SloppyCell User Documentation

August 29, 2017

Contents

Welcome to SloppyCell!

SloppyCell is a software environment for building and analyzing computational models of many-parameter complex systems. To date, our focus has been on biochemical network models based on the Systems Biology Markup Language (SBML) [1]. Many of the techniques and much of the code is, however, applicable to other types of models.

SloppyCell's goal is to provide a flexible environment that facilitates interactive exploratory model development while also remaining efficient for computation-intensive analyses. This goal is facilitated by our use of the computing language Python (`http://www.python.org`) for both writing and driving SloppyCell. SloppyCell scripts are Python programs, but please don't let that frighten you. Python has proved easy to learn and there are numerous tutorials (e.g. `http://wiki.python.org/moin/BeginnersGuide`) and useful books [2]. An excellent introduction to Python in the context of scientific computing is the May/June issue of Computing in Science and Engineering.

SloppyCell's development has been driven by the research interests of the Sethna group. For examples of the types of analysis we do, see our papers [3, 4, 5, 6, 7, 8, 9, 10].

This document opens begins with an example application, followed by a high-level overview of SloppyCell's architecture. We then delve into additional features and installation (Section 3), and we close with some troubleshooting.

Note: If you are using a binary installer, you will still want to download the source distribution (as described in Section 3.3) so that you will have access to the `Example` and `Doc` directories.

1 JAK-STAT Example

Here we consider an example application using the JAK-STAT model of Swameye et al. [11], which was also used as an example by the SBML Parameter Estimation Toolkit (SBML-PET) [12].

Using the script shown in Listing 1, we'll fit the model, build an ensemble, and then estimate some prediction uncertainties. (This script is also found in the `Example/JAK-STAT` directory of the SloppyCell source distribution.) Some of the concepts may be a little subtle, so don't be afraid to turn to any section in the Overview and come back here.

Lines 1 through 3 `import` code from the Python packages we'll use this session: matplotlib, SciPy, and SloppyCell.

On line 5 we load the model network from the SBML file. (Note that this file has been modified from the one included with SBML-PET to fix some bugs in it and to define the species `'data1'` and `'data2'` which correspond to the measurements we'll fit.) Given the space constraints for listings here, the `Experiment` object is defined in another file, which we `import` on line 8. We use that `Experiment`, along with our previously created `Network`, to create the `Model` object on the following line.

Our initial parameters are defined as a `KeyedList` starting on line 11. We could have specified them as a simple `list`, without the names, but we find things are much clearer when the names are visible as well.

```
from pylab import *                                                          #
from scipy import *
from SloppyCell.ReactionNetworks import *                                    #

net = IO.from_SBML_file('JAK-STAT_SC.xml', 'net1')                           #
net.set_var_ic('v1', 'v1_0') # Won't need given initial assignments.         #

import JAK_expt                                                              #
m = Model([JAK_expt.expt], [net])

params = KeyedList([('r1', 0.5), ('r3', 2), ('tao', 6.0),                    #
                    ('r4_0', 1.35), ('v1_0', 1.19)])

res = Residuals.PriorInLog('r3_prior', 'r3', 0, log(sqrt(1e4)))              #
m.AddResidual(res)
res = Residuals.PriorInLog('tao_prior', 'tao', log(4), log(sqrt(4)))
m.AddResidual(res)                                                           #

print 'Initial cost:', m.cost(params)                                        #
params = Optimization.fmin_lm_log_params(m, params, maxiter=20, disp=False)  #
print 'Optimized cost:', m.cost(params)
print 'Optimized parameters:', params

# Plot our optimal fit.
figure()
Plotting.plot_model_results(m)                                               #
savefig('model_results.pdf')

j = m.jacobian_log_params_sens(log(params))                                  #
jtj = dot(transpose(j), j)                                                   #

print 'Beginning ensemble calculation.'
ens, gs, r = Ensembles.ensemble_log_params(m, asarray(params), jtj, steps=7500)#
print 'Finished ensemble calculation.'

pruned_ens = asarray(ens[::25])                                              #

figure()
hist(log(pruned_ens[:,1]), normed=True)                                      #
savefig('hist.pdf')

times = linspace(0, 65, 100)
traj_set = Ensembles.ensemble_trajs(net, times, pruned_ens)                  #
lower, upper = Ensembles.traj_ensemble_quantiles(traj_set, (0.025, 0.975))

figure()
plot(times, lower.get_var_traj('frac_v3'), 'g')
plot(times, upper.get_var_traj('frac_v3'), 'g')
plot(times, lower.get_var_traj('frac_v4'), 'b')
plot(times, upper.get_var_traj('frac_v4'), 'b')
savefig('uncerts.pdf')

show()
```

