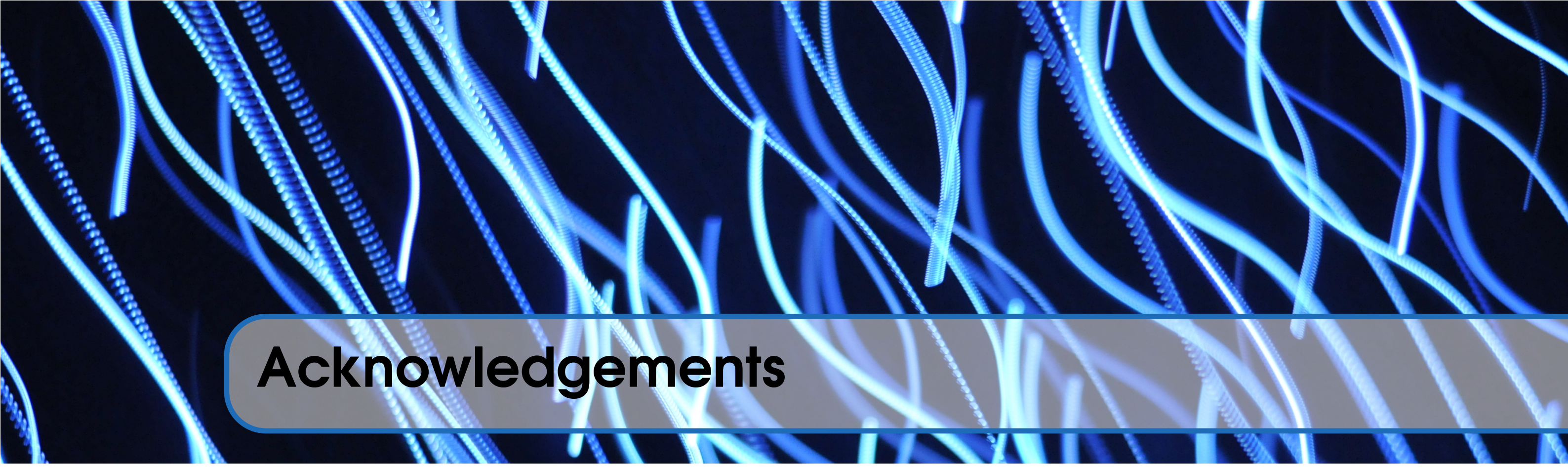
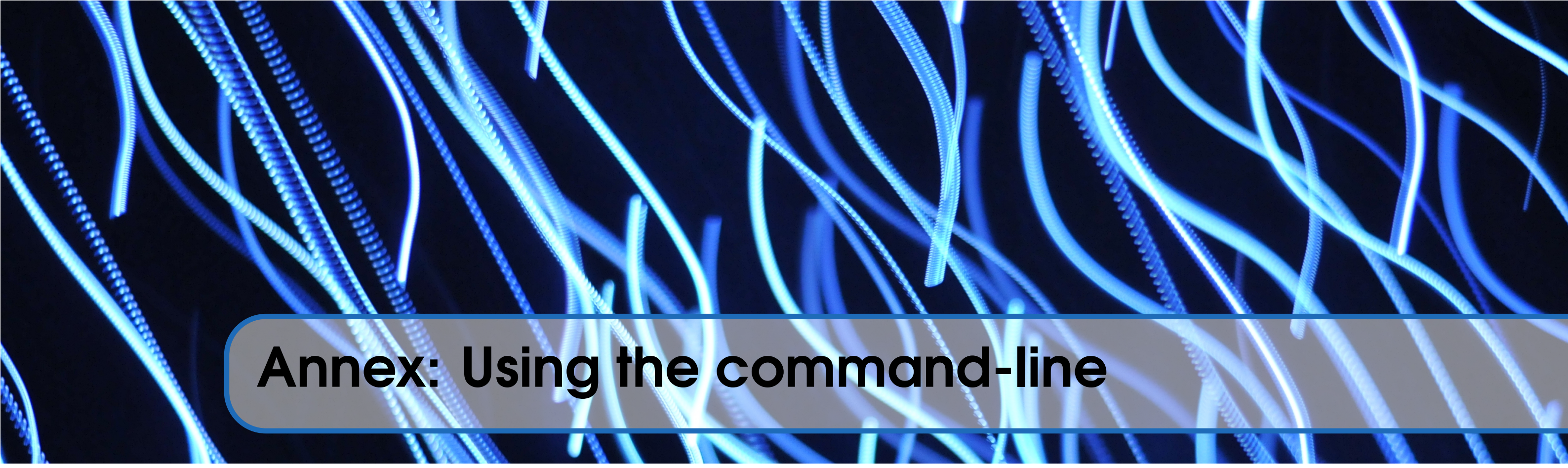


