Table of Contents

Chapter 1

FFPilot simulation: conceptual overview

1.1 How to use FFPilot simulation

1.1.1 Use cases

FFPilot can be used to greatly accelerate the simulation of a wide variety of systems. The primary use case for FFPilot is the simulation of systems with two (or more) well defined meta-stable states. With the correct setup, FFPilot will ”ratchet” the simulation from one state to the other, yielding a great deal of information about the transition process and the probability landscape around and in-between the states.

In general, FFPilot should have a speed advantage over standard replicate simulation for any system that has a rough probability landscape with at least some large energetic barriers (equivalently, for any system encompassing rare events).

1.1.2 Non-use cases

In theory, FFPilot simulation should be faster (in terms of internal simulation time) than replicate sampling in all cases. However, for certain types of systems the benefit of using FFPilot is expected to be minor:

- **Systems with flat landscapes/no rare events**
  Systems in which the probability landscape is nearly flat, in that it lacks any significant barriers. In other words, systems with 0 fixed points.
- **One-state systems**
  Systems that spend virtually all of their time in the neighborhood of a single point in their state space. In other words, systems with 1 fixed point.

For these types of systems replicate simulation will likely be faster (in terms of wall clock time) than FFPilot, due to various optimizations in the current version of Lattice Microbes. Thus, it is better to avoid the extra setup involved with using FFPilot when modeling these types of systems.

1.2 FFPilot specific concepts

FFPilot simulation is straightforward to set up, given that you at least know the rough locations (in terms of your system's state space) of the steady states of interest.

1.2.1 Order parameters

FFPilot uses an order parameter to distinguish one stable state from another, and to measure progress along a transition path. If FFPilot is thought of as driving a system from one state to another, then the order parameter defines the direction in which that driving occurs.

An order parameter is a function $\mathcal{O} : \left\{\mathbb{N}^{s}_{\geq 0}\, ,\ \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^{O}\right\}$ of the species counts and time of a system:

$$\mathcal{O}\left(s_0\, ,\ \cdots\, ,\ s_s\, ,\ t\right) = \left\{o_0\, ,\ \cdots\, ,\ o_O\right\}$$

An order parameters can thought of as a quantitative, mathematically defined version of a reaction coordinate.

1.2.2 Tilings

A tiling is a set of bins that spans the state space of a system. A tiling is defined in terms of an order parameter, a set of edges, and one or more initial basins (from which to start trajectories during phase zero). Multiple basins can be defined in order to carry out both forward and reverse simulation in a single run of Lattice Microbes ES (LMES).

Tilings can be defined in terms of a set of bin edges (also sometimes referred to as a set of interfaces). An FFPilot simulation will have as many

- The edges are held in a 1D array called (appropriately enough) Edges. - The initial basins are held in an N x D array called Basins, where N is the count of basins and D is the total number of unique chemical species in your system. - Each row in Basins represents a different basin, and each column in Basins holds the count of one particular chemical species in the model. - In other words, a basin is a single coordinate in the state space of the model. - Each initial basin should either be in front of the zeroth edge or behind the last edge. - Mathemtically, this means that one of

$$\mathcal{O}(\text{basin}) < \mathcal{O}(\text{Edges}[0])$$

or

$$\mathcal{O}(\text{basin}) > \mathcal{O}(\text{Edges}[-1])$$

should be true, where $\mathcal{O}$ is the order parameter of the tiling.

The rows in the ‘Basins‘ array determine where trajectories are started from during phase zero. A separate pilot and production stage (ie a whole separate FFPilot simulation) is run for each row in ‘Basins‘. Thus, setting two different basins is a convenient way to run the simulations required to calculate the MFPT of both the forward switch (low A → high A) and the reverse switch (high A → low A) of the self regulating gene.

Chapter 2

Calculating values of interest from FFPilot output: theory

2.1 Mean first passage time

The mean first passage time $MFPT_i$ to each edge i of the tiling is calculated in terms of the weights w_i as:

$$MFPT_i = \begin{cases} w_0 & \text{for } i = 0 \\ \frac{w_0}{\prod_{j=1}^{i} w_i} & \text{for } i > 0 \end{cases} \tag{2.1}$$

Thus, the overall $MFPT$ from the low A state to the high A state can be calculated as:

$$MFPT = MFPT_N = \frac{w_0}{\prod_{j=1}^{N} w_i} \tag{2.2}$$

where N is the index of the final edge.

In terms of the phase weights w_i, the formula for the mean first passage time to the jth edge is:

$$\frac{w_0}{\prod_{i=1}^{j} w_i}$$

2.2 Landscapes produced by FFPilot

One of the primary goals of stochastic simulation of biological systems is to calculate the probability landscapes of those systems. For nonequilibrium steady-state systems, calculating the landscape is equivalent to solving the system's master equation. From it you can obtain complete information about the occupancies and fluxes of the system.

When working with FFPilot simulation output, the calculation of the complete unbiased probability landscape of a system can be thought of as the recursive process of building up larger histograms (in terms of their information content) from smaller ones. Of course the final, complete landscape is the one of primary interest, but the phase, stage, and transition region landscapes that you build in the process can be helpful for investigating the fine details of a system or of the simulation process itself.

2.2.1 FFPilot phase landscapes

The first step in assembling the complete landscape hist is also the simplest. You just take the state samples collected during the FFPilot simulation and group them by phase. You then just bin them together into sparse histograms, one per phase run during a production stage.

2.2.2 FFPilot stage landscapes

All of the phase $i > 0$ hists (the phase 0 hists are dealt with later) are separated into groups based on their originating stage. Next, the phase hists in each group are combined via a simple weighted sum of their values. The weights, which we'll call the landscape weights $\mathbb{l}_i$, are calculated by the following formula:

$$\mathbb{l}_i = \begin{cases} \frac{1.0}{n_1} & \text{for } i = 1 \\ \\ \frac{\prod_{j=1}^{i-1} w_j}{n_i} & \text{for } i > 1 \end{cases} \tag{2.3}$$

where w_i is the phase i weight and n_i is the total count of trajectories run in phase i[*].

The indices in eqrefeq:landweight are a bit confusing, so to give you a concrete example say that you ran a simulation that had 4 phases per stage. Phase 0 doesn't get a landscape weight, so your three $\mathbb{l}_i$ values would be:

$$\mathbb{l}_1 = \frac{1.0}{n_1}$$
$$\mathbb{l}_2 = \frac{w_1}{n_2}$$
$$\mathbb{l}_3 = \frac{w_1 w_2}{n_3}$$

Due to the boundary conditions imposed in phases $i > 0$, the stage hist will only cover the region of state space that was spanned by the tiling[†] used to setup the FFPilot simulation. We will refer to this span as the transition region, since by construction it will lie in-between two stable fixed points. The ith stage hist can be thought of as the conditional probability landscape of the transition region, given that the ith basin was the most recently visited. In other words, if a trajectory is in (or was most recently in) basin i, the ith stage hist will be effectively the instantaneous probability landscape for that trajectory.

[*]Eq 2.3 only applies if you're taking state samples at a constant interval. Though discussion of a variable sampling time step is outside of the scope of this tutorial, the landscape weight formula can be modified [1] to allow for one.

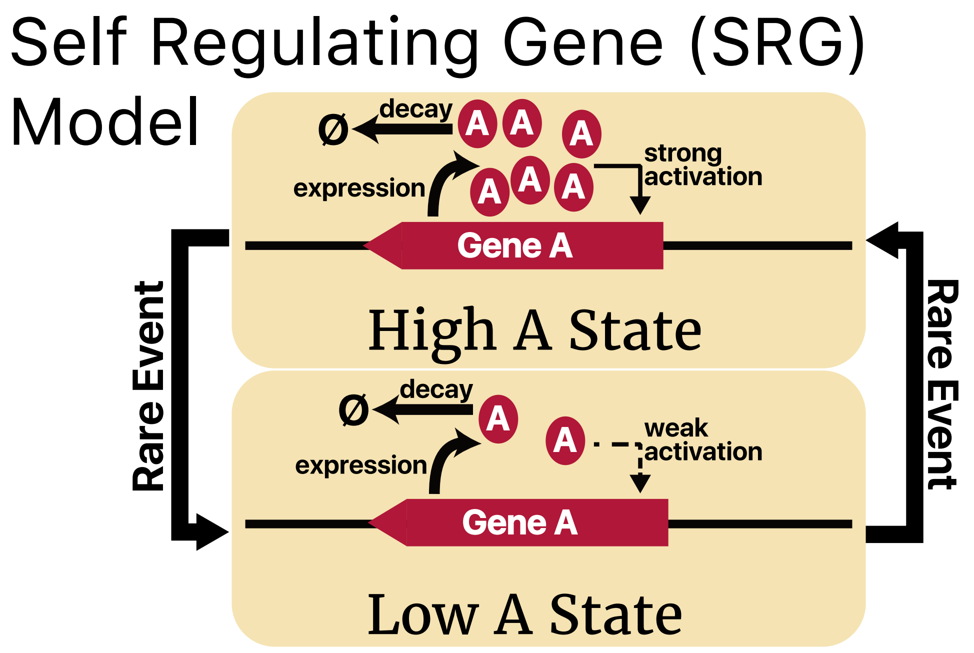
[†]Technically, what we mean by spanned region is all points x in state space such that $\lambda_0 <= \mathcal{O}(x) < \lambda_N$

Chapter 3

Using FFPilot to calculate mean first passage time (MFPT): self regulating gene

3.1 Overview

In this section I'll go over how to use FFPilot to calculate the mean first passage time (MFPT) of the self-regulating gene (SRG). SRG consists of a single protein, which I'll call protein A. Protein A participates in a positive feedback loop in which it upregulates its own production. Additionally, it experiences constant degradation. These properties make SRG bistable, and it will stochastically switch from one state, with low levels of protein A, to another, with high levels of protein, and back again over time.



Species	Description
A	protein

Reactions	Rates	Description
$\varnothing \longrightarrow \text{A}$	$k_{low} + (k_{high} - k_{low}) \frac{\text{A}^h}{k_{50}^h + \text{A}^h}$	expression
$\text{A} \longrightarrow \varnothing$	βA	decay

Starting from a SRG model file included with this tutorial, `notebooks/self_regulating_gene.sbml`, I'll show you how to set up the simulation input file, execute the simulation, and analyze the output. If you execute the example commands and code exactly as written, you should end up with a new directory `notebooks/srg_mfpt/data` that contains `srg.lm`, the simulation input file, and `srg_out.sfile`, the simulation output file. Pre-made example input and output files can be found in `notebooks/srg_mfpt/data_worked`.

3.2 Input file setup

The input file for an FFPilot simulation is the same as for standard (*i.e.* direct sampling) simulation. However, some extra information is required, in the form of an order parameter and a tiling.

3.2.1 Convert .sbml -> .lm

Models of biochemical systems are often distributed in the SBML format. These can be converted to the `.lm` format required by Lattice Microbes using the `lm_sbml_import` utility.

Open a terminal and `cd` to the `notebooks/srg_mfpt` directory. You will then execute the following commands:

```
user@host:srg_mfpt$ mkdir -p data
user@host:srg_mfpt$ lm_sbml_import data/srg.lm ../self_regulating_gene.sbml
```

The first command creates a separate `notebooks/srg_mfpt/data` directory which we will use to hold any simulation input/output that we produce during this section of the tutorial. The last command converts `notebooks/self_regulating_gene.sbml` file using `lm_sbml_import` and then places the resulting `.lm` file at `notebooks/srg_mfpt/data/srg.lm`.

3.2.2 Add an order parameter

If fed into LMES in its current form, `srg.lm` could be used to run a standard replicate simulation. However, in order to use `srg.lm` as input for an FFPilot simulation, we'll have to first open it up and add some extra information: an order parameter and a tiling. We'll start with the order parameter.

Internally, `.lm` files are based on the `hdf5` format. This means that all of the existing `hdf5` software tools can directly interact with `srg.lm`, just the same as for any `hdf5` file. We'll be using two of these `hdf5` tools in this tutorial:

`h5py`

A Python package that can open and/or modify `hdf5` files from within Python code. We'll be using `h5py` to modify the input file/add the extra information.

`h5dump`

A command line tool that can be used to dump the contents of a `hdf5` file in plain text to the `stdout` of a terminal. We'll be using `h5dump` to show the correct format for the extra input required by FFPilot.

The order parameter that we will use with SRG, which I will call $\mathbf{A}$, will just be the count of the single chemical species in the system, protein A. Order parameter $\mathbf{A}$ is a linear combination of species counts and can thus be represented in LMES as a linear order parameter. The following Python script will open up `srg.lm` and add the definition of $\mathbf{A}$ in the required format:

```python
import h5py

with h5py.File('data/srg.lm') as f:
    # remove any existing OrderParameters group
    if 'OrderParameters' in f.keys():
        del f['OrderParameters']

    # create the OrderParameters group
    oparams = f.create_group('OrderParameters')

    # add the subgroup for order parameter 0
    oparam0 = oparams.create_group('0')

    # set this order parameter's ID to 0, and its Type to 0
    oparam0.attrs['ID'] = 0
    oparam0.attrs['Type'] = 0
```

```python
# add the SpeciesIDs. This array should always be of type `int`
speciesIDs = np.array([0], dtype=int)
oparam0.create_dataset('SpeciesIDs', data=speciesIDs)

# add the SpeciesCoefficients. This array should always be of type `float`
speciesCoefficients = np.array([1.0], dtype=float)
oparam0.create_dataset('SpeciesCoefficients', data=speciesCoefficients)
```

The above script adds a new group, `OrderParameters`, to `srg.lm`, the contents of which we can inspect using `h5dump` by opening a terminal, `cd`-ing to `notebooks/srg_mfpt`, and then running the following command:

```
user@host:srg_mfpt$ h5dump --group=OrderParameters data/srg.lm
```

The dump will show that the `OrderParameters` group contains one subgroup, `0`:

```
HDF5 "data/srg.lm" {
GROUP "OrderParameters" {
   GROUP "0" {
```

Group `OrderParameters/0` contains the definition of $\mathbf{A}$. Two attributes are set on group `0`, `ID` and `Type`:

```
      ATTRIBUTE "ID" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
         }
      }
      ATTRIBUTE "Type" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
         }
      }
```

These attributes, which are required in every order parameter definition, do the following:

`ID` This attribute (which should match the name of the `OrderParameters` subgroup on which it is set) is the name by which LMES will refer to $\mathbf{A}$ internally.

`Type`

This attribute tells LMES what kind of order parameter this is, and how it should interpret the order parameter's definition when calculating its value. A `Type` of `0` tells LMES that $\mathbf{A}$ is a linear order parameter.

