eSTGt V1.0

Stochastic Simulations of Lineage Trees

User Guide

Table of Contents

Introduction

eSTGt is a MATLAB tool that enables to execute stochastic simulations that generate lineage trees. The input programs of the tool are based on a language formalism called environmental dependent Stochastic Tree Grammars (eSTG) that is described in a published paper[1]. Briefly, the formalism extends the notion of Stochastic Tree Grammars (STG)[2] by incorporating both rates and probabilities to the transition rules. These can be dynamically updated by defining them as functions of the system's state, which includes global values such as current population size or elapsed time. In addition, we extend the system by allowing each individual to hold its own internal states, which can be inherited. The species fate can also be controlled through conditional transitions on the system's state.
By executing a program the tool generates the corresponding lineage tree, which can be further analyzed either through the tool's extensions or by exporting the results into MATLAB's environment (by generating a .mat file) or by exporting the tree itself into a Newick format. If needed, the tool can perform multiple executions of a program by using several random seeds, which enables to stochastically sample multiple instances from the space of possible outcomes.
The simulations are carried out using the well-known Gillespie stochastic simulation algorithm[3]. Gillespie's implementation uses the rates of all possible reactions and chooses stochastically the next reaction by assuming that the time to the next reaction is exponentially distributed with rate parameters corresponding to the reaction rates.
The tool enables to execute the programs either through a GUI or through a command-line that enables easier batch processing (e.g. on a cluster).

Installation

The tool runs on MATLAB R2012b and later. In order to run the tool you should download latest version from Github: https://github.com/shapirolab/eSTGt and copy the files into a separate folder. There are two folders:

1) eSTGt - includes the source code
2) Programs - includes program examples

For ease of use add the eSTGt folder into your MATLAB's path.

Using eSTGt

Program Definition

The program definition is based on the eSTG formalism[1]. An example of a basic transition rule is:

$$A \xrightarrow{r} \{\mathrm{A},\mathrm{A}\}_{p_1} \big| \{\mathrm{A},\mathrm{B}\}_{p_2} \big| \{\mathrm{B}\}_{p_3} | \{\phi\}_{\mathrm{ow}}$$

where r is the rate of the transition and $p_1 + p_2 + p_3 + ow = 1$ are the transition probabilities ("ow" stands for "otherwise"). A program can have multiple transition rules and the rates and

probabilities can be updated according to the system's state (see examples below). Each species (e.g., A or B) contains at most one transition rule in which it is the source of the rule (i.e., on the left side of the transition) but can participate as a target (right hand side) as many times as needed. Each species can also include an internal state which can be updated and inherited through each transition execution.
In eSTGt we encode the program definition using an XML file along with accompanied MATLAB functions. The following examples demonstrate the structure of the program definition:

Example 1: A single species with feedback regulation

The transition rule:
$A \overset{r}{\rightarrow} \{A,A\}_p|\{\phi\}_{1-p}$

with the updating of the probability p as a function of the population size $|A|$ resulting in a logistic feedback regulation:

$$p = 1 - \frac{|A|}{200}$$

is represented in the following XML:

```
<Program>
<ExecParams>
      <SimTime>
            12
      </SimTime>
      <Seed>
            0
      </Seed>
</ExecParams>

<FunHandleName> updating_LogisticVerhulst </FunHandleName>
<Rule>
      <Prod>
            A -> 1 {A,A}_1 |{0}_0
      </Prod>
      <InitPop>
            1
      </InitPop>
</Rule>
</Program>
```

where `SimTime` is the simulation time, `Seed` is the random seed, `Prod` is the transition (or production) rule, and `InitPop` is the initial population size of the species A. The value within the tag `FunHandleName` corresponds to a MATLAB function with the same name that contains the updating rules during an execution of that transition. In this case, we define a MATLAB file with the name `updating_LogisticVerhulst.m` that contain the following code:

updating_LogisticVerhulst.m:

```
function Rules = updating_LogisticVerhulst(Rules,T,X)
p = Rules.Prod{1}.Probs(1);
K = 100;

pNew = 1-X(1)/(2*K); % Verhulst
pNew = min(1,max(0,pNew));

Rules.Prod{1}.Probs(1)  = pNew;
Rules.Prod{1}.Probs(2)  = 1-pNew;

end
```

This function updates the probability p according to the definition.