Figure 9.1: Average runtimes per population size for different time scales. The initial population was characterized by a monoclonal cell with a single 194 bp gene. The time step for snapshots was set at 0.01∗(number of generations) for each scenario. Each point represents the average runtime over 100 simulations for a particular condition. Error bars represent standard error of the mean.



Acknowledgements

We would like to thank members of the Serohijos lab for testing the program and contributing valuable questions and ideas. Part of this research was funded by NSERC & start-up funds from UdeM.



Annex: Using the command-line

The command-line interface is a powerful way to interact with the computer. In its simplest form, the command-line is a space where you type commands for the computer to execute. On Mac OS/X, the command-line is an application called Terminal. It is located in the `/Applications/Utilities/` folder.

The overwhelming majority of command-line programs follow the same simple syntax. A command can be broken down into three basic components:

- The *utility*, also referred to as the program. Utilities execute a sequence of actions depending on their input. In some cases, you may use a utility without any flag or argument.
- The *flags*: flags are like options. They allow you to modify the behavior of the utility.
- The *arguments*: some utilities take arguments. These are commonly files, but they can be numbers, words or special characters.

Here is a simple command. We will break it down by component.

```
ls -l Documents/
```

`ls` is the utility. It lists the content of directories in the command-line interface.

`-l` is a flag. It indicates that we want more information than is provided by default. Flags are most often preceded by a hyphen ('-') and consist of single characters. Some flags can also be specified using a word preceded by a double hyphen ('--'). Using one or the other will have the interpretation by the command-line. For instance, typing

```
man -h
```

or

```
man -help
```

will output the same response from the terminal. The `man` utility displays the manual pages for a specific utility. The command is used in conjuction with another utility as its *argument*. For example,

```
man ls
```

will display the manual page for `ls`.

The last component of our initial command is the argument 'Documents/'. It tells `ls` that we want to show the content of that directory. And that?s it! Besides `ls`, you will need to know how to use a few other basic utilities in order to get started.

```
cd
```

is the command to **c**hange **d**irectory. By default, without any flag or argument, `cd` will move up one folder. You can navigate down a folder by giving the name of the folder you wish to move to as an argument. If you are unsure of the folder you are currently in, type

```
pwd
```

The command stands for **p**rint **w**orking **d**irectory, and it does exactly that.

If you want to view the contents of a text file, say 'some_file.txt', try

```
less some_file.txt
```

This will display the content of the file in the terminal. You can scroll through the file using the up and down arrows. Pressing Q will quit the utility and bring you back to the command-line prompt.

If you want to copy a file to a specific location, say 'Data/', type

```
cp some_file.txt Data/
```

Likewise, you can instead *move* the file using a similar syntax:

```
mv some_file.txt Data/
```

Finally, you can remove a file using

```
rm some_file.txt
```

> **Note** A handy feature of the terminal is TAB autocompletion. Whenever you are typing an argument, say a path or a filename, you can type the first few letters and press TAB. This will list all the files and folders accessible from your current directory corresponding to that prefix. If there is only one match, it will autocomplete the argument for you. Getting familiar with this option will help you to use the command-line swiftly. ■

Bibliography

[1] Rhiju Das and David Baker. Macromolecular modeling with rosetta. *Annual Review of Biochemistry*, 77(1):363–382, 2008. PMID: 18410248.

[2] Raphael Guerois, Jens Erik Nielsen, and Luis Serrano. Predicting changes in the stability of proteins and protein complexes: A study of more than 1000 mutations. *Journal of Molecular Biology*, 320(2):369 – 387, 2002.

[3] A. W. Serohijos and E. I. Shakhnovich. Contribution of selection for protein folding stability in shaping the patterns of polymorphisms in coding regions. *Mol Biol Evol*, 31(1):165–76, 2014.

[4] S. Yin, F. Ding, and N. V. Dokholyan. Eris: an automated estimator of protein stability. *Nat Methods*, 4(6):466–7, 2007.