Inside of group `0` are two 1D arrays (by convention, `hdf5` tools refer to arrays as datasets), `SpeciesIDs` and `SpeciesCoefficients`:

```
      DATASET "SpeciesCoefficients" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
         DATA {
         (0): 1
         }
      }
      DATASET "SpeciesIDs" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
         DATA {
         (0): 0
         }
      }
```

LMES uses the values in these arrays to define the formula for the value of an order parameter. For any linear order parameter $\mathcal{O}$, LMES will use the following formula to calculate its value:

$$\mathcal{O} = \sum_{i=0}^{\texttt{len(SpeciesIDs)}-1} \texttt{Counts[SpeciesIDs[i]]} * \texttt{SpeciesCoefficients[i]} \tag{3.1}$$

where `Counts` is a 1D array containing the present count of each species in the system being simulated. Eq 3.1 can be simplified considerably for $\mathbf{A}$:

$$\mathbf{A} = \texttt{Counts[SpeciesIDs[0]]} * \texttt{SpeciesCoefficients[0]}$$
$$\mathbf{A} = \texttt{Counts[0]}$$

3.2.3 Add a tiling

Now we add a tiling to the self regulating gene. The tiling will be defined in terms of order parameter $\mathbf{A}$. In the previous section we added $\mathbf{A}$ to `srg.lm` and gave it an `ID` of `0`. The following Python script will open up `srg.lm` and add a new tiling with an `ID` of `0`:

```python
import h5py

with h5py.File('data/srg.lm') as f:
    # remove any existing Tilings group
    if 'Tilings' in f.keys():
        del f['Tilings']

    # create the Tilings group
    tilings = f.create_group('Tilings')

    # add the subgroup for tiling 0
    tiling0 = tilings.create_group('0')
    # set this tiling's ID to 0, its OrderParameterID to 0, and its Type to 0
    tiling0.attrs['ID'] = 0
    tiling0.attrs['OrderParameterID'] = 0
    tiling0.attrs['Type'] = 0

    # add the Basins. There is 1 chemical species in this model, and we are
    # going to set 2 initial basins, so Basins will be a 2 x 1 array
    basins = np.zeros((2,1), dtype=int)
    basins[0,0] = 10
    basins[1,0] = 160
    tiling0.create_dataset('Basins', data=basins)

    # add the Edges
    edges = np.linspace(23.0, 150.0, num=13)
    tiling0.create_dataset('Edges', data=edges)
```

Once the script has run, we can inspect the contents of the newly added `Tilings` group in `srg.lm` by opening a terminal, `cd` -ing to `notebooks/srg_mfpt`, and then running the following command:

```
user@host:srg_mfpt$ h5dump --group=Tilings data/srg.lm
```

The dump shows one subgroup, `0`, in the `Tilings` group:

```
HDF5 "data/srg.lm" {
GROUP "Tilings" {
   GROUP "0" {
```

Group `Tilings/0` has three required attributes set on it:

```
      ATTRIBUTE "ID" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
         }
      }
      ATTRIBUTE "OrderParameterID" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
         }
      }
      ATTRIBUTE "Type" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SCALAR
         DATA {
         (0): 0
         }
      }
```

the description of which are as follows:

`ID` This attribute (which should match the name of the `Tiling` subgroup on which it is set) is the name by which LMES will refer to this tiling internally.

`Type`
A tiling `Type` of `0` tells LMES that this is a 1D linear tiling[*].

`OrderParameterID`
The `ID` of the order parameter (in this case, **A**) that will be used in the definition of this tiling. Specifically, the placement of the edges that separate the tiles/bins of this tiling will be specified in terms of the tiling's order parameter.

Inside of group `Tilings/0` are a 2D array, `Basins`, and a 1D array, `Edges`:

```
      DATASET "Basins" {
         DATATYPE  H5T_STD_I64LE
         DATASPACE  SIMPLE { ( 2, 1 ) / ( 2, 1 ) }
         DATA {
         (0,0): 10,
         (1,0): 160
         }
      }
      DATASET "Edges" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 13 ) / ( 13 ) }
         DATA {
         (0): 23, 33.5833, 44.1667, 54.75, 65.3333, 75.9167, 86.5, 97.0833,
         (8): 107.667, 118.25, 128.833, 139.417, 150
         }
      }
```

These arrays are used in the following way:

`Edges`
Each entry in this array holds a single coordinate given in terms of the tiling's order parameter. LMES builds up a tiling by placing edges at these coordinates. In turn, the tiles/bins themselves are defined as the space in between any two adjacent edges. The size of the `Edges` array is directly related to the number of separate phases run during an FFPilot simulation. During each stage there will be exactly as many phases executed as there are edges in the tiling used to set up the stage.

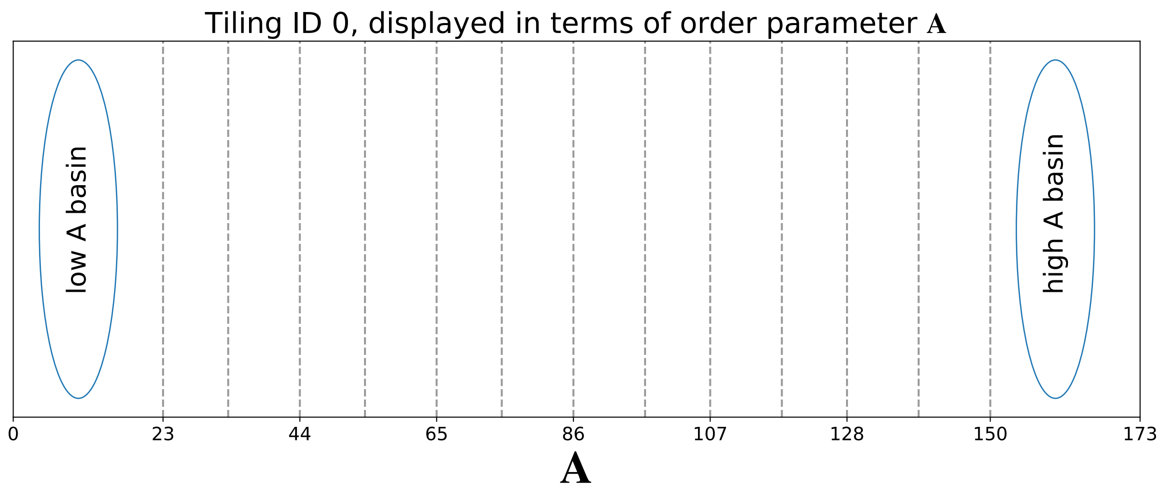
`Basins`
An N x D array, where N is the count of basins and D is the total number of unique chemical species in your system. The basins (which are defined in terms of a model's complete state space) are the points in the state space of a system from which the initial trajectories of each

[*]1D linear is the only kind of tiling currently implemented in LMES. Thus, for now, this is just a placeholder value that should always be set to 0. In the future, more complex tilings, such as ones based on the Voronoi tessellation [2], may be implemented.

stage (*i.e.* the phase zero trajectories) will be launched. Each row in `Basins` is treated as a separate basin.

The rows in the `Basins` array determine where trajectories are started from during phase zero. One complete self-contained FFPilot simulation (*i.e.* a pilot stage following by a production stage) will be executed for each basin specified in the simulation input's tiling. Thus, setting two different basins is a convenient way to run the simulations required to calculate the MFPT of both the forward switch (low A → high A) and the reverse switch (high A → low A) of the SRG model.

The Python script we used to create our tiling will have set up two basins. The initial basin is at 10 counts of protein A, corresponding to the low A state of SRG. This is located in front of the zeroth edge of tiling (which sits at 23 counts of A). The other basin is at 160 counts of protein A, corresponding to the high A state of SRG. This basin is located behind the last edge of the tiling (which sits at 150 counts of A). The following figure gives a sense of what the tiling actually looks like:



3.2.4 (advanced) FFPilot batch mode (multiple tilings)

In normal usage, the input file for an FFPilot simulation should have exactly one tiling. However, FFPilot simulations can be run in large batches by specifying multiple tilings. In this case, each separate tiling will be treated as if it describes a separate simulation. In other words, an FFPilot simulation run using an input that contains M separate tilings is equivalent to running a batch of M separate FFPilot simulations.

NB: when creating an input file to use with FFPilot batch mode, each tiling should have its own unique `ID`.

3.2.5 Customize FFPilot simulation options

FFPilot is designed to require as little user input as possible. All of the simulation options/parameters that FFPilot uses have reasonable (and, in some cases, adaptive) default values. On the other hand, FFPilot supports a number of options that can be used to control many different aspects of the simulation. In `.lm` input files, options are set as attributes on the top level `Parameters` group.

NB: `lmes` only recognizes options set with values of type `str`. All non-string options (*i.e.*`int`, `bool`, etc.) should be converted to `str` before they are set.

We'll set up some options for our FFPilot SRG simulation using the following python script (which contains several examples of `str` conversion):

```python
import h5py

with h5py.File('data/srg.lm') as f:
    # NB: convert all option values to str before setting them

    # turn on output of pilot stage data
    f['Parameters'].attrs['ffluxPilotOutput'] = str(True)

    # set a target error goal for the simulation as a whole. Defaults to .05
    f['Parameters'].attrs['errorGoal'] = str(.10)
```

In the above script, we set three options:

`ffluxPilotOutput`
: `bool`, defaults to `False`. Turns on output of records from any pilot stages run during the simulation. Normally only simulation output from the production stages is saved, and the data from every pilot stage is discarded once its corresponding production stage has been set up. When this flag is set to `True`, however, output from both the pilot and the production stages is saved.

`errorGoal`
: Percentage, specified as a `float` in the range `(0-1.0]`, defaults to `.05`. Sets a goal for the (percent) error in the overall basin-to-basin MFPT calculated by each production stage. Based on the outcome of the preceding pilot stage, FFPilot automatically determines how many trajectories to launch in each phase of a production stage in order to achieve a sampling error at or below the set error goal. Set `errorGoal` to `.01` for a high accuracy simulation, or set it to `.10`+ for a relatively fast simulation.

3.3 Running the simulation

Now that the FFPilot input is prepared, you can execute the simulation by opening a terminal, `cd`-ing to `notebooks/srg_mfpt`, and entering the following command:

```
user@host:srg_mfpt$ lmes -fflux -f data/srg.lm
```

As shown above, running an FFPilot simulation requires two flags to be passed to the LMES executable:

`-ffpilot`
: Tells LMES to run in FFPilot mode. Without this flag a default replicate simulation would be run instead.

`-f <input-path>`
Tells LMES that the simulation input file can be found at `<input-path>`.

The path at which the simulation output will be saved is automatically determined from the input path. In this case, that means the simulation output will be saved to `data/srg_out.sfile`. Alternatively, you can explicitly set the simulation output path by specifying the `-fo <output-path>` flag on the command line when you execute LMES.

Since this simulation is set up using a relatively loose 10% error goal, it should run to completion fairly quickly. When run on my laptop (3.1 GHz, 4 CPU cores), the simulation finishes in about 30 seconds.

3.3.1 (advanced) Preserving the simulation log

Normally, the `lmes` simulation log gets dumped directly to the terminal as the simulation runs. You can instead preserve the log for later perusal by running `lmes` with a slightly modified command:

```
user@host:srg_mfpt$ lmes -fflux -f data/srg.lm > data/srg.log
```

where we've saved the simulation log to `notebooks/srg_mfpt/data/srg.log` using `>`, the Linux redirection operator.

One downside of redirecting the log is that it will prevent any progress messages from getting printed out to the terminal, making it hard to tell if the simulation is running correctly. One workaround is to monitor the log file in real time using the `watch` command while the simulation adds to it:

```
user@host:srg_mfpt$ watch tail -n 25 data/srg.log
```

where the `-n` flag controls the number of lines that are displayed.

3.4 Analyzing the log

As an `lmes` simulation runs, an informative log is dumped to `stdout`. Aside from helping to monitor simulation progress, a large quantity of information about the outcome of an FFPilot simulation can be gleaned from the simulation log alone. For example, the MFPT estimate is printed out at the end of every stage, eliminating the need to dig around in the actual simulation output file for this value. As well, the log contains some unique metadata about the simulation, such as the total elapsed wall-clock time.

3.4.1 FFPilot simulation progress log messages

FFPilot simulations, like all `lmes` simulations, begin with a kickoff phase, during which input is parsed and all of the parallel processes are initialized and coordinated. Once the kickoff is complete (usually only a few seconds when running `lmes` on a single computer), an FFPilot simulation begins in earnest with the start of the pilot stage, as signaled by the following line getting printed to the log:

```
Forward Flux stage 0 started (tiling_id: 0, basin_id: 0, stage_type: Pilot)
```

This type of progress message is printed out at the start of every simulation stage. Each comma-separated term in between the parentheses holds a piece of information about the current simulation stage:

`tiling_id`
: This gives the `ID` of the tiling used to set up the simulation stage. This matches the `ID` of the tiling we created earlier (see Sec 3.2.3), and confirms that it was used to set up this simulation stage.

`basin_id`
: This corresponds to the row index of `Basins` from which the basin species counts for this stage were taken. These basin species counts will be used to initialize phase 0 trajectories during this stage.

`stage_type`
: Tells you whether this is a pilot or a production stage. One of each kind of stage is run for each basin in each tiling present in the simulation input.

Another kind of progress message is printed out at the start of each phase zero:

```
Forward Flux phase 0 started (zeroth_edge: 23.00, \
    phase_limit: FORWARD_FLUXES >= 10000.00)
```

The terms in the parentheses give some information about how this phase will be executed:

`zeroth_edge`
: The coordinate of the tiling edge (given in terms of the tiling's order parameter) closest to the basin in which this phase's trajectories are initialized. Each time a phase zero trajectory crosses the `zeroth_edge` while traveling away from its starting basin, the simulation counts this as a single forward flux event. Additionally, the state that the trajectory was in when it fluxed forward is added to the dictionary of starting states that will be used to initialize trajectories during phase 1.

`phase_limit`
: Describes the condition that must be fulfilled in order for this phase to be considered complete. This particular `phase_limit` means that 10,000 forward flux events (summed across all phase zero trajectories) need to be observed before this phase will be terminated. Phase zero is always run using a `FORWARD_FLUXES` type `phase_limit`.

A slightly different progress message is printed out at the start of each phase $i > 0$:

```
Forward Flux phase 4 started (starting_edge: 54.75, goal_edge: 65.33, \
    phase_limit: FORWARD_FLUXES >= 10000.00)
```

where the values in the parentheses mean:

`starting_edge`
: The edge along which lies all of the starting states used to initialize trajectories during this stage.

`goal_edge`
: Once a trajectory has been launched from `starting_edge` it will continue to run until it either crosses the `zeroth_edge` (as specified in the most recent phase zero progress message) or this `goal_edge`. In either case the simulation is immediately terminated. However, if the trajectory was stopped because it crossed the `goal_edge`, the simulation counts this as a forward flux event. As well, the state the trajectory was in when it fluxed forward is added to the dictionary of starting states that will be used to initialize trajectories in the next phase.

The `phase_limit` term means the same thing in the phases $i > 0$ progress messages as it does in the phase zero progress message. However, unlike phase zero, the type of `phase_limit` used to run any given phase $i > 0$ varies depending on whether the phase is part of a pilot or a production stage. Essentially, the type of the `phase_limit` determines the phase's termination condition:

`FORWARD_FLUXES >= n`
: The type of `phase_limit` used during pilot stages. Each phase $i > 0$ will only be considered complete once `n` forward flux events (*i.e.* the count of trajectories that reached the `goal_edge`) have been observed, regardless of how many trajectories have been launched in total.

`TRAJECTORY_COUNT >= n`
: The type of `phase_limit` used during production stages. Each phase $i > 0$ is considered complete once the total number of trajectories that have been launched and run to termination is `n`.

3.4.2 Stage outcome log messages

Every time a pilot stage finishes, some of the highlights from its output are added to the log file:

```
The phase costs are:
[4.62, 0.151, 1.61, 2.41, 2.72, 2.09, 1.65, 1.22, 0.967, 1.01, 0.951, 1.25, 1.5]
The phase weight sample variances are:
[152, 0.018, 0.12, 0.228, 0.245, 0.155, 0.0665, 0.0211, 0.00473, 0.00296, \
    0.000693, 0.000495, 0.000396]
Conservative estimates of the phase weights are:
[4.62, 0.0179, 0.137, 0.345, 0.56, 0.8, 0.922, 0.974, 0.993, 0.995, 0.998, \
    0.998, 0.999]
Attempting to achieve error goal 0.05 (confidence level 0.95) \
    with the following optimized trajectory counts:
[32385, 497266, 51746, 23142, 14016, 9022, 5917, 3830, 2210, 1792, 1148, 1000, 1000]
```

phase costs
: The ith entry of this array holds the average cost, in terms of the internal simulation time, of collecting a single sample (*i.e.* observation of a forward flux event) during phase i.

variances
: The ith entry of this array holds the variance of the samples collected during phase i. The samples in question are used to estimate the phase weights at the end of each phase.

conservative weight estimates

The ith entry of this array holds the estimate of phase weight i that is calculated at the end of each phase i. of the samples collected during phase i.

The values in these first 3 arrays are used to parameterize the FFPilot optimizing equation. The FFPilot optimizing is used to predict the optimal simulation plan (*i.e.* count of trajectories to launch in each phase) for the upcoming production stage. By optimal, I mean the simulation plan that will achieve the specified error goal with the least computational effort possible.