This updating function is called upon each execution of a transition rule and can update the rates and the probabilities. Let us take a closer look at this updating function and define its input and output:

```
function Rules = updating_LogisticVerhulst(Rules,T,X)
```

`Rules` is a struct that encodes that entire set of the program's transition rules (in this case we have only one transition rule) along with the current values of the corresponding rates and probabilities:

`Rules.Prod{i}.Rate` includes the current rate of transition `i`

`Rules.Prod{i}.Probs(j)` includes the current values of the `j`th probability of transition `i`.

We can thus update these values accordingly.
The two remaining inputs `T` and `X` include the current simulation time and the current population size respectively. If there are more than one species, `X` is an array such that `X(i)` is the population size of species `i`.

Example 2: Two interacting species

The predator/prey model of Lotka-Volterra[4] is usually defined using the following ODEs:

$$\frac{dPrey}{dt} = Prey(c_1 - c_2 Predator)$$
$$\frac{dPredator}{dt} = -Predator(c_3 - c_2 Prey)$$

These ODEs can be translated into the following eSTGs[1]:

$$Prey \xrightarrow{r_1} \{Prey, Prey\}_{p_1} | \{\phi\}_{ow}$$

$$Predator \xrightarrow{r_2} \{Predator, Predator\}_{p_2} | \{\phi\}_{ow}$$

with the following updating of the rates and probabilities:

$$r_1 = c_1 + c_2 \cdot |Predator|$$
$$r_2 = c_2 \cdot |Prey| + c_3$$
$$p_1 = \frac{c_1}{r_1}$$
$$p_2 = \frac{c_2 \cdot |Prey|}{r_2}$$

The following XML represents the transition rules:

```
<Program>
<ExecParams>
      <SimTime>
            10
      </SimTime>
      <Seed>
            0
      </Seed>
</ExecParams>

<FunHandleName> updating_LotkaVolterraExample </FunHandleName>
<Rule>
      <Prod>
            Prey -> 1 {Prey,Prey}_0.5 | {0}_0.5
      </Prod>
      <InitPop>
            900
      </InitPop>
</Rule>
<Rule>
      <Prod>
            Predator -> 1 {Predator,Predator}_0.5 | {0}_0.5
      </Prod>
      <InitPop>
            900
      </InitPop>
</Rule>
</Program>
```

There are now two transition rules for each of the species with simulation time of 10 and initial population size of 900 for both the Prey and the Predator.
The updating function is defined as follows:

updating_LotkaVolterra.m:

```
function [ Rules ] = updating_LotkaVolterra (Rules,T,X)
c1=2; c2=0.01; c3=5;

p1New = c1/(c1+c2*X(2));
```

```
    p2New = 1-c3/(c3+c2*X(1));
    r1New = c1+c2*X(2);
    r2New = c3+c2*X(1);

    p1New = min(1,max(0,p1New));
    p2New = min(1,max(0,p2New));
    Rules.Prod{1}.Probs(1)  = p1New;
    Rules.Prod{1}.Probs(2)  = 1-p1New;
    Rules.Prod{2}.Probs(1)  = p2New;
    Rules.Prod{2}.Probs(2)  = 1-p2New;

    Rules.Prod{1}.Rate = r1New;
    Rules.Prod{2}.Rate = r2New;

    end
```

In this case the updating function updates all the rates and the probabilities according to the definition.

Example 3: Internal states

In this example we simulate stem cell differentiation. SC (stem cells) divide symmetrically with rate 0.1, while self-renewing or differentiating with the same probability (50%), and $Diff$ (differentiated cells) can either proliferate (with probability 49%) or die (with probability 51%) at rate 1.
We define two internal states called MS, which simulates somatic mutations of Microsatellites(MS)[5] and Gen, which counts the number of generations since each differentiation event (the following rules are repeated for each internal state for readability):

MS internal state:

In this example, we define a vector of n variables $\overrightarrow{MS} = (MS_1, \dots MS_n)$, which correspond to the number of repeats in n MS loci in the DNA. In every cell division, the number of MS repeats for each locus changes according to the stochastic function f_{MS}, which can cause either a decrease or an increase of one repeat with probability p[6]:

$$f_{MS}(x) = \begin{cases} x + 1 \; with \; probability \frac{p}{2} \\ x - 1 \; with \; probability \frac{p}{2} \\ x \; otherwise \end{cases}$$