Listing 1: This script reproduces some of the results from [13]. For detailed comments see Section 1.

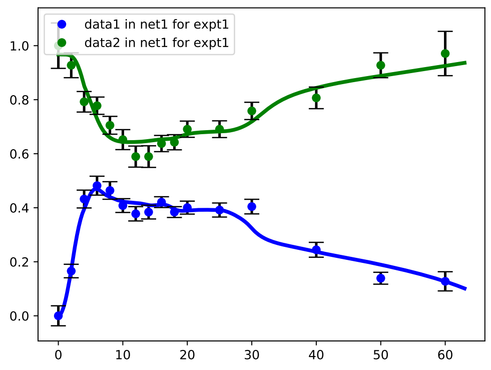

Figure 1: Plotted is the optimal fit for our example model, generated using `Plotting.plot_model_results(m)` in Listing 1.

A couple of priors need to be defined to keep model parameters from drifting too far. The prior on `'r3'` (line 14) constrains it (within 95% probability) to lie between 10^{-4} and 10^{4}, while the prior on `'tao'` constrains it to lie between 1 and 16.

The initial cost is about 560, which is very high given that we only have 32 data points. Thus we run several iterations of Levenberg-Marquardt on line 20. (We limit the number of iterations here merely for expediency. This number gets us very close to the actual minimum.) The final cost should be about 18.2. For a perfectly fitting model, we expect a cost of 1/2 the number of data points, so this cost indicates a good fit. On line 26 we generate Figure 1, which compares our best fit with the data.

Our group's philosophy is, however, not to trust solely in the best fit, so we'd like to build an ensemble of parameters. Before we can build an ensemble, we need to build a matrix to guide the sampling. Here we use the $J^{\mathsf{T}}J$ approximation to the Hessian, which we calculate on lines 29 and 30. (As an aside, the eigenvalues and eigenvectors of this $J^{\mathsf{T}}J$ are 'sloppy', as with the models discussed in [3, 6, 8].)

On line 33 we build a parameter ensemble. We only build a 7500 step ensemble because the model is quite small and well-constrained; with 5 parameters the correlation time should be only about 25 steps. Also, we cast the `params` `KeyedList` to an array in the call; this makes our returned `ens` be composed of arrays rather than `KeyedLists`, which is more memory efficient. Calculating a 7500 member ensemble for this model takes approximately 15 minutes on a modern PC. On line 36 we prune the ensemble; using slicing notation to take every 25th element (25 being the correlation time). We also convert to an array, so that we can use the more powerful slice syntax of arrays.

What does the ensemble actually look like? On line 39, we use matplotlib's `hist` function to build a histogram of the logarithm of `'r3'` over the ensemble. The result is shown in figure 2. Note that the upper bound on `'r3'` is set by the prior we added, while the lower bound is constrained by fitting the data.

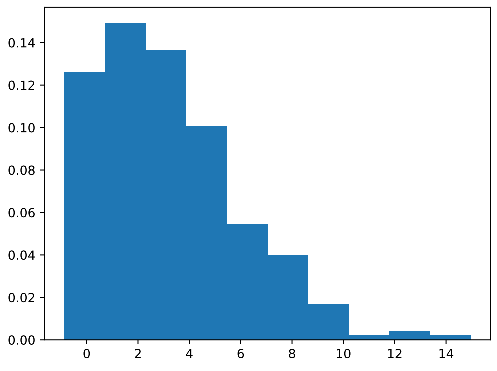

Figure 2: Shown is a histogram of log `'r3'` for the JAK-STAT ensemble. Note that the value of `'r3'` is bounded from below at about 10^0, but the upper end is only bounded by the prior.