The reason why the weight estimates shown above are referred to as ”conservative” is that, following their initial calculation, they have each been run through a set of biased estimators. These biased estimators have been designed such that when the conservative weight estimates are used to parameterize the FFPilot optimizing equation, the optimizing equation is in turn slightly biased towards overestimating the number of trajectories required to achieve the error goal. This has the beneficial effect of helping to ensure overall simulation accuracy, albeit at a small cost in computational efficiency.

The last array in the pilot stage output log message contains the prediction of the FFPilot optimizing equation, given the values in the first 3 arrays:

optimized trajectory counts

The trajectory counts in this array are used to set up the `phase_limits` during the pilot stage. In other words, each phase $i > 0$ of the production stage will be run until the count of trajectories that have run to completion reaches `optimized_trajectory_counts[i]`.

Each production stage also adds some the highlights from its results to the log once it finishes:

```
The phase costs are:
[4.72, 0.151, 1.58, 2.42, 2.71, 2.08, 1.66, 1.28, 1.02, 1.03, 0.974, 1.24, 1.49]
The phase weights are:
[4.72, 0.0186, 0.145, 0.353, 0.573, 0.814, 0.926, 0.977, 0.993, 0.998, 1, 1, 1]
The first passage times to each tile edge are:
[4.72, 254, 1.76e+03, 4.97e+03, 8.67e+03, 1.07e+04, 1.15e+04, 1.18e+04, 1.19e+04, \
    1.19e+04, 1.19e+04, 1.19e+04, 1.19e+04]
The overall first passage time from the starting basin to the last tile edge is:
1.19e+04
```

phase costs and weights

Same as from the pilot stage, but produced by the more accurate production stage.

mean first passage times

The ith entry in this array is the expected value of the time it takes to get from the simulation’s starting basin to the ith edge of the tiling.

overall mean first passage time

The expected value of the time it takes to get from the simulation’s starting basin to the ending basin. Equivalent to the final entry in the MFPT array printed above. The basins can be thought of as the metastable states of the system, so the overall MFPT can also be thought of as the inverse of the switching rate between those states.

3.5 Browsing FFPilot output with dumpSFile

Output from an FFPilot simulation takes the form of an `SFile`. By default, given that your simulation input is named `<fname>.lm` the output is saved to `<fname>_-_out.sfile` (this can be overridden by setting the `-fo <output-name>` cmd line option when running LMES). `SFile` is a binary file format that holds information in the form of records. Each record begins with a short metadata section (record name, record type, data size in bytes) followed by a data section. For LMES output in the `SFile` format, the data section takes the form of a serialized protocol buffer message.

Although `SFiles` are not human readable, the `dumpSFile` tool provided by the `robertslab` Python package can be used to easily view the contents of any `SFile`. For each record in an `SFile`, `dumpSFile` converts the metadata into a human readable format, deserializes and formats the data portion, and then prints the results to `stdout`. The following command will dump the entire contents of `data/srg_out.sfile` to the terminal:

```
user@host:srg_mfpt$ dumpSFile data/srg_-_out.sfile
```

The output will look something like this:

```
data_worked/srg_-_out.sfile

/Simulations/0/Tilings/0/Basins/0/Stages/Pilot    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [4.91, 270, 1.9e+03, 5.42e+03, 9.46e+03, 1.17e+04, 1.26e+04, 1.29e+04, 1.3e+04, 1.3e+04, 1.3e+04, 1.3e+04, 1.3e+04]
costs: [4.91, 0.151, 1.6, 2.41, 2.71, 2.08, 1.67, 1.23, 0.96, 1, 0.968, 1.22, 1.5]
weights: [4.91, 0.0182, 0.142, 0.351, 0.573, 0.811, 0.925, 0.976, 0.993, 0.998, 1, 1, 1]
edges: [23, 33.6, 44.2, 54.8, 65.3, 75.9, 86.5, 97.1, 108, 118, 129, 139, 150]

/Simulations/0/Tilings/0/Basins/0/Stages/Production    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [4.76, 256, 1.82e+03, 5.1e+03, 8.81e+03, 1.08e+04, 1.19e+04, 1.21e+04, 1.21e+04, 1.22e+04, 1.22e+04, 1.22e+04, 1.22e+04]
costs: [4.76, 0.152, 1.59, 2.38, 2.7, 2.02, 1.78, 1.17, 0.989, 1.03, 1.05, 1.31, 1.51]
weights: [4.76, 0.0186, 0.141, 0.356, 0.579, 0.812, 0.909, 0.986, 0.995, 0.998, 0.998, 1, 0.999]
edges: [23, 33.6, 44.2, 54.8, 65.3, 75.9, 86.5, 97.1, 108, 118, 129, 139, 150]

/Simulations/0/Tilings/0/Basins/1/Stages/Pilot    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [0.323, 3.81, 9.21, 18.5, 44.2, 111, 343, 1.08e+03, 2.74e+03, 5.65e+03, 7.99e+03, 8.99e+03, 9.16e+03]
costs: [0.323, 0.0343, 0.451, 0.967, 1.76, 2.51, 3.48, 4.28, 4.6, 4.92, 3.96, 2.89, 2.12]
weights: [0.323, 0.0849, 0.413, 0.497, 0.42, 0.397, 0.324, 0.317, 0.396, 0.485, 0.706, 0.889, 0.981]
edges: [150, 139, 129, 118, 108, 97.1, 86.5, 75.9, 65.3, 54.8, 44.2, 33.6, 23]

/Simulations/0/Tilings/0/Basins/1/Stages/Production    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [0.308, 3.6, 8.71, 17.4, 40.5, 102, 322, 999, 2.54e+03, 5.25e+03, 7.38e+03, 8.42e+03, 8.58e+03]
costs: [0.308, 0.0341, 0.445, 0.956, 1.76, 2.51, 3.53, 4.28, 4.66, 4.94, 3.94, 3, 2.08]
weights: [0.308, 0.0856, 0.413, 0.502, 0.429, 0.396, 0.318, 0.323, 0.393, 0.484, 0.712, 0.876, 0.982]
edges: [150, 139, 129, 118, 108, 97.1, 86.5, 75.9, 65.3, 54.8, 44.2, 33.6, 23]
```

The reason why the short simulation we just ran produced so much data is that we had turned on extra output via the `ffluxPilotOutput` option[†].

For each record it finds, `dumpSFile` will first print out a line containing the record's metadata. This metadata has the following format:

```
record.name    record.dataType    record.dataSize
```

The data section of the record will then be printed out starting on the next line after the metadata.

[†] If the simulation had been run using the default output options, the output would contain only 2 records, the summaries from each of the production stages.

`dumpSFile` has many options that can help to tame the torrent of data. Here's a couple of the more useful ones:

`-i/--include <pattern0> <pattern1> ...`
: When this option is set, `dumpSFile` will attempt to match each regular expression `<patterni>` to each record. Records will only be printed out if every pattern matches either the record's `name` or its `dataType`. Otherwise, `dumpSFile` will skip the record. Patterns passed to `-i` are case insensitive.

  For example, the following `dumpSFile` command, which passes the terms "summary", "production", and "basins/1" to the `-i` flag:

```
user@host:srg_mfpt$ dumpSFile data/srg_-_out.sfile -i summary production basins/1
```

  The "basins/1" and "production" terms match to `record.name`, and the "summary" term matches to `record.dataType`. This limits output to a single stage summary record, the one from the production stage that was initialized in basin `ID 0`:

```
data_worked/srg_-_out.sfile

/Simulations/0/Tilings/0/Basins/1/Stages/Production    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
first_passage_times: [0.308, 3.6, 8.71, 17.4, 40.5, 102, 322, 999, 2.54e+03, 5.25e+03, 7.38e+03, 8.42e+03, 8.58e+03]
costs: [0.308, 0.0341, 0.445, 0.956, 1.76, 2.51, 3.53, 4.28, 4.66, 4.94, 3.94, 3, 2.08]
weights: [0.308, 0.0856, 0.413, 0.502, 0.429, 0.396, 0.318, 0.323, 0.393, 0.484, 0.712, 0.876, 0.982]
edges: [150, 139, 129, 118, 108, 97.1, 86.5, 75.9, 65.3, 54.8, 44.2, 33.6, 23]
```

`-e/--exclude <pattern0> <pattern1> ...`
: The opposite of `-i/--include`. Records will be excluded if every `<patterni>` matches either its `name` or its `dataType`.

`-l/--list-only`
: When this flag is passed to `dumpSFile`, only the metadata line of each record is printed out. This can be very useful for getting a sense of the complete contents of an `SFile`. It can also be combined with the `--exclude/--include` options.

  For example, the following command:

```
user@host:srg_mfpt$ dumpSFile data/srg_-_out.sfile -l -i pilot
```

  will output only the metadata from the pilot stage records:

```
data_worked/srg_-_out.sfile

/Simulations/0/Tilings/0/Basins/0/Stages/Pilot    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
/Simulations/0/Tilings/0/Basins/1/Stages/Pilot    protobuf:lm.fflux.io.FFluxStageOutputSummary    426
```

Run `dumpSFile --help` to get a detailed description of all of the options that `dumpSFile` supports.

3.6 Analyzing FFPilot output data using the robertslab.sfile Python package

3.6.1 Fetching and analyzing MFPT values in FFPilot simulation output

Although `dumpSFile` offers an easy way to browse the contents of FFPilot simulation output, it is less than ideal when used as a tool to perform detailed analysis. When performing any non-trivial calculations on FFPilot simulation output, the current recommended best practice is to use the `robertslab.sfile` Python package.

In this section I will go over how to retrieve and analyze data from an FFPilot output file using Python code. Specifically, I will walk you through the steps required to fetch the phase weights and MFPTs from the production stage summary records, and show you how to recalculate the MFPTs from the phase weights.

As can seen in the output of `dumpSFile` in the previous section, stage summary records each contain 4 arrays:

`edges`
: The positions of the edges in the tiling used to run the stage that produced this output.

`costs`
: The mean computational cost per-trajectory launched during each phase in this stage.

`first_passage_times`
: The expected value of the simulation time required to reach each tiling edge from the starting basin.

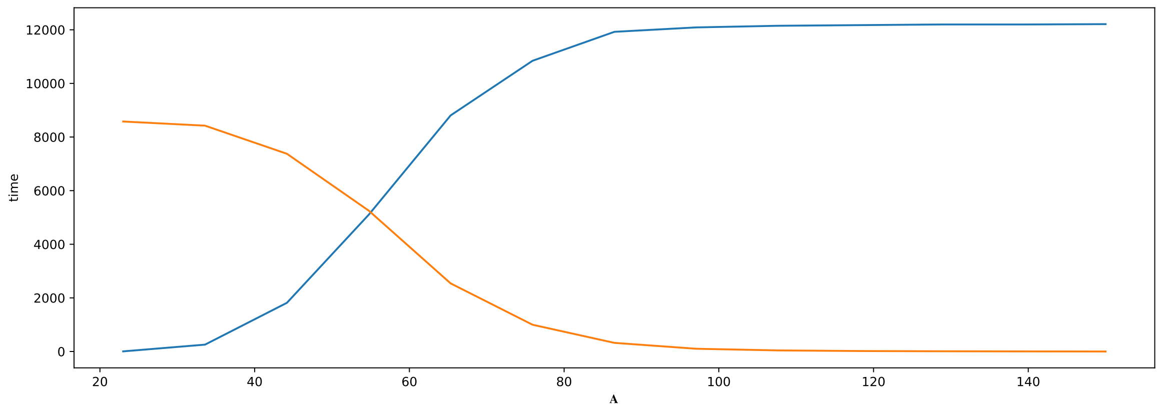
`weights`
: The phase weights.

For this exercise we want the stage summary records from the two production stages. These records can be fetched from `srg_out.sfile` with the following Python script:

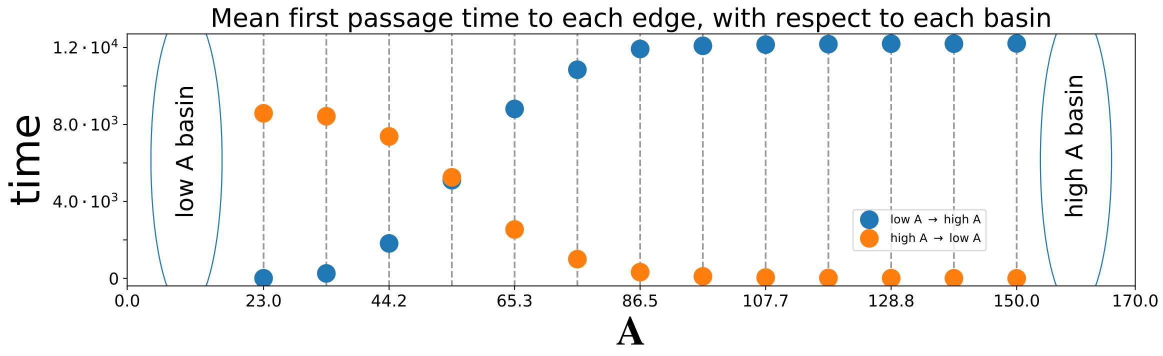
```python
from robertslab.sfile import SFileProto

with SFileProto.open('data/srg_-_out.sfile') as f:
    summaries = list(f.msgs(include=('production', 'summary')))
```

A simple plot of the MFPT data can be made using the `matplotlib` Python plotting package:

```python
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12,4))
ax = fig.add_axes([0, 0, 1, 1])

for summary in summaries:
    ax.plot(summary.edges, summary.first_passage_times)

ax.set_xlabel(r'$⍰$')
ax.set_ylabel('time')
pass
```



The plot can be made much nicer with a bit of configuration:



the details of which are outside of the scope of this tutorial. However, for the curious, the complete code used to produce the above figure can be found in `notebooks/src/plotTiling.py`, and a concrete example of its use is in the `notebooks/srg_mfpt.ipynb` notebook.

3.6.2 Calculating the MFPT to each tiling edge from the phase weights

FFPilot simulation calculates the MFPT to the ith edge of the tiling via a combination of the phase weights,

$$\frac{w_0}{\prod_{i=1}^{j} w_i}$$

(as per Eq 2.2). The value of this formula at all of the edges can be quickly calculated using a cumulative product. Given a sequence in which the zeroth term is w_0 and remaining terms are the inverse values each $w_{i>0}$, the ith element of the cumulative product of this sequence will be the expected value of the time required to reach the ith edge:

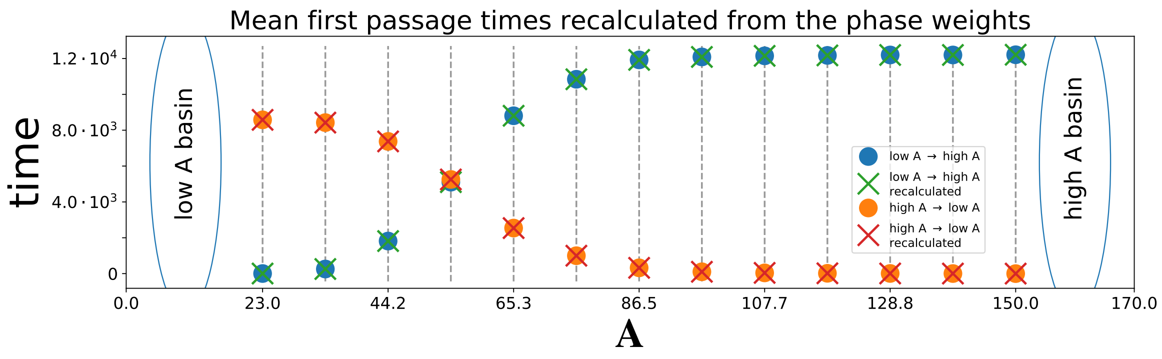
$$\text{cumprod}(\{w_0, \frac{1}{w_1}, \frac{1}{w_2}, \cdots, \frac{1}{w_N}\}) = \{w_0, \frac{w_0}{w_1}, \frac{w_0}{w_1 w_2}, \cdots, \frac{w_0}{\prod_{i=1}^{N} w_i}\}$$

The `sfile` package, in conjunction with `numpy`, can be used to easily apply this formula:

```
import numpy as np
from robertslab.sfile import SFileProto
```

```python
with SFileProto.open('data/srg_-_out.sfile') as f:
    summaries = list(f.msgs(include=('production', 'summary')))

mfptRecalcs = []
for summary in summaries:
    # get a copy of the phase weights
    mfptRecalc = np.array(summary.weights)

    # invert all of the weights aside from the zeroth
    mfptRecalc[1:] = 1/mfptRecalc[1:]

    # now take the cumulative product accross all of the phase weights
    mfptRecalc[...] = np.cumprod(mfptRecalc)

    mfptRecalcs.append(mfptRecalc)
```

If you then plot the recalculated MFPT values against the MFPT values that were originally present in the output file you get:



demonstrating that they are indeed identical.