We define the following transition rules:

$$SC(\overrightarrow{MS} = \vec{x}_{MS}) \xrightarrow{0.1} \left\{SC\left(\overrightarrow{MS} = f_{MS}(\vec{x}_{MS})\right), SC\left(\overrightarrow{MS} = f_{MS}(\vec{x}_{MS})\right)\right\}_{0.5} |$$
$$\left\{Diff(\overrightarrow{MS} = f_{MS}(\vec{x}_{MS})), Diff((\overrightarrow{MS} = f_{MS}(\vec{x}_{MS}))\right\}_{0.5}$$
$$Diff(\overrightarrow{MS} = \vec{x}_{MS}) \xrightarrow{1} \left\{Diff(\overrightarrow{MS} = f_{MS}(\vec{x}_{MS})), Diff(\overrightarrow{MS} = f_{MS}(\vec{x}_{MS}))\right\}_{0.49} | \{\phi\}_{0.51}$$

Gen **internal state:**
The following transition rules include the internal state Gen, which counts the number of generations since each differentiation event:

$$SC \overset{0.1}{\rightarrow} \{SC, SC\}_{0.5} | \{Diff(Gen = 1), Diff(Gen = 1)\}_{0.5}$$
$$Diff(Gen = x) \overset{1}{\rightarrow} \{Diff(Gen = x + 1), Diff(Gen = x + 1)\}_{0.49} | \{\phi\}_{0.51}$$

The following XML represents the above transition rules and internal states:

```
<Program>
<ExecParams>
      <SimTime>
            50
      </SimTime>
      <Seed>
            0
      </Seed>
</ExecParams>

<FunHandleName> updating_Empty </FunHandleName>
<Rule>
      <Prod>
            SC -> 0.1 {SC,SC}_0.5 | {Diff,Diff}_ow
      </Prod>
      <InternalState>
            <Name> MS </Name>
            <InitVal> 0 </InitVal>
            <FuncHandleName> FuncUpdateMS </FuncHandleName>
            <DuplicateNum> 100 </DuplicateNum>
      </InternalState>
      <InitPop>
            5
      </InitPop>
</Rule>
<Rule>
      <Prod>
            Diff -> 1 {Diff,Diff}_0.49 | {0}_ow
      </Prod>
      <InternalState>
            <Name> MS </Name>
            <InitVal> 0 </InitVal>
            <FuncHandleName> FuncUpdateMS </FuncHandleName>
            <DuplicateNum> 100 </DuplicateNum>
      </InternalState>
      <InternalState>
            <Name> Gen </Name>
            <InitVal> 0 </InitVal>
            <FuncHandleName> FuncUpdateGen </FuncHandleName>
      </InternalState>
      <InitPop>
            5
      </InitPop>
</Rule>
</Program>
```

Note, that in this example the rates and probabilities are not updated and so the updating function is empty:

updating_Empty.m:

```
function [ Rules ] = updating_Empty( Rules,T,X)
%Empty
end
```

However, we now have to define an updating function for the internal states `MS` and `Gen`, namely:

FuncUpdateMS.m:

```
function [ newMS ] = FuncUpdateMS( MS )
%FuncUpdateMS updates the MS values according to the stepwise model
Mu = 1/10000; % the mutation rate (10^-4)

n = length(MS);
MutateVector = binornd(1,Mu,1,n);
MutDirection = binornd(MutateVector,1/2);
newMS = MS + (2*MutDirection - MutateVector);

end
```

FuncUpdateGen.m:

```
function [ newGen ] = FuncUpdateGen( Gen )
%FuncUpdateGen updates the generation
newGen = Gen + 1;

end
```

The input of an internal state updating function the current value of the internal state and the output is the updated value.