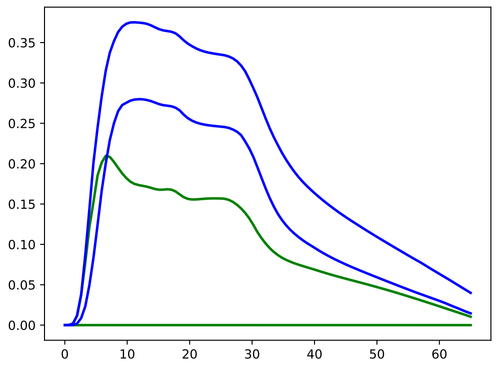

Figure 3: In green is the 95% uncertainty bound on cytoplasmic dimers, while blue is the 95% bound on nuclear dimers, given the model and data that have been fit.

Figure 4: The architecture of some of the primary SloppyCell tools is illustrated here. `Model`, `Network`, and `Experiment` are classes, and a model is composed of one or more `Networks` and `Experiments`. The oval boxes represent modules (collections of functions) and the dotted arrows indicated the class of objects they primarily act upon.

Once we have our ensemble, we can make some predictions. On line 43 we calculate trajectories for `net` over our pruned ensemble, and on the following line we generate `lower` and `upper` trajectories that bound the central 95% of the values for each variable. We then plot these bounds for the variables `'frac_v3'` and `'frac_v4'`, which have been defined to be `'2*v3/v1'` and `'2*v4/v1'` respectively. These are the fractions of total STAT that are involved in cytoplasmic and nuclear dimers. Figure 3 shows the resulting figure. Note the relatively large uncertainty of the cytoplasmic dimers (green), which gets very close to zero.

Finally, we end our script with a call to `show()`. This `pylab` command ensures that the plots pop up. It may be unnecessary if you're running the script from within an IPython session started with the `-pylab` command-line option.

1.1 Other Examples

The JAK-STAT example presented here covers the most basic of SloppyCell's functionality. More extensive examples can be found in the `Examples` directory of the source distribution.

2 Overview

2.1 Working Interactively

Python is an interpreted language, which means that there is no compile step between writing and running the code. Thus Python can be used interactively, and this is one of its greatest strengths, particularly when used with the enhanced IPython shell [14].

A typical pattern for working with SloppyCell is to have one window running the IPython shell and another open to your favorite text editor. Commands can be directly typed and run in the shell, and when the results are satisfactory they can be recorded into a `.py` script using the text editor. This pattern is particularly powerful when used with IPython's 'magic' `%run` and `%run -i` commands, which allow external scripts to be run as if all their commands were typed into the shell.

The interactive nature of Python is also important for getting help. Information and useful documentation can be accessed about any object using `help(<object>)`, and all the attributes (data and methods) of an object can be listed using `dir(<object>)`. IPython makes such exploration even more powerful, as it offers `<tab>` completion of object attributes. For example, to see all the method of the object `net` that begin with `calc`, one would type `net.calc<tab>`.

Rather than expounding upon the details of every function and method in SloppyCell, this document will focus on a higher-level description. Our aim here is to show you what SloppyCell can do and where to look for that functionality. Once you have that, the interactive help should guide you on how exactly the functions work.

2.2 Accessing SloppyCell

To access the tools provided by SloppyCell, most user scripts should have

```
from SloppyCell.ReactionNetworks import *
```

near the top of the script. This imports most of SloppyCell's modules into the top-level namespace where they can be easily accessed.

2.3 Networks

At the heart of most SloppyCell projects is a collection of `Network` objects. A `Network` describes a set of chemical species, their interactions, and a particular set of experimental conditions. SloppyCell's `Networks` are based on the SBML Level 2, Version 3 specification (`http://sbml.org/documents/`). A concise and mostly complete summary of this specification can be found in the paper by Hucka et al. [1] (available from `http://sbml.org/documents/`).