Chapter 4

Calculating the probability landscape of the self regulating gene (SRG) with FFPilot simulation

4.1 Overview

Now we move on to the calculation of a system's probability landscape. The landscape is a map which assigns a probability to every occupied state in a system's state space. Producing a landscape from FFPilot output required a much more in-depth post-simulation analysis is needed to get MFPT values. However, the simulation itself is very similar.

Just like in chapter 2, we are going to run an FFPilot simulation of the SRG model. In fact, the setup of the simulation input file will be almost identical to that used in the previous chapter. The only difference comes in how the simulation output options are set. A large quantity of data is required to calculate the probability landscape of any given system. When attempting to calculate a highly accurate landscape, the size of the required simulation output files can easily climb into the terrabytes. Thus, landscape output is turned off by default.

Three different types of FFPilot simulation output are required in order to calculate a landscape. Here are the parts of each that are required for the landscape calculation (see Sec 2.2 for full explanations):

SpeciesTimeSeries (turn on with `WriteInterval` option)
  `counts` (samples of the species counts) `first_passage_times` (MFPT to each tiling edge)

FFluxStageOutputRaw (turn on with `FFluxStageOutputRaw` option)
  `first_trajectory_ids` (the id of the first trajectory in each phase)
  `final_trajectory_ids` (the id of the final trajectory in each phase)

FFluxStageOutputSummary (on by default)
  `weights` (the phase weights)

In brief, binning together the species count samples gives a biased landscape, and the other factors can be used to reweight it into the unbiased landscape.

4.2 Set up input for FFPilot landscape simulation of SRG

In order to create the needed simulation input file, first follow the procedure given in Secs 3.2.1-3.2.3, but do not follow the instructions for setting the options given in Sec 3.2.5. Instead, we will set the options needed to get all of the landscape output.

4.2.1 Enable landscape output via the FFPilot options

The following Python script sets all of the options needed for the landscape simulation:

```python
import h5py

with h5py.File('data/srg.lm') as f:
    # NB: convert any numerical types to strings before setting option values

    # the trajectory state output writing interval (in terms of the
    # internal simulation time).
    f['Parameters'].attrs['writeInterval'] = str(1.0)

    # we'll need some information from the StageOutputRaw record in order
    # to calculate the landscape
    f['Parameters'].attrs['ffluxStageOutputRaw'] = 'True'

    # set an error goal, which controls the overall accuracy of
    # the simulation. Defaults to .05
    f['Parameters'].attrs['errorGoal'] = str(.10)
```

`writeInterval`
: If set, the species counts of every trajectory will be sampled and written out at the specified interval. For example, if `writeInterval` is set to `4.0`, samples are taken when the internal time of a trajectory reaches `4.0`, `8.0`, `12.0`, etc.

`ffluxStageOutputRaw`
: `bool`, defaults to `False`. When this flag is set to `True`, it turns on output of the ”raw” stage record. By default, each stage only saves a ”summary” record to the output. The raw record contains a bunch of detailed, lower-level data that is used by FFPilot during a running simulation. The raw record is needed for certain post-simulation analyses, such as the calculation of the probability landscape.

As in Sec 3.2.5, `errorGoal` is set slightly higher than the default value (`.05`) in order to produce a faster (though less accurate) simulation for demonstration purposes.

As a rule of thumb, a good value of `writeInterval` is 1 divided smallest rate constant in the model. For the SRG model we are using, the smallest rate constant is the one controlling the decay reaction, and it has a value of 1.

4.3 Running the simulation

The simulation is run in an identical fashion to the one we performed in Sec 3.3:

```
user@host:srg_landscape$ lmes -fflux -f data/srg.lm
```

The only noticeable difference should be that the simulation output file is significantly larger than it was before. This is caused by all of the system state samples that get written to the output due to the `writeinterval` option being set.

4.4 Calculating probability landscapes of SRG from FFPilot output

4.4.1 Ingesting the phase landscapes from the simulation output

We begin our analysis of the output of the SRG landscape simulation by ingesting the state samples (and other needed data) from the raw simulation output itself. Most importantly, we will bin all of the state samples that were collected during the same phase together into the phase histograms (see Sec 4.1 for details). The following Python script uses the `robertslab.SFile` package to accomplish this:

```python
"""Ingest the raw data you'll need to calculate the landscape from the
simulation output SFile
"""
import re
from robertslab.sfile import SFileProto

basinRe = re.compile('Basins/(\d*)')
stageRe = re.compile('Stages/(\w*)')
phaseRe = re.compile('Phases/(\d*)')

basinEdges = {}
basinPhaseHists = {}
basinMFPTs = {}
basinPhaseWeights = {}
basinTrajectoryCounts = {}

# open the simulation output and show a progress bar
with SFileProto.open('data/srg_-_out.sfile', progress=True) as f:
    phaseHists = {}

    # iterate through each record in the output
    for rec in f.records():
        # determine the basin and stage during which the record was written
        basin = int(basinRe.search(rec.name).group(1))
        stage = stageRe.search(rec.name).group(1)

        if stage != 'Production':
            # if this isn't a record from a production stage, skip it
            continue

        if rec.dataType.find('SpeciesTimeSeries') > -1:
            # determine the phase during which the record was written
            phase = int(phaseRe.search(rec.name).group(1))

            # fetch the phaseHists for this basin
            basinPhaseHists[basin] = phaseHists = basinPhaseHists.get(basin, dict())

            # add to the count of the appropriate phaseHist
            phaseHists[phase] = phaseHist = phaseHists.get(phase, dict())

            # increment the values of the phaseHist based on the species count
            # samples in the record
            for count in (tuple(row) for
                          row in rec.msg(unpackNDArray=True)[1]['counts']):
                phaseHist[count] = phaseHist.get(count, 0) + 1

        elif rec.dataType.find('StageOutputSummary') > -1:
            # get the first passage times and the phase weights
            # from the StageOutputSummary record
            summaryMsg = rec.msg()
            basinEdges[basin] = np.array(summaryMsg.edges)
            basinMFPTs[basin] = np.array(summaryMsg.first_passage_times)
            basinPhaseWeights[basin] = np.array(summaryMsg.weights)

        elif rec.dataType.find('StageOutputRaw') > -1:
            # get the trajectory counts from the StageOutputRaw record
            rawMsg = rec.msg()
            basinTrajectoryCounts[basin] = np.array(rawMsg.final_trajectory_ids) -  \
                                           np.array(rawMsg.first_trajectory_ids) + 1
```

In brief, the script opens the output file and iterates through the records. When it finds a record containing needed data, it performs a particular action based on the record's type:

SpeciesTimeSeries
: The script retrieves the appropriate phase landscape histogram (one is created for each combination of basin and phase), unpacks the `counts` array of the record, and then adds it to that histogram.

FFluxStageOutputRaw (turn on with `FFluxStageOutputRaw` option)
: The script calculates the count of trajectories run in each phase then stores the results.

FFluxStageOutputSummary (on by default)
: The script fetches the `weights` and the `first_passage_times` and stores them.

The outcome of this script is the creation of a number of python dictionaries that contain all of the data needed for the subsequent analyses. Each of these dictionaries contains exactly two entries, one for each basin/production stage in the simulation we just ran. The nature of these entries is as follow:

`basinPhaseHists[i]`
: The phase hists from the production stage initialized at basin i.

`basinMFPTs[i]`
: The MFPT to each tiling edge of stage i.

`basinPhaseWeights[i]`
: The phase weights of stage i.

`basinTrajectoryCounts[i]`
: The total count of trajectories, whether they succeeded or failed, run during each phase of stage i

4.4.2 How histograms are represented in this analysis

The phase landscape histograms built up by the script will each contain the species count observations collected during a single particular phase. They are represented as Python dictionaries with the following format:

KEY	VALUE
$\cdots$	
$(s_{k_0}, s_{k_1}, \cdots, s_{k_S})$	x_k
$(s_{k+1_0}, s_{k+1_1}, \cdots, s_{k+1_S})$	x_{k+1}
$\cdots$	

Where s_{i_j} is the ith observed state (*i.e.* count) of a system's j unique chemical species, and x_i is the number of times the ith state has been observed.

For example, from our SRG landscape simulation, the contents of the histogram created from the samples collected from phase 4 of the simulation stage initialized from basin 1 may look like this:

```
((108,), 22)
((149,), 28)
((109,), 67)
((110,), 72)
((147,), 74)
...
((124,), 514)
((122,), 517)
((121,), 521)
((127,), 522)
((120,), 532)
```

4.4.3 Basic manipulation of histograms

Before moving on to the landscape calculation proper, we'll first define a few basic functions to help manipulate the histograms:

```python
"""Basic histogram manipulation functions.
Assumes that the histograms are dictionaries with keys (x0, x1, ..., xn)
that represent discrete states and values m that represent the number
of times each state was observed
"""
def addHists(h0, h1, weight=1.0):
    """Add two hists together
    """
    hsum = dict(h0)
    for obs,val in h1.items():
        h0[obs] = h0.get(obs, 0) + val*weight

    return hsum

def calcOParamHist(h, oparam):
    """Produce a new hist from `h` in which all of the states have been
    transformed to their corresponding order parameter value.
    Transforms the states using the passed-in `oparam` function
    """
    oparamHist = {}

    for count,val in h.items():
        opval = oparam(count)
        oparamHist[opval] = oparamHist.get(opval, 0) + val

    return oparamHist

def normHist(h, scale=1.0):
    """Normalizes a hist such that sum(hist.values())==1
    """
    histsum = sum(list(h.values()))
    return {obs:scale*val/histsum for obs,val in h.items()}

def sparseToDense1D(h):
    """Takes a 1D hist in the dictionary (ie sparse) representation
    and produces an array (ie dense) representation of the same hist
    """
    # sort the (observation, val) pairs in the sparse histogram
    hsorted = sorted([(obs[0], val) for obs,val in h.items()])

    # split the observations and values into their own sequences
    edges,dense = [np.array(data) for data in zip(*hsorted)]

    return dense,edges
```

These 4 functions are general histogram operations:

`addHists`
Adds the values from two histograms `h0` and `h1` together into a new histogram. For example:

```
         h0                        h1                   addHists(h0,h1)

    [0, 0, 0] → [150]     [0, 0, 0] → [59]      [0, 0, 0] → [209]
    [0, 0, 1] → [141]     [0, 0, 1] → [148]     [0, 0, 1] → [289]
    [0, 1, 0] → [132]     [0, 1, 0] → [152]     [0, 1, 0] → [284]
    [0, 1, 1] → [86]      [0, 1, 1] → [139]     [0, 1, 1] → [225]
    [1, 0, 0] → [90]      [1, 0, 0] → [143]     [1, 0, 0] → [233]
    [1, 0, 1] → [136]     [1, 0, 1] → [154]     [1, 0, 1] → [290]
    [1, 1, 0] → [130]     [1, 1, 0] → [123]     [1, 1, 0] → [253]
    [1, 1, 1] → [135]     [1, 1, 1] → [82]      [1, 1, 1] → [217]
```

`calcOParamHist`
: Takes a histogram of state's (*i.e.* species counts) and converts it to a histogram of order parameter values. Uses the passed-in function `oparam` to perform the conversion.

`normHist`
: Normalizes the values of the histogram. The normalized value $\bar{x}_k$ associated with the kth observed state is found with the following simple formula:

$$\overline{x_k} = \frac{x_k}{\sum_k x_k} \tag{4.1}$$

where x_k is the kth unnormalized observation value. The values of the normalized histogram are gauranteed to sum to 1.0, meaning that a normalized histogram represents a formal probability distribution.

`sparseToDense1D`
: Given that `h` is a 1D sparse histogram (for example, any histogram returned by `calcOParamHist` when a 1D order parameter is passed in), this function will convert it to a dense histogram. This dense histogram takes the form of two 1D arrays, one for the observations and one for the values.

We will also define a Python version of the 1D order parameter $\mathbf{A}$:

```python
"""Order parameter function(s). These convert a species count to an
order parameter value.
"""
def oparamAlpha(count):
    alpha = sum(count[id]*coeff for id,coeff in zip((0,), (1,)))

    # return as a tuple
    return alpha,

oparam1D = oparamAlpha
```

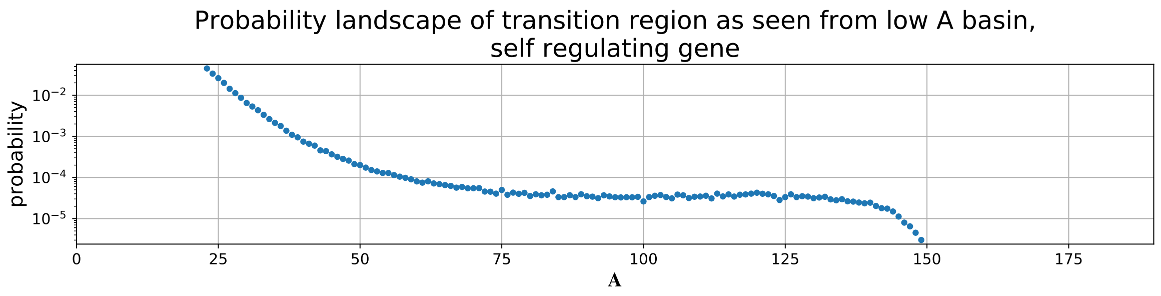
`oparamAlpha`
: Takes a species count (as a tuple) for input and returns the value of the order parameter (also as a tuple). For SRG, it may seem trivial and unnecessary to store species counts/order parameter values as tuples, since they're always single-valued, or to transform them using a function, since the species count and the order parameter always have the same value.

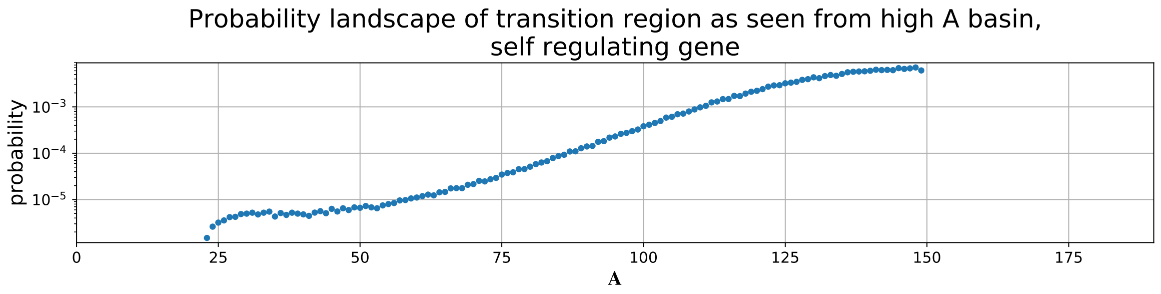
However, writing the code this way makes most of it reusable when we start dealing with systems with higher-dimensional state spaces and order parameters in the next chapter.