Example 4: Conditional Transitions

Conditional transitions enable to transform each individual instance into another species or to termination (death) if a certain condition on its internal states is met. Each individual instance is

examined upon each transition event and if its internal state follows the defined condition that individual is transformed to the defined target.
The following toy example shows three species, two types of "stem-cells" where one divides symmetrically and another one divides asymmetrically and a differentiated cell, which divides symmetrically or die. The asymmetric stem cells and the differentiated cells contain an internal state counter, which increases its value stochastically. The asymmetric cell includes a conditional transition which causes it to transform into a differentiated cell when the counter reaches a certain threshold, and the differentiated cells include a conditional transition which causes them to die when the counter reaches a second threshold. This is described by following rules:

$$SCASym \overset{0.1}{\rightarrow} \{SCASym, SCSym\}_1$$
$$SCSym(CounterStoch) \overset{0.1}{\rightarrow} \{SCSym, SCSym\}_1$$
$$Diff(CounterStoch) \overset{1}{\rightarrow} \{Diff, Diff\}_{0.5} | \{\phi\}_{0.5}$$

$$CounterStoch = CounterStoch + normrnd(1{,}0.1)$$

```
$if\ (CounterStoch > 5)\ then$
   $SCSym \rightarrow Diff$
$endif$

$if\ (CounterStoch > 10)\ then$
   $Diff \rightarrow \phi$
$endif$
```

where $normrnd(1{,}0.1)$ is a random sampling for a normal distribution with mean 1 and std of 0.1.

The following XML represents the above transition rules, internal states and conditional transitions:

```
<Program>
<ExecParams>
      <SimTime>
            100
      </SimTime>
      <Seed>
            0
      </Seed>
</ExecParams>
<FunHandleName> updating_Empty </FunHandleName>
<Rule>
      <Prod>
            SCASym -> 0.1 {SCASym,SCSym}_1
      </Prod>
      <InitPop>
            1
      </InitPop>
</Rule>
<Rule>
      <Prod>
```

```
            SCSym -> 0.1 {SCSym,SCSym}_1
      </Prod>
      <InternalState>
            <Name> CounterStoch </Name>
            <InitVal> 0 </InitVal>
            <FuncHandleName> FuncUpdateCounterStoch </FuncHandleName>
      </InternalState>
      <ConditionalTransition>
            <Condition> CounterStoch > 5 </Condition>
            <Transition> Diff </Transition>
      </ConditionalTransition>
      <InitPop>
            0
      </InitPop>
</Rule>
<Rule>
      <Prod>
            Diff -> 1 {Diff,Diff}_0.5 | {0}_ow
      </Prod>
      <InternalState>
            <Name> CounterStoch </Name>
            <InitVal> 0 </InitVal>
            <FuncHandleName> FuncUpdateCounterStoch </FuncHandleName>
      </InternalState>
      <ConditionalTransition>
            <Condition> CounterStoch > 10 </Condition>
            <Transition> {0} </Transition>
      </ConditionalTransition>
      <InitPop>
            0
      </InitPop>
</Rule>
</Program>
```

The structure of the conditional transition is such that the `<Condition>` tag includes any condition on the internal states of that species using MATLAB syntax and the `<Transition>` tag includes the name of the target species (to which the species is transformed into) or `{0}` for termination (death).
The updating function of the internal state `CounterStoch` is the following:

`FuncUpdateCounterStoch.m:`

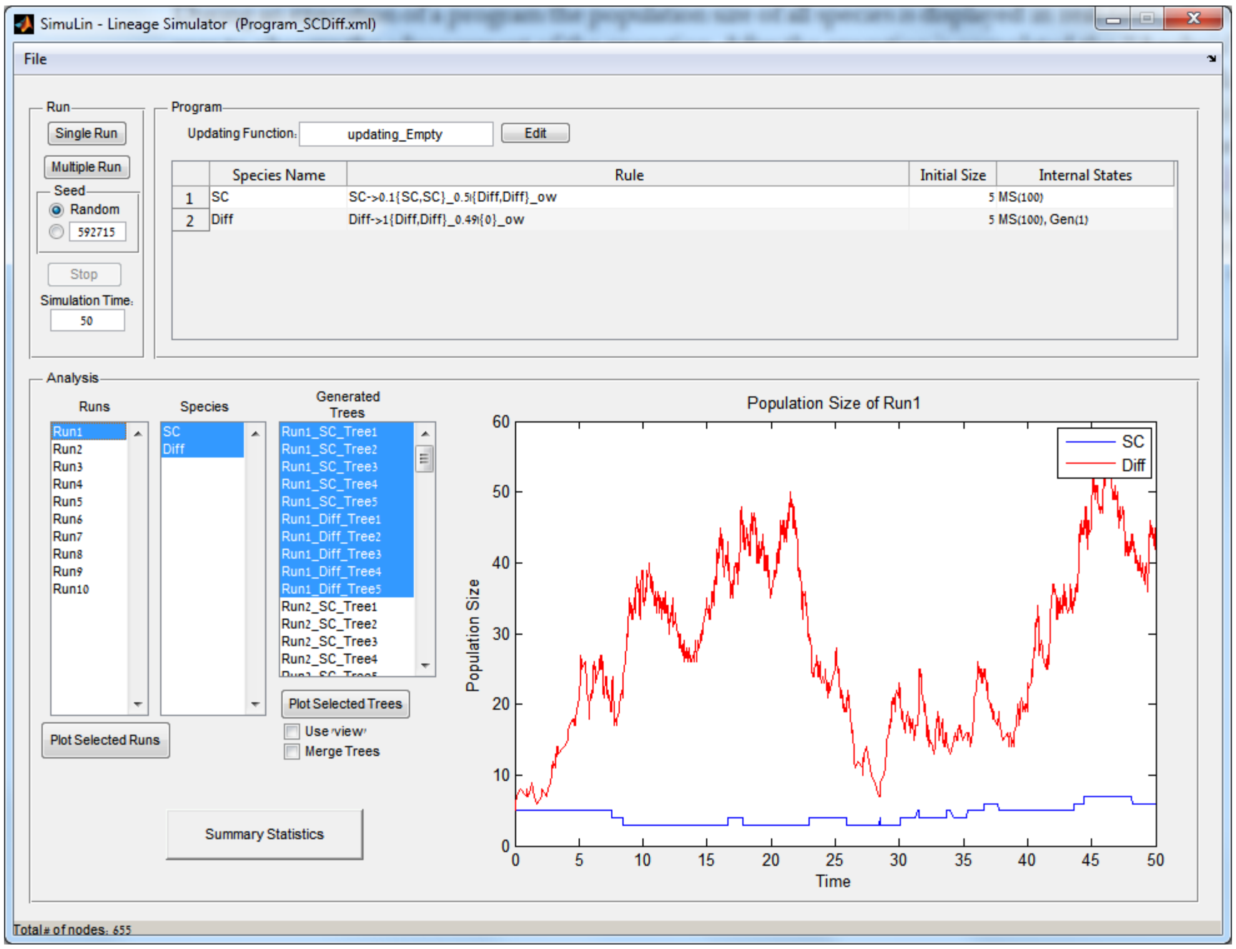
```
function [ newCounterStoch ] = FuncUpdateCounterStoch( CounterStoch )
%FuncUpdateCounterStoch updates the stochastic counter
newCounterStoch = CounterStoch + normrnd(1,0.1);

end
```

The counter internal state increases stochastically each time a transition event is executed.

Using the GUI Interface

In order to open the GUI interface, go to the eSTGt folder and type "eSTGt".

The main window of the GUI interface is presented in Figure 1. The window is divided into 3 panels, namely "Run", "Program", and "Analysis". The GUI enables to load an eSTG program, run it using various random seeds and analyze the results.



Seed: [✓ Random] 592715

Simulation Time: 50

Updating Function: updating_Empty

	Species Name	Rule	Initial Size	Internal States
1	SC	SC->0.1{SC,SC}_0.5{Diff,Diff}_ow	5	MS(100)
2	Diff	Diff->1{Diff,Diff}_0.49{0}_ow	5	MS(100), Gen(1)

Runs: Run1, Run2, Run3, Run4, Run5, Run6, Run7, Run8, Run9, Run10

Species: SC, Diff

Generated Trees: Run1_SC_Tree1, Run1_SC_Tree2, Run1_SC_Tree3, Run1_SC_Tree4, Run1_SC_Tree5, Run1_Diff_Tree1, Run1_Diff_Tree2, Run1_Diff_Tree3, Run1_Diff_Tree4, Run1_Diff_Tree5, Run2_SC_Tree1, Run2_SC_Tree2, Run2_SC_Tree3, Run2_SC_Tree4

Total # of nodes: 655

Figure 1. The main windows of the GUI interface after loading the program from Example 3 and executing it 10 times using different random seeds.

Loading a program

In order to load a program go to *File-->Load Rules…* and select the appropriate XML eSTG program file. In order to use one of the provided examples, select the XML file located in the "Programs" folder. After loading the file the "Program" panel presents the transition rules along with the initial population size and the Internal States (with duplication number in parenthesis). The updating function along with an "Edit" button is also presented in that panel.