Briefly, an SBML network consists of *species* (that exist inside *compartments*) whose dynamics are controlled by *reactions* and *rules* (*assignment*, *rate*, *algebraic*, *initial assignment*). A network also describes *parameters* which typically quantify the interactions, and *events* which cause discontinuous changes in model components given specified triggers. A network can also specify mathematical *function definitions* for use in other expressions.

`Networks` are constructed simply as

```
net = Network('example').
```

All the SBML components are added to a network using methods that begin with `add`, e.g.

```
net.add_parameter('kf', 0, is_optimizable=True).
```

This example shows one additional attribute SloppyCell assigns to parameters; if they are `constant`, they can additionally be declared `optimizable`. When Networks are composed into a `Model`, the `optimizable` parameters are exposed at the `Model` level so they can be tweaked by optimization algorithms or when building an ensemble.

Often one wishes to study several slight modifications to a single `Network`. To that end, `Networks` have a `copy` method.

SloppyCell supports most of the current SBML specification, but there are some exceptions. First, SloppyCell does no processing of units. It is assumed that all numerical quantities in the `Network` have compatible units. Second, we don't support delay elements in math expressions. (Delays in event execution times are supported.) Finally, because we often take analytic derivatives of the equations, using discontinuous functions (e.g. `abs()` or `ceil()`) in the math for reactions or rules will cause integration problems. Discontinuities should be coded using events, and it is supported to use `piecewise` in the event assignments.

2.4 Dynamics

The `Dynamics` module contains methods to integrate a `Network`'s equations. The most basic functionality is `Dynamics.integrate` which simply integrates a model's differential equations forward in time and returns a `Trajectory` object containing the result.

Also quite useful is `Dynamics.integrate_sensitivity`, which returns a `Trajectory` object containing the *sensitivity* trajectories. These trajectories are $\partial y(t, \theta)/\partial \theta_i$, the derivatives of a given variable at a given time with respect to a given optimizable variable (indexed by e.g. `(y, theta_i)`). These trajectories are useful for optimizing parameters or experimental designs.

Finally, SloppyCell implements basic fixed-point finding in `Dynamics.dyn_var_fixed_point`.

2.5 Models

A `Model` object unites one or more `Network`s with the data contained in one or more `Experiments`:

```
m = Model([<list of expts>], [<list of nets>])
```

A `Model`'s primary task is to calculate the cost C for a set of parameters θ, defined as:

$$C(\theta) \equiv \frac{1}{2} \sum_i \left(\frac{B_i \, y_i(\theta) - d_i}{\sigma_i} \right)^2 + \text{priors}. \tag{1}$$

Here $y_i(\theta)$ is the model prediction, given parameters θ, corresponding to data point d_i, and σ_i is the uncertainty of that data point. The B_i are *scale factors* which account for data that are only relative, not absolute, measurements (e.g. Western blots). See Section 2.6.1 for more on scale factors. The 'priors' term in the costs represents additional components of the cost, often designed to steer parameters in particular directions.

2.5.1 Priors

Often it is useful to add additional 'prior' terms to the cost. These may reflect previous direct measurements of a particular parameter, or restrict them to physically reasonable values. Prior terms are added using `m.add_residual(res)` where `res` is a `Residual` object. By far the most common form of additional residual we use is `Residuals.PriorInLog`. Such a residual adds a term to the cost of the form:

$$\frac{1}{2} \left(\frac{\log \theta_i - \log \theta_i^*}{\sigma_{\log \theta_i}} \right)^2 . \tag{2}$$

This acts to keep the logarithm of parameter θ_i from deviating much more than $\sigma_{\log \theta_i}$ from $\log \theta_i^*$.

2.6 Experiments

An `Experiment` object contains describes a set of data and how it should be compared with a set of `Networks`s.

2.6.1 Scale Factors

Each `Experiment` defines a set of measurements in which *all measurements of the same quantity share a scale factor*. Scale factors are important for many forms of biological data which do not give absolute measurements. For example, the intensity of a band in a Western blot is proportional to the concentration of that protein in the sample, but converting it to an absolute value may be very difficult to do reliably. The optimal conversion factor for comparison to a given set of `Network` results is, however, easy to calculate analytically and need not be included as an extra fitting parameter. For historical reasons, by default SloppyCell assumes that *all data involve scale factors that should be optimized.* If you know the absolute scale of your data, use `expt.set_fixed_sf` to specify *fixed* scale factors. Or, if you know that two variables should share a scale factor which needs to be optimized, use `expt.set_shared_sf`.