4.4.4 Calculate the stage histograms

The first step to calculating the landscape is to take the phase hists from all of the phases that were run during each of the stages and combine them into stage hists. The landscape weights $\mathbb{l}_i$ needed to combine the phases hists within a stage can be calculated from Eq 2.3 (see Sec 2.2.2 for details). The following Python script will calculate the $\mathbb{l}_i$ values, weight the phase hists, and then combine them into two stage hists:

```python
"""create the stage hists by reweighting and combining the phase hists.
One stage hist will be created per basin set in the simulation's input
"""
basinIDs = sorted(basinPhaseHists.keys())

basinStageHists = {}
for bid in basinIDs:
    phaseHists = basinPhaseHists[bid]
    phaseWeights = basinPhaseWeights[bid]
    trajectoryCounts = basinTrajectoryCounts[bid]

    # calculate the weight the landscape sampled during each phase will have
    landscapePhaseWeights = np.ones(phaseWeights.size, dtype=float)
    landscapePhaseWeights[2:] = phaseWeights[1:-1]
    landscapePhaseWeights = landscapePhaseWeights.cumprod()
    landscapePhaseWeights[1:] /= trajectoryCounts[1:]

    # calculate the stage hist by combining all of the phase hists
    stageHist = {}
    for phaseID,weight in enumerate(landscapePhaseWeights[1:].flat):
        addHists(stageHist, phaseHists[phaseID+1], weight=weight)

    basinStageHists[bid] = stageHist
```



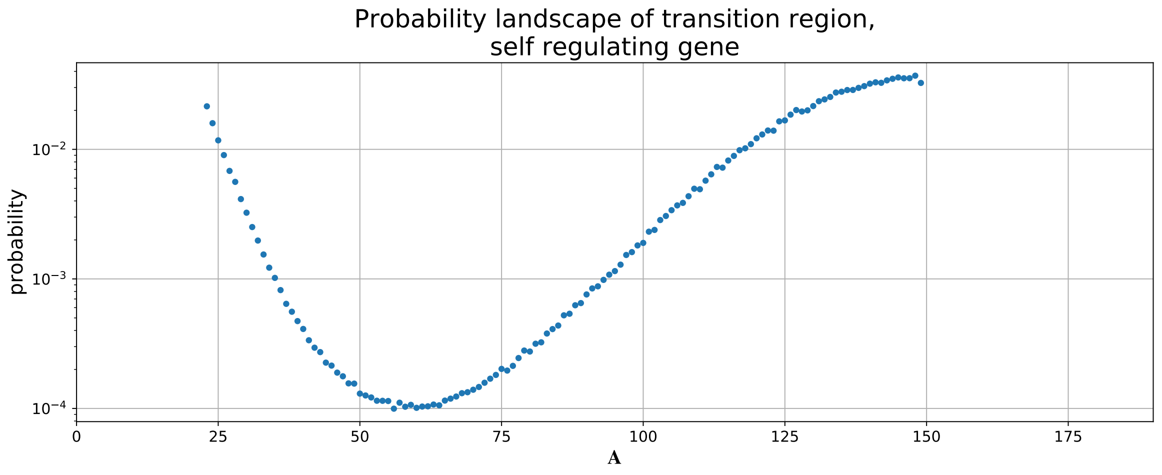


4.4.5 Combine the stage landscapes into the transition landscape

Now we find the transition region landscape by taking the sum of the two stage landscapes weighted by the stage landscape factors $\mathbb{S}_i$. The following script find the $\mathbb{S}_i$ values and performs the combination of hists:

```
"""calculate and combine the stage hists from the two basins in order
to get an unbiased hist of the transition region (ie the region
covered by the tiling used in the simulation)
"""
basinIDs = sorted(basinPhaseHists.keys())

# calculate the weight that the stage hist from each basin
# should have in the combined hist
phaseZeroWeights = {i:basinPhaseWeights[i][0] for i in basinIDs}
mfptOveralls = {i:basinMFPTs[i][-1] for i in basinIDs}

# combine the stage hists into the transition region hist
transitionHist = {}
for bid in basinIDs:
    # calculate the weight each stage hist should have in the combined hist
    stageWeight = mfptOveralls[bid]/((mfptOveralls[0]
                                      + mfptOveralls[1])*phaseZeroWeights[bid])

    # weight and add the stage hists
    addHists(transitionHist, basinStageHists[bid], weight=stageWeight)

# normalize the transition hist
transitionHist = normHist(transitionHist)
```

We now have an unbiased landscape of the SRG, though for now it only spans the transition region in between the basins:



4.4.6 Fit together the complete landscape

The final step of assembling the complete landscape is also the most complicated. It proceeds in roughly 3 steps:

1. Sort all of the phase zero samples into two histograms
One with the samples to the left of the transition minimum, and one with all of the sample to the right. The following script will accomplish this:

```
# find an interval (in terms of the 1D order parameter) containing
# the transition state minimum
width = 20
transitionMinOpval = sorted(transitionHist.items(),
                            key=lambda x: x[1])[0][0][0]

transitionMinLeft = transitionMinOpval - width/2
transitionMinRight = transitionMinOpval + width/2

# Sort the phase zero data into a left and right hist (with respect to
# the transition region minimum)
phaseZeroHists = [basinPhaseHists[i][0] for i in basinIDs]
phaseZeroLeftHist = {}
for count,val in phaseZeroHists[0].items():
    if oparam1D(count)[0] < transitionMinLeft:
        phaseZeroLeftHist[count] = val
phaseZeroRightHist = {}
for count,val in phaseZeroHists[1].items():
    if oparam1D(count)[0] >= transitionMinRight:
        phaseZeroRightHist[count] = val
phaseZeroSideHists = [normHist(h) for h in (phaseZeroLeftHist, phaseZeroRightHist)]
```

2. Fit the left and right phase zero histograms onto the transition region landscape
Independently, we find the weights that will smoothly fit both the left and right phase zero histograms

to the transition landscape. We do this using a straightforward minimization approach, with a least-squares minimizer and a Jenson-Shannon divergence*. The following script will perform this fitting and report the weights:

```python
import scipy.optimize as opt

# fit the left and right hists seperately to the transition hist
transitionHist1D = calcOParamHist(transitionHist, oparam1D)
sideLandscapeWeights = []
for pzHist in phaseZeroSideHists:
    pzHist1D = calcOParamHist(pzHist, oparam1D)

    # define a Jensen-Shanon divergence (symmetrized KL divergence) function
    # for the minimizer
    def _jsdiv(x):
        div = 0

        for opval in (opval for opval in transitionHist1D.keys()
                      if opval in pzHist1D):
            transval,pzval = transitionHist1D[opval], x*pzHist1D[opval]

            # iterate only over states in which both hists contain
            # at least some density
            if transval==0 or pzval==0:
                continue

            meanval = .5*(transval + pzval)
            div += .5*(transval*np.log(transval/meanval)
                       + pzval*np.log(pzval/meanval))

        return div

    # start the fitting weight at 2.0
    weightFit0 = 2.0

    # fit by minimizing the difference between the transition hist and the section
    # of the phase zero hist that overlaps with the transition hist
    weightFitOut = opt.minimize(_jsdiv, x0=weightFit0, method='Nelder-Mead')
    sideLandscapeWeights.append(weightFitOut.x[0])

# print some info about the fit of the left and right sides of the landscape
print('The left side of phase zero will have a weight of: %.3f'
      % sideLandscapeWeights[0])
print('The right side of phase zero will have a weight of: %.3f'
      % sideLandscapeWeights[1])
```

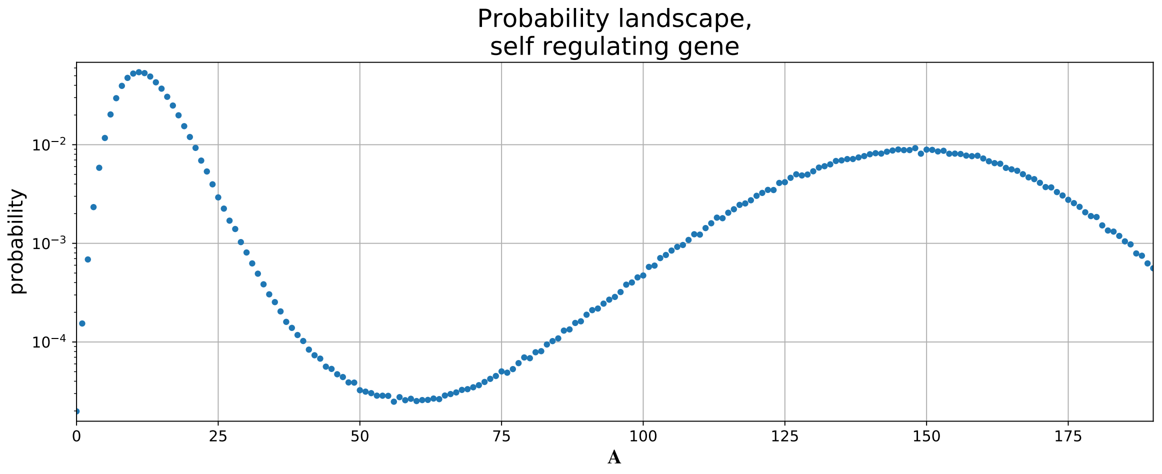
3. Patch together the phase zero histograms and the transition landscape
Finally, we patch the weighted phase zero histograms together with the transition landscape in order to produce the complete landscape. The following script performs this patching:

*a symmetrized form of the Kullback-Leibler divergence

```
# initialize the hist that will cover the entire landscape by
# starting with the transition region hist
landscapeHist = dict(transitionHist)

# get the range of the transition region, in terms of the
# 1D order parameter values
oparamvals = [oparam1D(count) for count in landscapeHist.keys()]
opmin,opmax = min(oparamvals), max(oparamvals)

# add the non-overlapping data from the left and
# right hists to the transition hist
for count,val in phaseZeroSideHists[0].items():
    opval = oparam1D(count)
    if opval < opmin:
        if count in landscapeHist: raise KeyError
        landscapeHist[count] = sideLandscapeWeights[0]*val
for count,val in phaseZeroSideHists[1].items():
    opval = oparam1D(count)
    if opval > opmax:
        if count in landscapeHist: raise KeyError
        landscapeHist[count] = sideLandscapeWeights[1]*val
landscapeHist = normHist(landscapeHist)
```

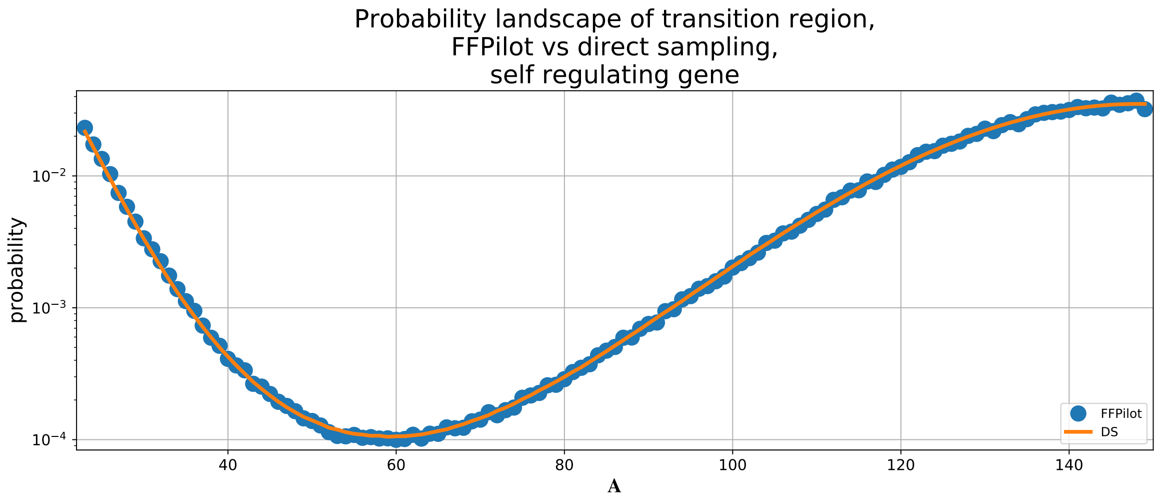
If any bin in the phase zero histograms overlaps with any bin in the transition landscape, we discard the information from the phase zero histograms in favor of that in the transition landscape, on the assumption that the transition landscape will in general be better sampled.

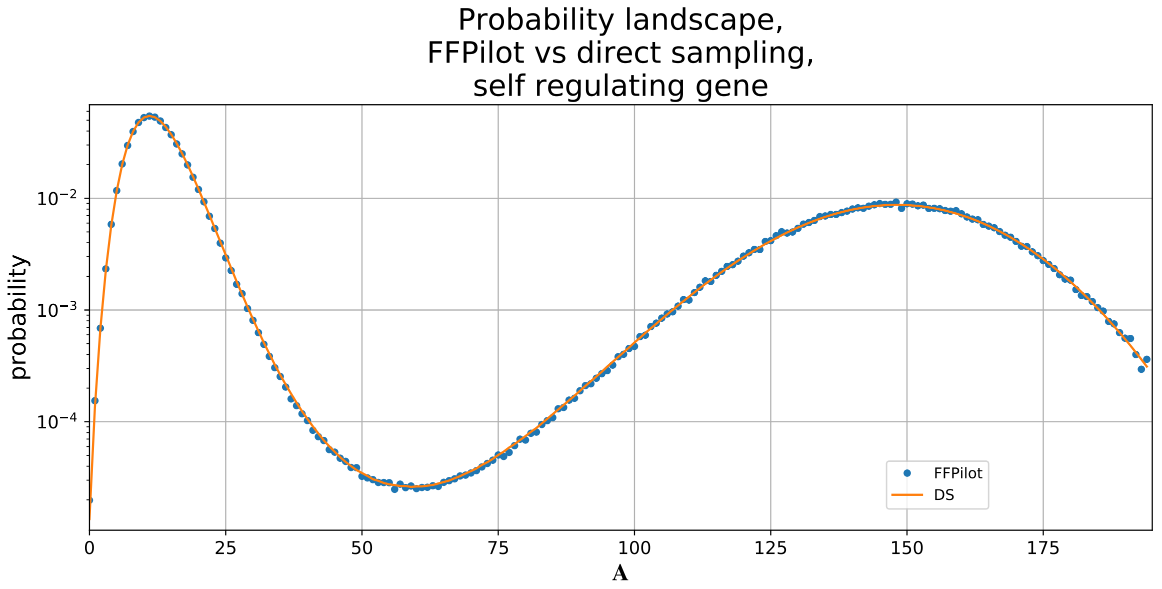
Now that it's all put together, the complete unbiased landscape looks like this:



4.4.7 Comparison of landscapes produced using FFPilot and direct sampling

Shown below are a comparison of landscapes calculated using a (blue dots) 10% error goal FFPilot simulation and (orange line) direct sampling simulation that collected 10^9 state samples.





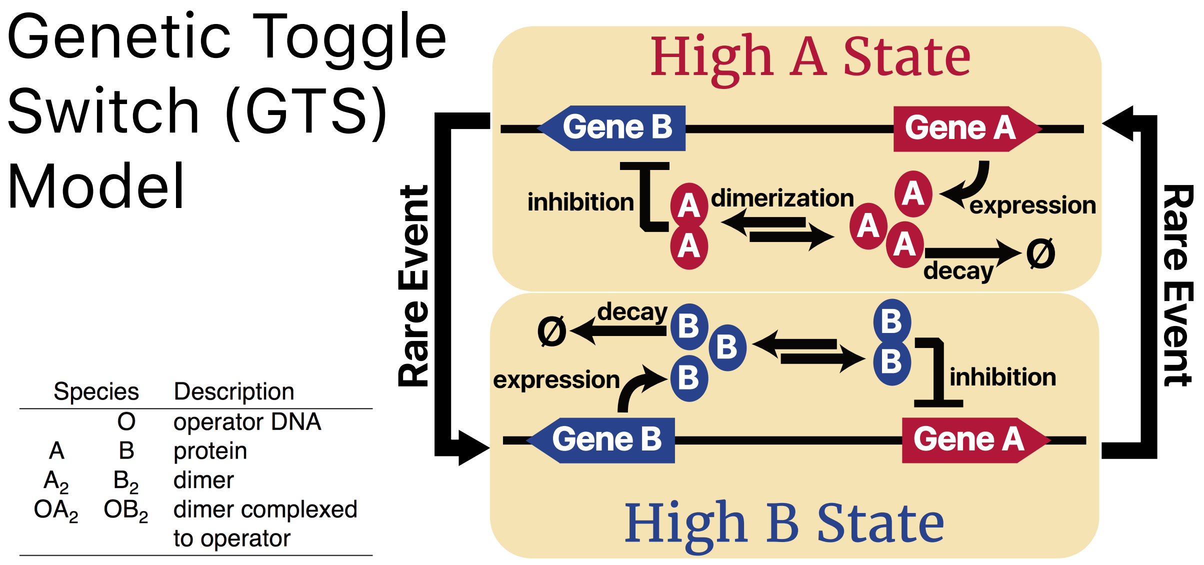
Although the FFPilot landscape is in this case slightly noisier, overall there is a high level of agreement between the FFPilot and direct sampling landscapes. This is made more impressive by the fact that, in terms of real wall-clock time, the FFPilot simulation ran to completion about 140x faster than the direct sampling one (~ 30 seconds vs ~ 70 minutes).