Executing the program

After loading the program the "Run" panel is used to execute the program. The "Seed" radio buttons allow the user to choose either specific random seeds (for reproducing specific results) or randomly selected seeds. The "Single Run" button executes the program once whereas the "Multiple Run" button executes the program several times (number is selected by the user) using randomly selected seeds.

The simulation time is read from the program file but the user can change it before the first execution. The "Stop" button can be used to halt a specific execution in the middle.

Analyzing the execution results

During an execution of a program the population size of all species is displayed in real-time, allowing the user to observe the advancement of the execution. After the execution is completed the "Analysis" panel displays details regarding the different executions (or Runs), the species, and the generated trees. Runs are named using consecutive indices (e.g. Run1, Run2 etc.), species are named according to the `<Name>` tag in the input XML program file, and trees are named by concatenating the run name, the species name which is the root of that tree and a consecutive index of the generated tree. For example, in Example 3 above there are 5 initiating individuals of type SC and 5 initiating individuals of type $Diff$, such that each initiating individual generates a separated lineage tree. The tree name Run1_SC_Tree2 is the generated tree of the second SC individual from the first run (see Figure 1). When selecting a run in the "Runs" listbox the population size of the selected species (from the "Species" listbox) is displayed.

Buttons:

- "Plot Selected Runs": Displays MATLAB figures of the selected runs population size.
- "Plot Selected Tree": Displays the lineage trees of the selected trees. There are two display options:
    - "Use 'view'": Uses MATLAB's 'view' instead of 'plot', which displays an interactive figure of the tree.
    - "Merge Trees": Displays a merged tree consisting of the selected trees. The roots are connected using an auxiliary node with distance 0.

- "Summary Statistics": Opens a window that presents additional data statistics over all the runs (see Figure 2 and next section).

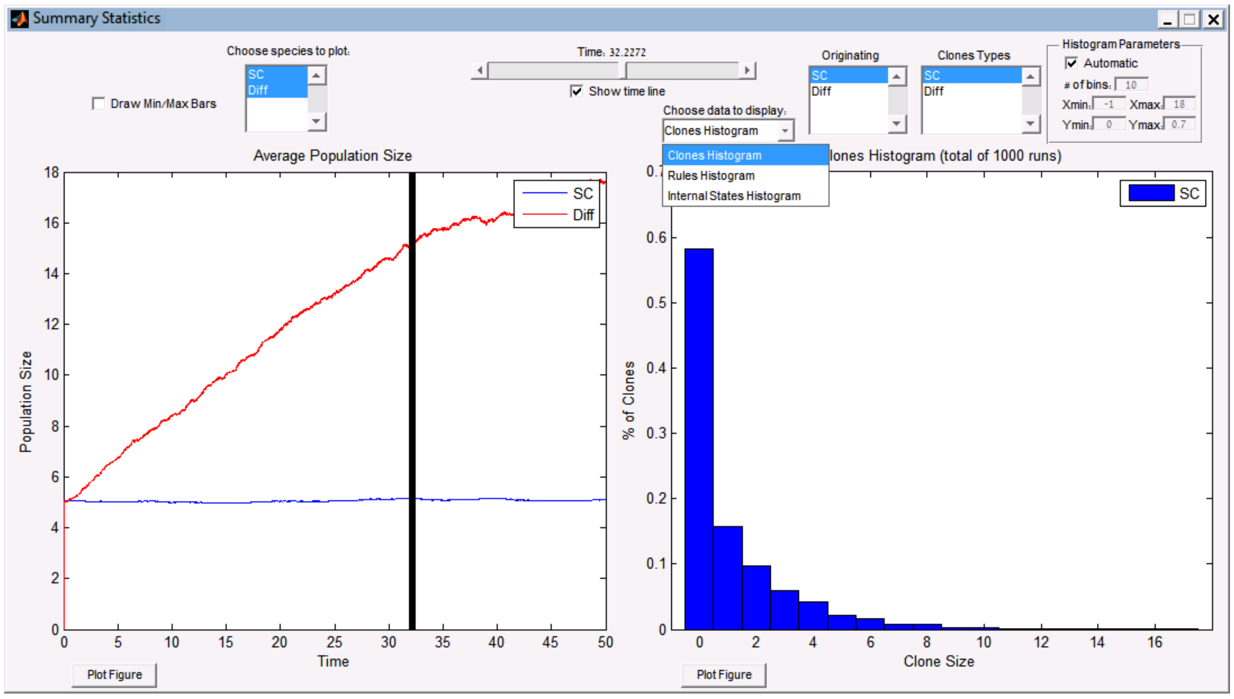
Summary Statistics

The "Summary Statistics" window presents various statistics over all simulation runs and enables to scroll over the simulation time in order to get snapshot of the statistics for each time point. The left figure presents the average population size over all simulation runs and the right figure can present three different types of statistics according to the selected option in the dropdown menu (see Figure 2). The options are:

- "Clones Histogram": A clone is defined as all individuals that are descendants of a single founder. The clone size is the number of living individuals of that clone. The histogram presents the percentage of clone sizes over all runs. The user can select the "Originating" species type and the species type of the resulting individuals within the clone. For example, in Example 3 an initiating SC can produce either SC or $Diff$ but initiating $Diff$ can produce only $Diff$ and so if one chooses $Diff$ in the "Originating" listbox and SC in the "Clones Types" listbox an empty histogram will be displayed.

- "Rules Histogram": Displays a histogram of the number of times each rule has been executed over all simulation runs.

- "Internal States Histogram": Displays a histogram of the Internal States values over all runs.



Time: 32.2272
☑ Show time line
☑ Automatic
# of bins: 10
Xmin: -1 Xmax: 18
Ymin: 0 Ymax: 0.7

Figure 2. Summary statistics of the different runs. The presented data is the result of 1000 stochastic simulations of the program in Example 3.

Menu Bar

The *File* menu bar consists of the following commands:

- *Load Rules* - Loads the eSTG rules from an XML file
- *Save Data to .mat file* - Saved the current session with the execution data to a MATLAB's formal .mat file.
- *Load Data from .mat file* - Loads a saved session.
- *Export Trees to Newick* - Saves the trees into text files using the Newick format. If the "Merge Trees" checkbox is marked it saves one Newick file with the merged trees.
- *Export Internal States* - Saves the Internal States values of each species type of each Run into a text file. Generates a separate file for each combination of Run, species type, and Internal State. In addition, generated a file that contains the corresponding individual names that match those of the exported trees.

Using the Command-line Interface

The command-line interface enables to run the simulations directly from MATLAB's command-line. The source code is located under the folder eSTGt. The following code uses one of the provided examples and executes the simulation 10 times using different random seeds (the current directory is the source code directory "eSTGt"):

```
ProgramFile = '../Programs/Example1/Program_LogisticVerhulst.xml';
Seeds = [1:10];
Rules = ParseeSTGProgram(ProgramFile);
[ Runs, RunsData ] = RunSim(Rules, Seeds, Rules.SimTime);
```

The first function `ParseeSTGProgram` receives as input an XML file containing the eSTG program and returns a struct that contains the corresponding rules. This struct is then passed to the second function `RunSim`, which receives as input the rules, an array of random seeds and the simulation time span. The output is a structure array of the runs for each seed and a common data structure for all runs. Fields of the output `Runs`:

- T: The events time points of the simulation.
- X: A matrix of the populations size of all species at all time points.
- R: The rule index that was executed at each time point.
- Nodes: A struct array of all the nodes of all species. The sturct holds all the information about the node including its name, type, parents, children, creation time and internal states.
- LiveNodes: For each time point lists the live nodes.

References

1. Spiro, A., Cardelli, L. & Shapiro, E. Lineage grammars: describing, simulating and analyzing population dynamics. *BMC Bioinformatics* **15**, 249 (2014).
2. Gonzalez, R.C. & Thomason, M.G. Syntactic pattern recognition : an introduction. (Addison-Wesley Pub. Co., Advanced Book Program, Reading, Mass.; 1978).
3. Gillespie, D.T. Stochastic Simulation of Chemical Kinetics. *Annual Review of Physical Chemistry* **58**, 35-55 (2007).
4. Fujii, T. & Rondelez, Y. Predator–prey molecular ecosystems. *ACS nano* **7**, 27-34 (2012).
5. Weber, J.L. & Wong, C. Mutation of human short tandem repeats. *Human molecular genetics* **2**, 1123-1128 (1993).
6. Valdes, A.M., Slatkin, M. & Freimer, N. Allele frequencies at microsatellite loci: the stepwise mutation model revisited. *Genetics* **133**, 737-749 (1993).