Because SloppyCell handles the scale factors implicitly, when building an ensemble we must account for their fluctuations by using a *free energy* rather than the previously mentioned `cost`. This free energy depends on prior assumptions about how the scale factors are distributed, and these priors can be changed using `expt.set_sf_priors`. For more on possible effects and subtleties of choosing these priors, see [10, Section 6.2].

2.6.2 Data Format

SloppyCell's data format is simply a set of nested Python dictionaries or `KeyedLists`. This can be unwieldy to write out by hand, but it provides flexibility for future addition of more complex forms of data, and it can easily be generated from other tables by simple scripts. The first level of nesting in the experimental data is keyed by the `id` of the `Network` whose results the data should be compared with, and the next level is keyed by the variable the data refers to. The final level is a dictionary or `KeyedList` of data-points, mapping times to tuples of `(<value>, <one-sigma uncertainty>)`.

The data format is best illustrated by example; see Listing 2. This `Experiment` contains data on two variables `'X'` and `'Y'` in two conditions, corresponding to the `Networks` `'net1'` and `'net2'`. Note that, because they are in the same experiment, the two data sets on `'X'` will be fit using the same scale factor. This might be appropriate, for example, if the data came from Western blots using very similar numbers of cells and the same antibody.

2.7 Optimization

After the `Model` has been created, it is very common to want to optimize the parameters θ to minimize the cost and thus best fit the data. SloppyCell includes several optimization routines in the `Optimization` module. These include wrappers around SciPy's Nelder-Mead and

```
expt.set_data({'net1':{'X': {2.0: (2.85, 0.29),
                             5.0: (4.9, 0.49),
                             },
                       'Y': {2.0: (1.25, 0.13),
                             5.0: (1.12, 0.13),
                             },
                       },
               'net2':{'X': {2.0: (6.7, 0.3),
                             4.2: (9.8, 0.2),
                             },
                       }
               }
              )
```

Listing 2: Shown is an example of SloppyCell's data format. This data set contains data on the species with `ids` `'X'` and 'Y' taken under two conditions corresponding to the `Networks` with `ids` `'net1'` and `'net2'`. Importantly, the two conditions by default *share* a floating scale factor for `'X'`.

conjugate gradient routines and SloppyCell's own implementation of Levenberg-Marquardt. The conjugate gradient and Levenberg-Marquardt routines use analytical derivatives calculated via sensitivity integration, and all the routines have versions for both logarithmic and bare parameters.

All the currently implemented methods do only local optimization, but minimizing the cost using any Python-based minimization algorithm is straight-forward. To further explore parameter space, consider building a parameter ensemble (Section 2.9).

2.8 Cost and Residual Derivatives

In both optimization and ensemble building, various derivatives of the cost function are very useful. These are all available using methods of the `Model` object, and in each case there are versions for logarithmic and bare parameters. Using sensitivity integration, we can calculate all first derivatives semi-analytically, without reliance on finite-difference derivatives and the resulting loss of precision.

Most basic derivative is the gradient of the cost which is useful for many deterministic optimization algorithms.

A slightly more complicated object is the Jacobian J, which is the derivative matrix of residuals versus parameters: $J_{i,j} \equiv dr_i/d\theta_j$. (The cost we consider (Equation 1) is a sum of squared residuals: $C(\theta) = \frac{1}{2}\sum_i r_i(\theta)^2$). The Jacobian is useful for understanding which parameters impact which features of the fit and for clustering parameters into redundant sets [15].

Finally, the Hessian H is the second derivative matrix of the cost: $H_{i,j} \equiv dC(\theta)/d\theta_i d\theta_j$. If calculated at a minimum of the cost, the Hessian describes the local shape of the cost basin.