Chapter 5

Using FFPilot to simulate systems with complex, high-dimensional state spaces: genetic toggle switch

5.1 Overview

In this chapter, we'll be working with the genetic toggle switch (GTS). genetic toggle switch (GTS) is to the study of genetic regulatory systems what the hydrogen atom was to the study of quantum mechanics. It is one of the simplest truly biphasic systems that has actually been experimentally instantiated in living cells [3]. GTS models come in many different variants, and we will work with one called the exclusive genetic toggle swtich. It consists of 7 distinct chemical species that participate in 14 separate reactions.



Species		Description
	O	operator DNA
A	B	protein
A_2	B_2	dimer
OA_2	OB_2	dimer complexed to operator

Reactions		Rate Constants		Description
Protein A	*Protein B*	*forward*	*reverse*	
$O \longrightarrow O + A$	$O \longrightarrow O + B$	θ		expression
$A \longrightarrow \varnothing$	$B \longrightarrow \varnothing$	$\theta/4$		decay
$2A \rightleftharpoons A_2$	$2B \rightleftharpoons B_2$	5	5	dimerization
$O + A_2 \rightleftharpoons OA_2$	$O + B_2 \rightleftharpoons OB_2$	5	1	operator binding
$OA_2 \longrightarrow OA_2 + A$	$OB_2 \longrightarrow OB_2 + B$	θ		bound expression

Protein A and protein B are both produced from a single piece of operator DNA. A and B can both dimerize, and each dimer can bind back to the operator DNA. Only one dimer can bind to the DNA at a time. When the DNA is unbound, it will produce both A and B at an equal rate. However, when a dimer of one of the proteins is bound, the DNA will only produce more of that same protein. This means that GTS spends the vast majority of its time in either a high A, low B state (state $\mathcal{A}$), or a high B, low A state (state $\mathcal{B}$).

SRG has 1 unique chemical species, and thus a 1D state space. GTS has 7 unique species and a 7D state space, and so is inherently a much more complex system to simulate and analyze. In particular, complex sources of error emerge in higher dimensional systems that aren't present in 1D systems like SRG. Helpfully, GTS has a feature that makes it easy to examine simulation error: it's perfectly symmetric in terms of proteins A and B. In fact, because the species and reactions containing A are identical to those containing B (aside from, of course, the change in protein), the MFPT of the $\mathcal{A} \to \mathcal{B}$ transition is equal to that of $\mathcal{B} \to \mathcal{A}$, and the probability landscape of GTS is perfectly symmetrical. The deviation from this 2-fold symmetry can thus be used as a simple measure of simulation accuracy.

Over the course of the previous chapters in this tutorial we established the theory and protocols for running and analyzing FFPilot simulations. In this chapter we'll focus on error, and how to think about simulation accuracy in general.

5.2 FFPilot simulation input for GTS

The protocol for setting up the input for an FFPilot simulation of GTS is very similar that used in the earlier chapters for setting up simulations of SRG. Essentially, certain array values, such as `SpeciesCoefficients` and `Basins`, which were 1D for SRG are 7D for GTS. Although this may sound complex, in practice it is not much more complex than setting up SRG.

Once again, we're going to use the `lm_sbml_import` to convert a `.sbml` file to an `.lm` file, and then use the `h5py` Python package to add the needed order parameter and tiling.

First, we run the `lm_sbml_import` utility on `genetic_toggle_switch.sbml` in order to produce `gts.lm`. Open a terminal, then `cd` to `notebooks/gts` and then execute:

```
user@host:gts$ mkdir -p data
user@host:gts$ lm_sbml_import data/gts.lm ../genetic_toggle_switch.sbml
```

Next we set the order parameter, the tiling, and the FFPilot simulation options needed for landscape output using the following Python scripts:

order parameter

```python
import h5py

with h5py.File('data/gts.lm') as f:
    # remove any existing OrderParameters group
    if 'OrderParameters' in f.keys():
        del f['OrderParameters']

    # create the OrderParameters group
    oparams = f.create_group('OrderParameters')

    # add the subgroup for order parameter 0
    oparam0 = oparams.create_group('0')

    # set this order parameter's ID to 0, and its Type to 0
    oparam0.attrs['ID'] = 0
    oparam0.attrs['Type'] = 0

    # add the SpeciesIDs. This array should always be of type `int`
    speciesIDs = np.array([0, 1, 2, 3, 4, 5], dtype=int)
    oparam0.create_dataset('SpeciesIDs', data=speciesIDs)

    # add the SpeciesCoefficients. This array should always be of type `float`
    speciesCoefficients = np.array([-1, -2, -2, 1, 2, 2], dtype=float)
    oparam0.create_dataset('SpeciesCoefficients', data=speciesCoefficients)
```

tiling

```
import h5py

with h5py.File('data/gts.lm') as f:
    # remove any existing Tilings group
    if 'Tilings' in f.keys():
        del f['Tilings']

    # create the Tilings group
    tilings = f.create_group('Tilings')

    # add the subgroup for tiling 0
    tiling0 = tilings.create_group('0')

    # set this tiling's ID to 0, its OrderParameterID to 0, and its Type to 0
    tiling0.attrs['ID'] = 0
    tiling0.attrs['OrderParameterID'] = 0
    tiling0.attrs['Type'] = 0

    # add the Basins. There are 7 chemical species in this model,
    # and we are going to set 2 initial basins, so Basins will be a 2 x 7 array
    basins = np.zeros((2,7), dtype=int)
    basins[0,:] = [4, 16, 1, 0, 0, 0, 0]
    basins[1,:] = [0, 0, 0, 4, 16, 1, 0]
    tiling0.create_dataset('Basins', data=basins)

    # add the Edges
    edges = np.linspace(-27.0, 27.0, num=13)
    tiling0.create_dataset('Edges', data=edges)
```

FFPilot simulation options

```python
import h5py

with h5py.File('data/gts.lm') as f:
    # NB: convert any numerical types to strings before setting option values

    # set a higher than default error goal (.05) for a quicker simulation
    f['Parameters'].attrs['errorGoal'] = str(.10)

    # turn on output of sage raw records
    f['Parameters'].attrs['ffluxStageOutputRaw'] = str(True)

    # 1 over the decay rate (.25) of the genetic toggle switch
    f['Parameters'].attrs['writeInterval'] = str(4.0)
```

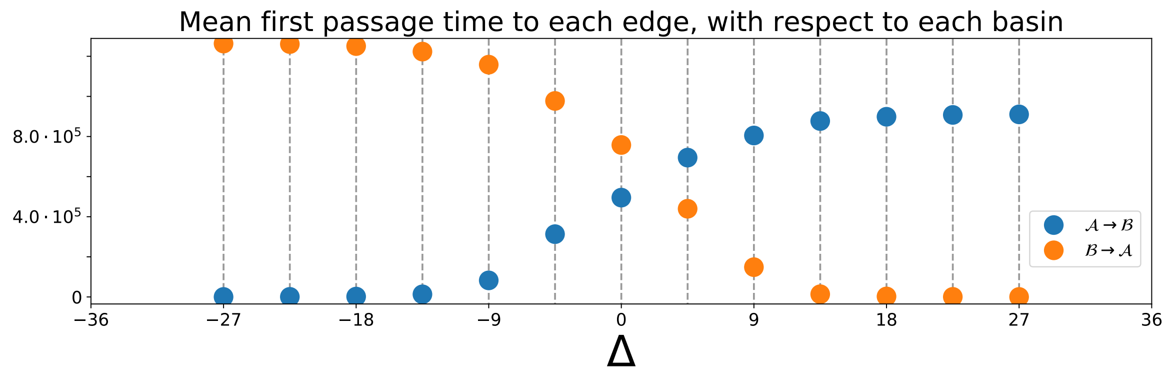
5.3 Running the simulation

No surprises here, as the GTS simulation can be executed with the same basic command that was used to run the SRG simulations:

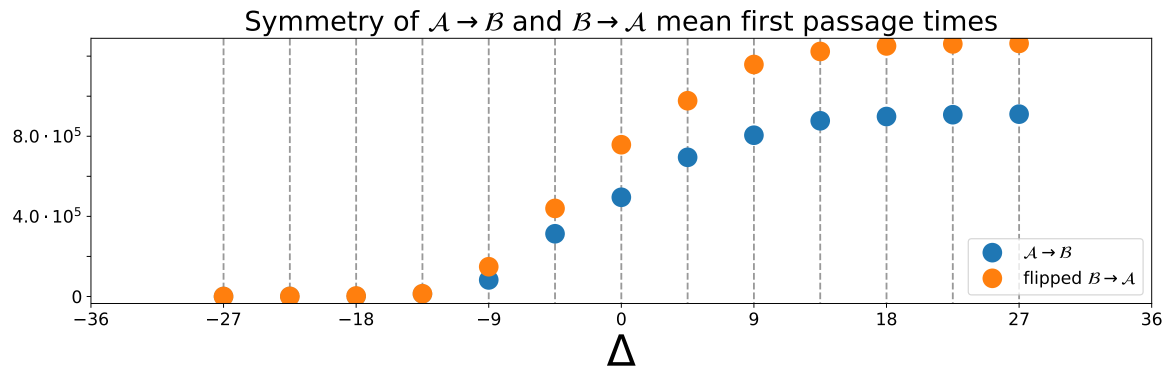
```
user@host:gts$ lmes -fflux -f data/gts.lm
```

5.4 MFPT to each edge of GTS tiling

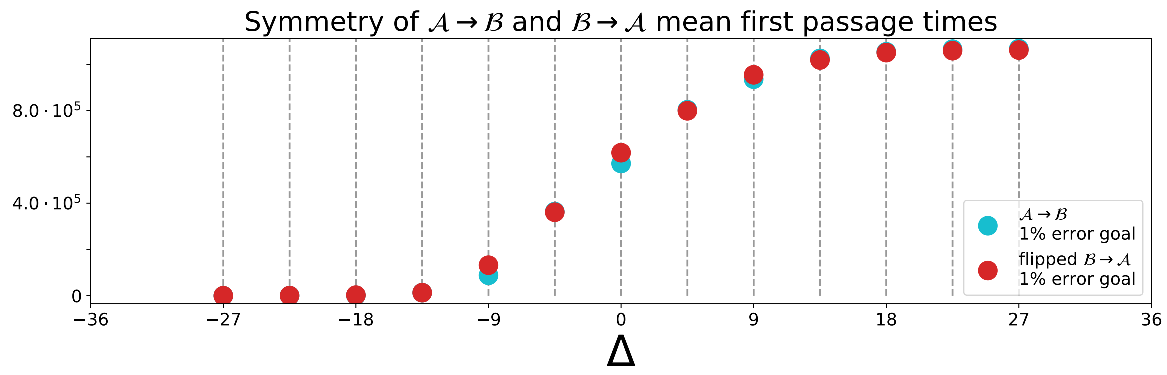
As with our SRG simulations, the MFPT values of GTS can be fetched directly from the simulation stage summary records. Follow the instructions given in Sec 3.6.1 in order to retrieve the MFPT results from the GTS simulation we just ran. Plotting those values should yield something along the lines of:



There are two sets of GTS MFPT values, those collected when switching $\mathcal{A} \to \mathcal{B}$ and when switching $\mathcal{B} \to \mathcal{A}$. Regardless of what kind of simulation was used to estimate them, they should be perfectly symmetrical. This means that if one set is flipped with respect to its edges, the two sets would then be equal. However, complete equality is unlikely to be observed in the results from a 10% error goal simulation. Instead, it is far more likely that some deviation between the two sets of MFPTs will occur:



When GTS is simulated using a 1% error goal, much less deviation occurs between the results from the $\mathcal{A} \to \mathcal{B}$ and $\mathcal{B} \to \mathcal{A}$ processes:



As can be seen above, even though some deviations do begin to develop in phases immediately before or around the transition midpoint (*i.e.* $\Delta = 0$), the simulation recovers from those deviations, and they do not persist into the overall basin-to-basin MFPT estimates. This can be though of as a signature of the technique which FFPilot uses to optimize computational efficiency. The optimizing equation that drives FFPilot tends to bias simulations against running too many trajectories during the most expensive phases (for GTS, the most expensive phases are indeed those in the vicinity of the transition midpoint), and makes up for any errors this might introduce by causing more simulations to be run during the least expensive phases.

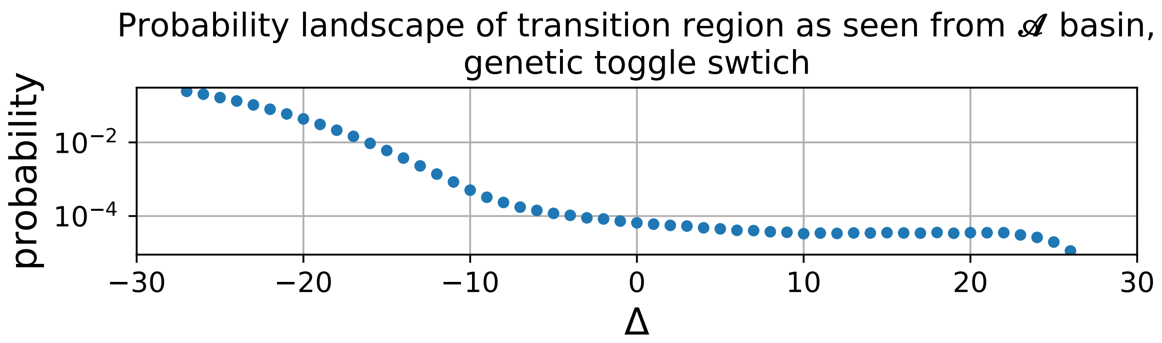
5.5 Landscape of the transition region

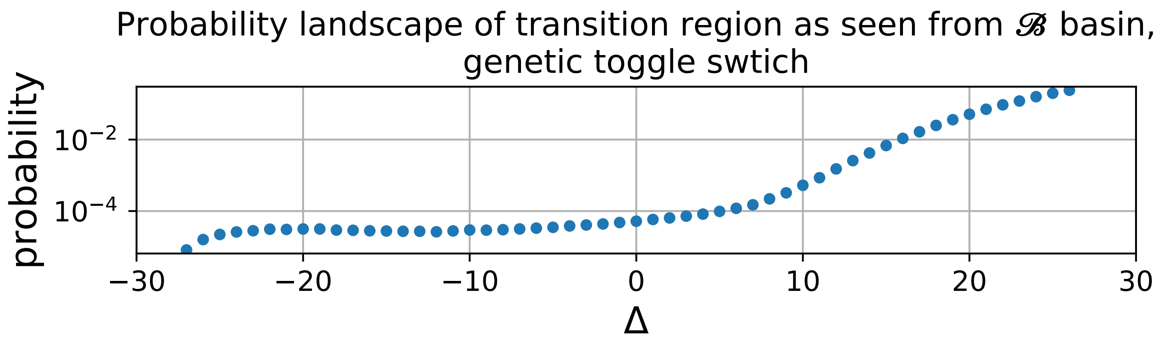
The code we introduced last chapter in Sec 4.4 for calculating the landscape of SRG is flexible enough to be reused almost in its entirety when calculating the landscape of the very different GTS. The only major differences are that we will use the GTS specific 1D order parameter □, and also define a 2D order parameter {**A**, **B**} to do some fancy plotting with:

```python
"""Order parameter functions. These convert a species count to an
order parameter value.
"""
def oparamDelta(count):
    delta = sum(count[id]*coeff for id,coeff in zip((0,1,2,3,4,5), (-1,-2,-2,1,2,2)))

    # return as a tuple
    return delta,

def oparamAB(count):
    A = sum(count[id]*coeff for id,coeff in zip((0,1,2), (1,2,2)))
    B = sum(count[id]*coeff for id,coeff in zip((3,4,5), (1,2,2)))

    return A,B

oparam1D = oparamDelta
oparam2D = oparamAB
```