This makes it useful for importance sampling when building an ensemble (Section 2.9). Note, however, that Hessian calculation relies on finite-difference derivatives, which can be difficult to calculate reliably. For our least-squares cost functions, a very useful approximation to the Hessian is $J^{\mathsf{T}}J$, which can be calculated as:

```
j = m.Jacobian_log_params_sens(log(params))
jtj = dot(transpose(j), j)
```

The approximation becomes exact when the model fits the data perfectly.

2.9 Ensembles

To explore the full nonlinear space of parameters that are statistically consistent with the model and the data, we build a Bayesian ensemble where the relative likelihood of any parameter set θ is:

$$P(\theta) \propto \exp\left(-G(\theta, T)/T\right). \tag{3}$$

Here $G(\theta, T)$ is the *free energy*, which is the cost plus a possible contribution due to fluctuations in scale factors. The temperature T controls how much we're willing to let the free energy deviate from the optimum. For strict statistical correctness, it should be one, but there are situations in which it is useful to adjust user larger values [5].

The ensemble is built using `Ensembles.ensemble_log_params`, using an importance-sampled Markov-Chain Monte-Carlo algorithm [16]. This algorithm builds the ensemble by taking a random walk through parameter space which, eventually, will converge to the correct probability distribution.

2.9.1 Assessing and Speeding Convergence

'Eventually' is a key word in describing ensemble convergence. Because we are taking a walk through parameter space, subsequent members of the ensemble are highly correlated. Generating a thoroughly converged ensemble that independently samples the distribution of parameters many times can be quite computationally intensive for large models.

There are several ways to assess convergence. Most thorough (but computationally expensive) is to start several ensembles from very different initial parameters and see that they give identical answers for your predictions.

Given a single ensemble, one can check for convergence using the autocorrelation function of the cost and logarithms of the parameter values using `Ensembles.autocorrelation`. Figure 5 shows example autocorrelation functions for a small model. The number of independent samples in an ensemble is approximately the length of the ensemble divided by the *longest* correlation time of any parameter in the ensemble. Scaling arguments suggest that, for the default ensemble parameters, the number of steps in one autocorrelation time is at least the square of the number of parameters [10, Section 3.5.1].

If the correlation time for your ensemble is much longer than the square of the number of parameters, your cost basin is probably substantially curved. For advice on how to deal with this, see [10, Section 6.3].

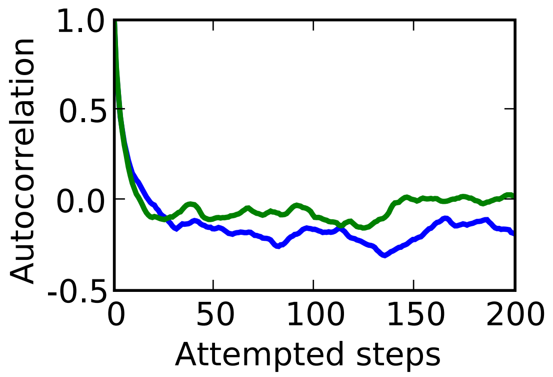

Figure 5: Shown are autocorrelation functions functions for the cost (blue) and the logarithm of a particular parameter (green) in an ensemble built for the small JAK-STAT example model (Section 1). The correlation time for both is about 25 steps, suggesting that parameter sets 25 steps apart are statistically independent.

2.9.2 Predicting from an Ensemble

Once your ensemble has converged, you're ready to make predictions. As a first step, you'll probably want to *prune* your ensemble. As mentioned previously, consecutive members of the ensemble are not independent, and it is independent samples that matter for predictions. Once you've estimated the longest correlation time (`corr_steps`), the ensemble can be pruned simply by taking one member per correlation time: `ens_pruned = ens[::corr_steps]`.

To calculate uncertainties for any quantity over the ensemble, simply calculate its value for each member of your pruned ensemble and note the spread in values. SloppyCell includes a few functions to make this easier in some common cases. `Ensembles.ensemble_trajs` will integrate a `Network` for a fixed set of times over all members of an ensemble. `Ensembles.traj_ensemble_quantiles` will calculate quantiles over those integrations, to show, for example, what the 95% confidence bounds are on the trajectory prediction. Similarly, `Ensembles.traj_ensemble_stats` will return trajectories containing the mean and standard deviation of each integration quantity over an ensemble.