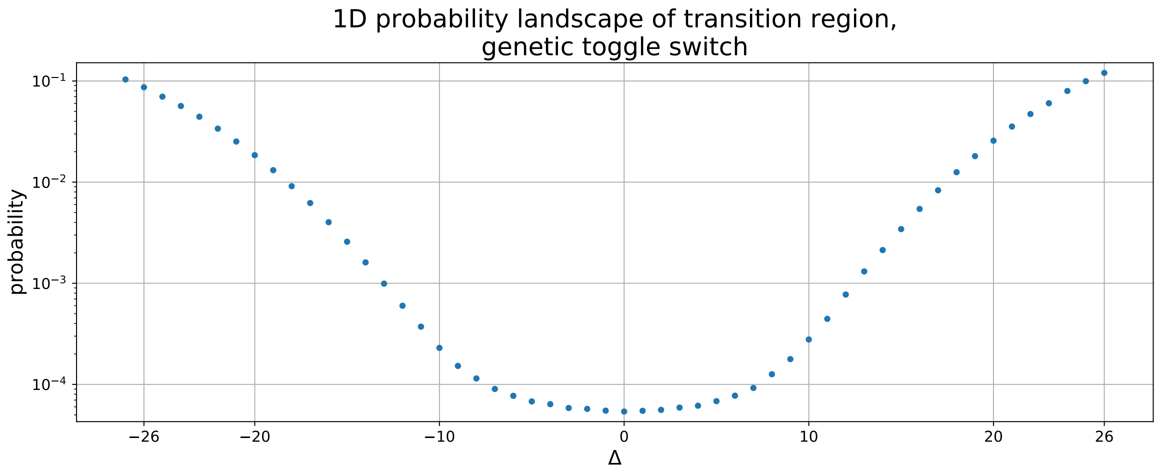
5.5.1 Calculating the transition region landscape

The code from Sec 4.4.4 can be used to produce the GTS stage histograms:

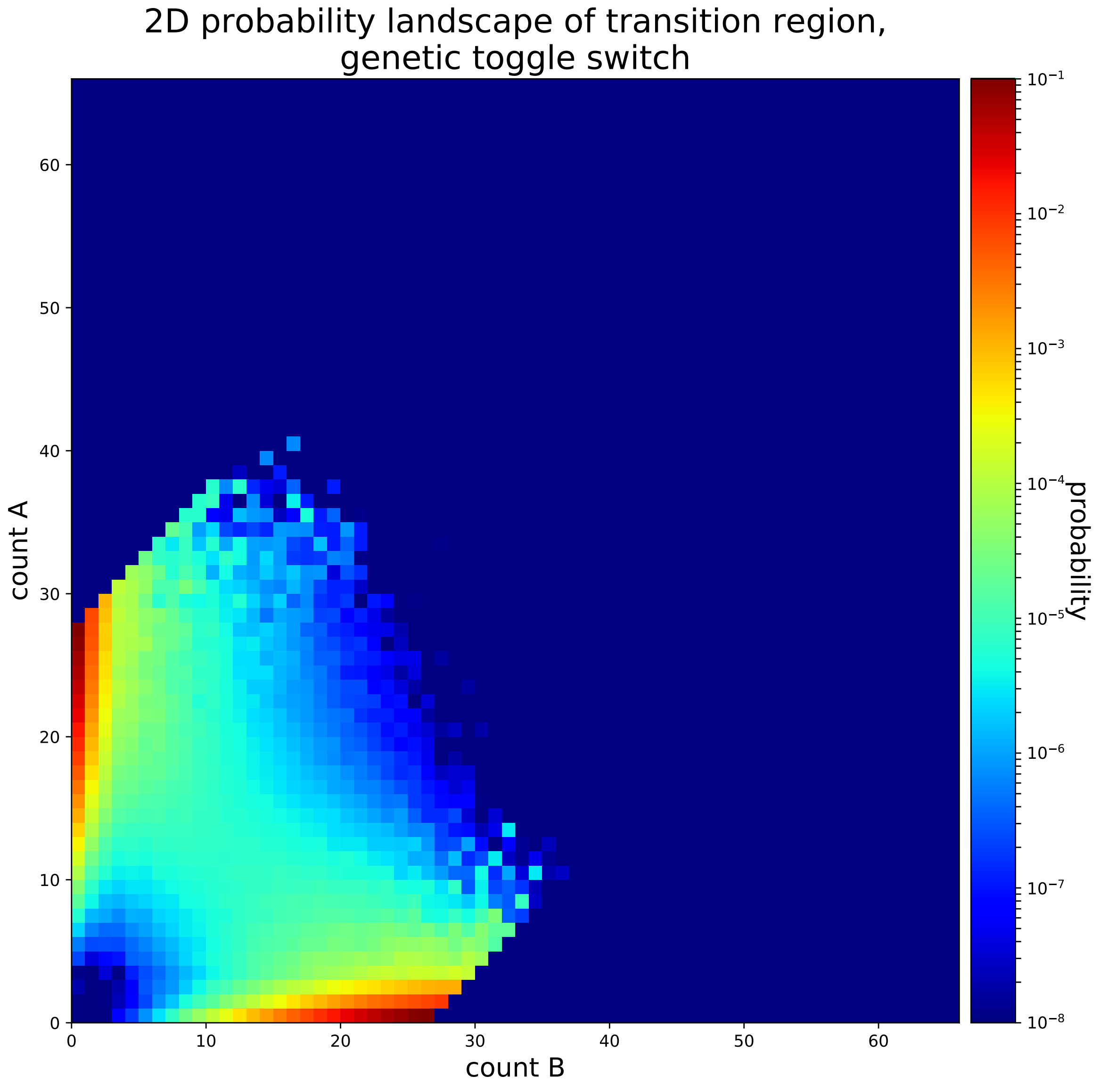




The GTS stage histograms can then be combined into a single transition region landscape using the code from Sec 4.4.5:



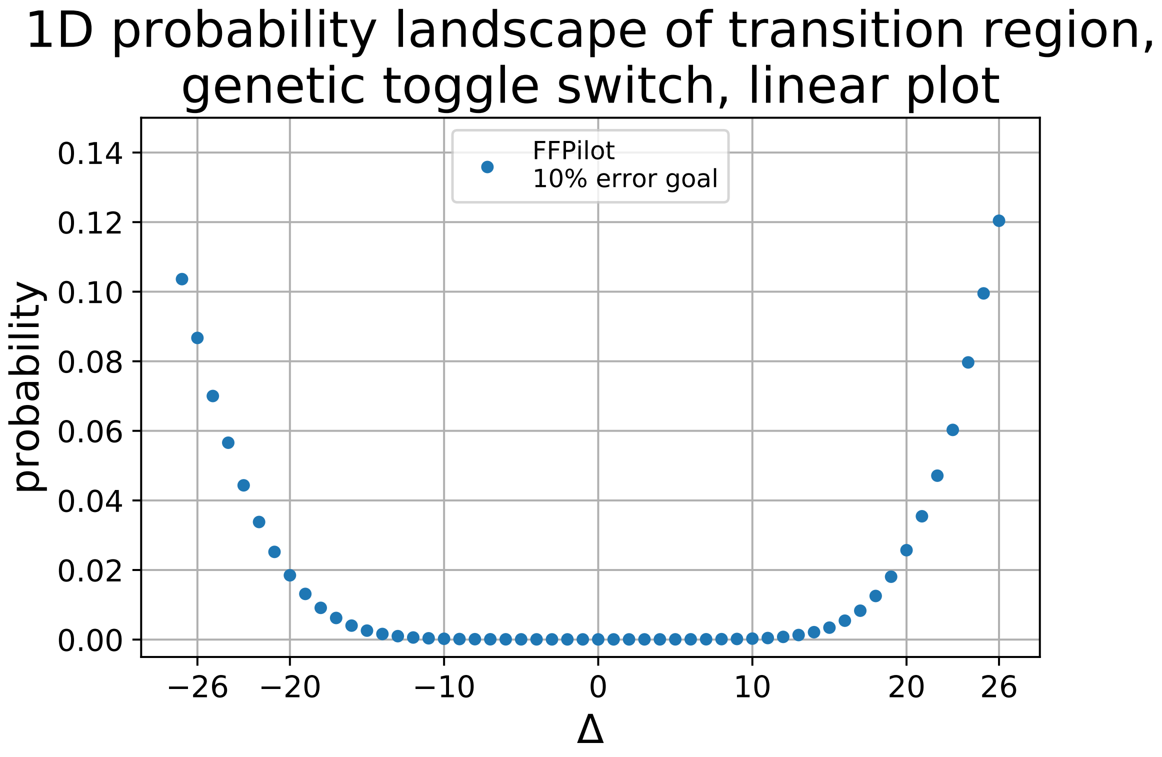
A great deal more detail can be gleaned when the transition is plotted using the 2D {**A**, **B**} order parameter:



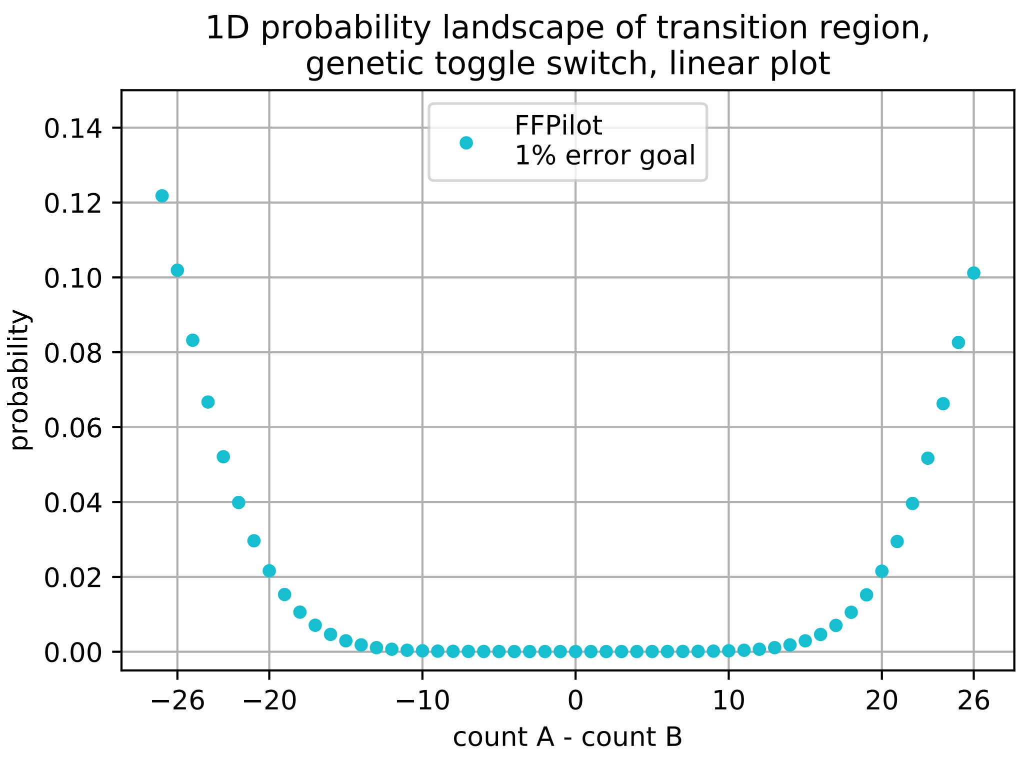
5.5.2 Accuracy of the calculated transition region landscape

In terms of the 1D order parameter Δ (ie count A − count B), the GTS landscape is symmetric over the origin. In terms of the 2D order parameter {count A, count B}, GTS is symmetric over the line $x = y$. These symmetries are important, since they offer a simple way to test the accuracy of our landscape assembly procedure (which includes both the simulation and the analysis). For any landscape we calculate, the closer its two halves are to perfect symmetry the more accurate the assembly procedure was.

Symmetry, or lack thereof, is usually easier to spot in plots of 1D order parameters. Though somewhat analytically crude, a comparison of any two symmetric points $-x, x$ in the landscape can be used as a simple, quantitative measure of the landscape's overall accuracy. A sense of how FFPilot error goal affects the accuracy of the landscape can be gained by comparing the two symmetric points $\Delta = -26$ and $\Delta = 26$. When a landscape is calculated from a 10% error goal FFPilot simulation, the divergence between the probabilities of -26 and 26 tends to be quite large, on the order of $10 - 20\%$:



On the other hand, as in the plot shown below, the probabilities of -26 and 26 are nearly equal when calculated using data from a 1% error goal simulation:

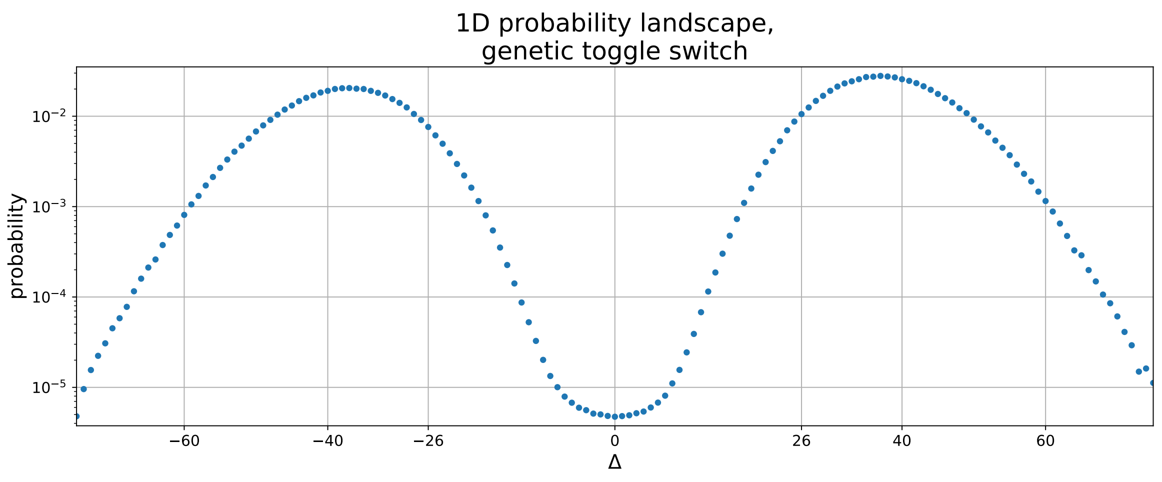


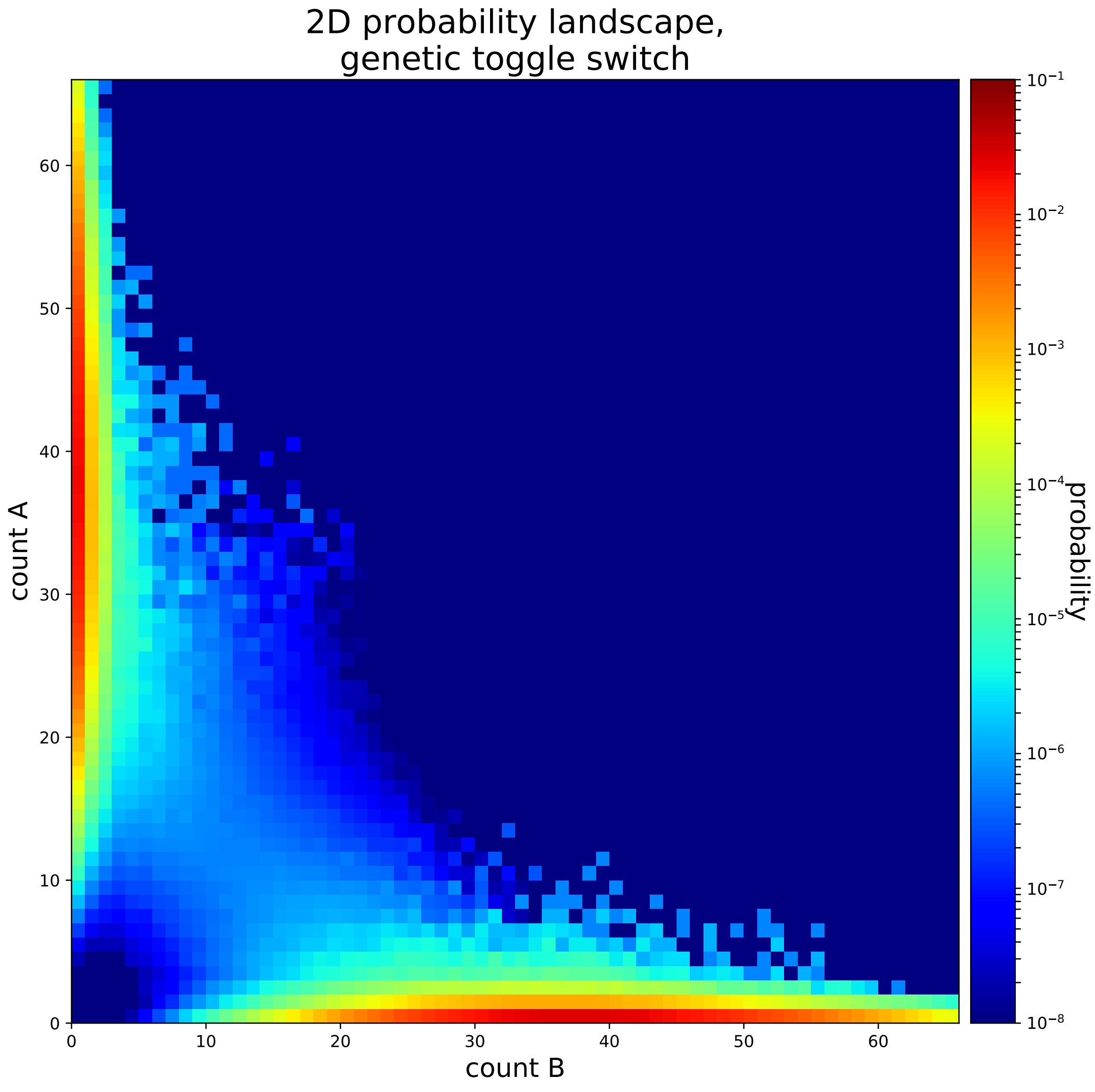
The landscape shown above was produced from the example data included with the notebook (in the `example_arrays/landscape_ffpilot_basin_%d.npz` files).

5.6 The complete unbiased landscape of GTS

5.6.1 Calculating the complete landscape

Now that we have the transition landscape assembled, the complete landscape can be calculated by simply reusing the code from Sec 4.4.6. Plotted in 1D (along the □ order parameter) and 2D (along the {**A**, **B**} order parameter) it looks like:

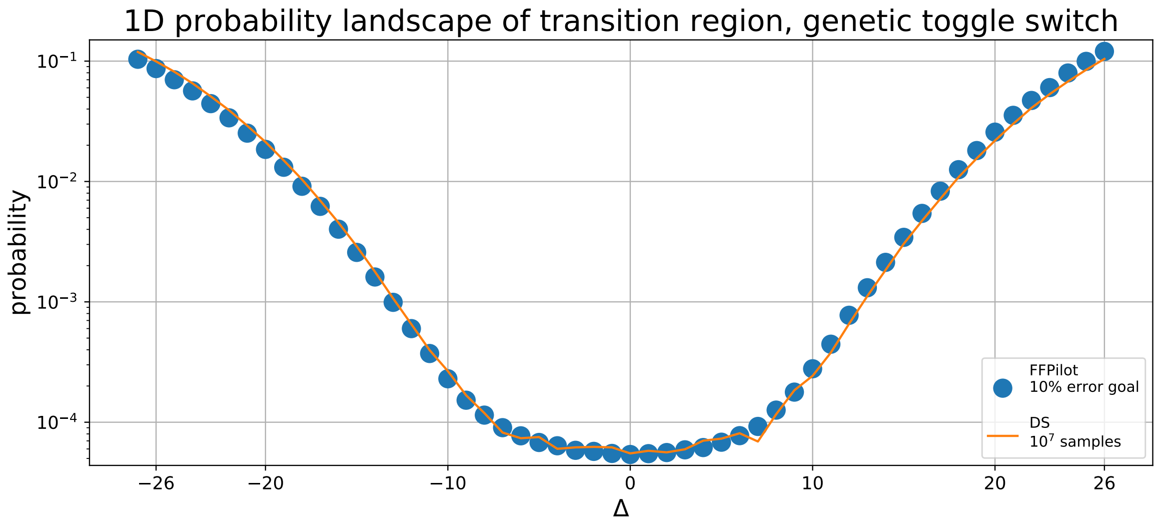




As you can see above, the fitting procedure in the code we wrote last chapter is robust and flexible enough that it is able to deal with both the SRG and the (very much different) GTS models.

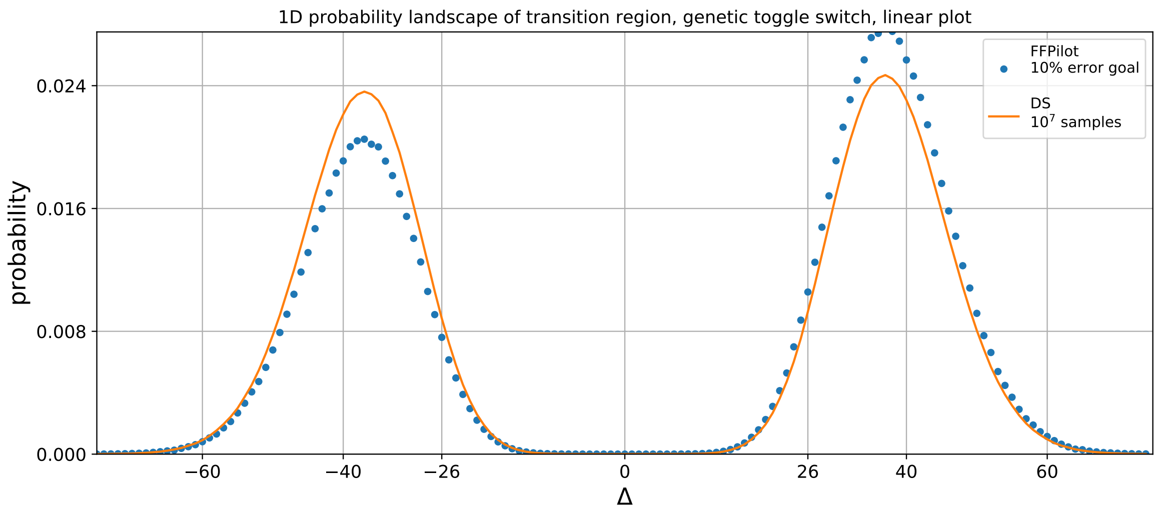
5.6.2 Accuracy of calculated complete landscape: comparison with replicate sampling

In addition to comparing them against each other, we can also gauge the accuracy of the two halves of a GTS FFPilot simulation by comparing them to the results from a direct sampling (DS) simulation. Though DS simulation is computationally inefficient, it is highly accurate. It has been shown [4] that DS simulation will converge roughly monotonically to the correct answer under a large variety of conditions. This convergence behavior, along with a history of decades of use and verification, allows DS results to be used as a sort of gold standard when examining novel simulation methods. Here is a comparison of the transition region calculated by FFPilot and DS simulations:



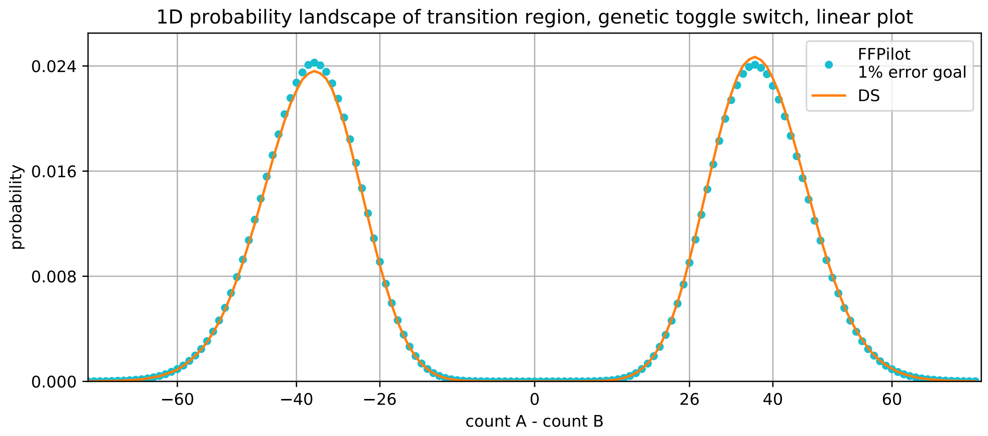
The landscape produced by FFPilot is much less noisy than the landscape from direct sampling in the immediate neighborhood of the transition point. In terms of landscapes, FFPilot simulation is at its most useful in any region of extremely low probability. Direct sampling tends to struggle with these low probability regions, and can only collect enough samples to map their landscapes with reasonable accuracy when vast computational resources are used.

This version of the genetic toggle switch is completely symmetric in terms of A and B. Thus, the two peaks in the probability distribution should be the same height. The peek on the left corresponds to state $\mathcal{A}$ (in which there is a high count of protein A and a low count of protein B) and the peek on the right corresponds to state $\mathcal{B}$ (in which there is a high count of protein B and a low count of protein A). In terms of the 1D □, the two peeks $\mathcal{A}$ and $\mathcal{B}$ fall at -37 and 37, respectively.



In the direct sampling data we show above, the height of the state $\mathcal{A}$ and state $\mathcal{B}$ peeks differ by about $\sim 4.5\%$ relative to one another. Whether FFPilot under- or out-performs this figure depends on the error goal with which a simulation is run. Error goal is expressed in terms of a desired level of error in the estimated MFPT from a simulation, but a lower error goal also tends to reduce error in other measures as well, including this landscape symmetry measure.

In landscapes produced from FFPilot simulations run at an error goal of 10%, the two peeks can differ by 10% or more. However, the difference in the peeks decreases dramatically for FFPilot simulations run with a 1% error goal. The landscape shown below was produced from the example data included with the notebook (in the `data_example/landscape_ffpilot_basin_%d.npz` files), which was obtained from an FFPilot simulation run with a 1% error goal:



In the landscape produced by FFPilot the heights of the $\mathcal{A}$ and $\mathcal{B}$ peeks are closely matched, and differ by less than 0.7%. In this case FFPilot outperformed direct sampling in terms of correctly reproducing the underlying symmetry of the genetic toggle switch's landscape.

Appendix A

Setting up FFPilot simulation input using the HDFView gui

A.1 Overview

A.2 Self regulating gene setup

1. Convert the `self_regulating_gene.sbml` file to the standard `.lm` simulation input format, following the instructions given in .
2. Use `HDFView` to manually add the FFPilot-specific simulation input to the resulting `self_regulating_gene.lm` file:
   (a) Open `self_regulating_gene.lm` in `HDFView`. `HDFView` is the standard file browser for `.hdf5` files and is available on the https://support.hdfgroup.org/products/java/hdfview/.
   (b) Add the order parameter:
      i. Right click on `self_regulating_gene.lm` in the column on the left, and then select `New → Group`. Enter `OrderParameters` as the new group's name and then hit `Ok`.
      ii. Right click on the new `OrderParameters` group and then repeat 2(b)i in order to create a new subgroup named `0000000`. NB: make sure the subgroup name is exactly seven zeros (internally, `LMES` uses a seven digit fixed-width format (c-style format `%07d`) for this subgroup name). .
      iii. Right click on `0000000`, choose `Show Properties`, and then click on the `Attributes` tab in the properties window. Add two attributes, `ID` and `Type`. Both should be 64-bit integer scalars, and both should have a value of 0.
      iv. Add a dataset called `SpeciesCoefficients` to the `0000000` group. It should be of type 64-bit float, and it should have a size of 1. Double click on `SpeciesCoefficients` to open it, and then set its single value to `1.0`.
      v. Add another dataset to `0000000`. Name this one `SpeciesIDs`, set its type to 32-bit unsigned integer, and set its size to 1. Open `SpeciesIDs` and make sure that its single value is set to 0 (which it should be by default).
   (c) Add the tiling (also called the interfaces, or the bins):
      i. Add a `Tilings` group with a `0000000` subgroup, just as you did for `OrderParameters`.

ii. Set three attributes on the `0000000` subgroup, `ID`, `OrderParameterID`, and `type`. All of them should be 64-bit integer scalars with a value of 0.

iii. Add a dataset called `Basins` to `0000000`. It should be of size 1, of type 32-bit unsigned integer, and its single value should be set to 10.

iv. Add a dataset called `Edges` to `0000000`. `Edges` should be of type 64-bit float, and should have a size of 13. The first value should be `23.0`, the last value should be `150.0`, and the remaining values should be evenly spaced in between. The easiest way to enter these values is to import them from `files/self_regulating_gene_mfpt/edges.txt`, a plain text file that contains the needed values, one per line. To import the edges open `Edges`, select `Import Data from Text File` from the `Table` menu in the upper left hand corner of `Edges`, select the `edges.txt` file, and then click okay on any prompts that pop up.

(d) No other data is required to run an FFPilot simulation. There are, however, a number of options that can be used to tweak FFPilot's execution. These options can be set by adding the appropriate attributes to the top level `Parameters` group in the input `.lm` file:

i. The overall accuracy of the simulation is controlled by the `errorGoal` option. Add an attribute to `Parameters` called `errorGoal` with a scalar float value of 0.05.

ii. By default, the output of an FFPilot simulation will consist of a single record, of type `FFPilotStageOutput`, that contains the primary results from the FFPilot production stage. For the purposes of this example simulation, we'll turn on the output of the `FFPilotStageOutputRaw` record, which contains various intermediate data. Add a `Parameters` attribute called `ffpilotStageOutputRaw` with a string value of ”True” .

iii. We'll also turn on the output of the `FFPilotStageOutput` and `FFPilotStageOutputRaw` for the pilot stage. Add a `Parameters` attribute called `ffpilotPilotOutput` with a string value of ”True”.

List of Abbreviations

DS	direct sampling
GTS	genetic toggle switch
LMES	Lattice Microbes ES
MFPT	mean first passage time
SRG	self-regulating gene

Bibliography

[1] Valeriani, C, Allen, RJ, Morelli, MJ, Frenkel, D, Rein ten Wolde, P (2007) Computing stationary distributions in equilibrium and nonequilibrium systems with forward flux sampling. *J Chem Phys* 127:114109.

[2] Dickson, A, Warmflash, A, Dinner, AR (2009) Nonequilibrium umbrella sampling in spaces of many order parameters. *J Chem Phys* 130:074104.

[3] Gardner, TS, Cantor, CR, Collins, JJ (2000) Construction of a genetic toggle switch in Escherichia coli. *Nature* 403:339–342.

[4] Gillespie, DT (2007) Stochastic simulation of chemical kinetics. *Annu Rev Phys Chem*.