2.10 Plotting

SloppyCell's plotting functionality is built upon matplotlib [17], also known as `pylab`; a nice tutorial is available at `https://matplotlib.org/users/pyplot_tutorial.html`. SloppyCell's `Plotting` module adds several routines that are convenient for analyzing fits, ensembles, and Hessians. For example, Figure 1 shows a plot of the best fit for our example model (Section 1) with no additional tweaking.

2.11 KeyedLists

Numerical algorithms are generally designed to work with arrays of values, while users don't want to remember which parameter was number 97. To solve both issues, SloppyCell uses a custom data type called a `KeyedList` which has properties of both Python lists and dictionaries. Like a normal list, values can be accesses using `[<index>]` notation. Additionally, each entry is associated with a key, so that values can be added and accessed using `get` and `set` methods like a Python dictionary.

2.12 Input and Output

SloppyCell's `IO` module includes several functions to facilitate import and export of useful representations of `Networks`.

`IO.from_SBML_file` and `IO.to_SBML_file` allow importing and exporting of SBML files. (Note that SloppyCell will not preserve annotations of an SBML network that has been imported.) `IO.eqns_TeX_file` will export a `tex` file that can be used with LaTeXto generate nicely-formatted equations. These are useful both for inclusion in publications and for debugging models.

2.13 Miscellaneous Utilities

To conveniently save results for future analysis, you can use `Utility.save` and `Utility.load`. Note that these rely on Python's binary 'pickle' format, so there is a slight danger that upgrades to Python or SloppyCell will render them unreadable. For more robust but less space-efficient saving, dump results to text files using, for example, `scipy.io.write_array`.

3 Installation

3.1 Required Dependencies

SloppyCell requires Python 2.x and the libraries NumPy, SciPy, and matplotlib. As of late 2017, the easiest way to install these is through the Anaconda (`https://www.anaconda.com/download/`) or Enthought (`https://www.enthought.com/product/canopy/`) distributions. Both include many scientific Python libraries and a package manager for easy installation.

3.2 Optional Dependencies

To read and write files encoded in the Systems Biology Markup Language [1], libSBML is required (`http://sbml.org/Software/libSBML/Downloading_libSBML#Python`).

For speed, SloppyCell by default generates C versions of the network equations, and this requires a C compiler. If a C compiler is not installed, SloppyCell will run with Python versions of the network equations, which may be up to a factor of 30 slower. For Windows, you'll need to install the version of Visual C++ that is compatible with your version of Python, most likely Visual C++ 9.0 (`https://wiki.python.org/moin/WindowsCompilers`). For OS X, install Xcode (`https://developer.apple.com/xcode/`). For Linux, most distributions come with a C compiler installed.

3.3 Testing the Installation

The SloppyCell source code includes a large package of test routines. To access them: download and unpack `SloppyCell-XXX.tar.gz`, descend into the SloppyCell-XXX directory (`cd SloppyCell-XXX`), and run `cd test; python test.py`. This will run an extensive series of tests, both with and without C-compilation.

4 Troubleshooting

4.1 Failing Integrations

Optimization or ensemble construction may explore regions of parameter space for which the model equations become very difficult to integrate, leading to many `daeint` exceptions being raised. One possible solution is to check the 'typical' values that are being assumed for each variable, accessible via `net.get_var_typical_vals()`. These are used to help set the absolute tolerances of the integrations, and if they are very different from the values the variable actually attains, the inconsistency can cause problems. The solution is then to set them to more reasonable values using `net.set_var_typical_val(<variable id>, <value>)`.

References

[1] Hucka M, Finney A, Sauro HM, Bolouri H, Doyle JC, et al. (2003) The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. Bioinformatics 19:524–531. back

[2] Lutz M, Ascher D (2003) Learning Python. O'Reilly, 2nd edition. back

[3] Brown KS, Sethna JP (2003) Statistical mechanical approaches to models with many poorly known parameters. Phys Rev E 68:021904. back

[4] Brown KS, Hill CC, Calero GA, Myers CR, Lee KH, et al. (2004) The statistical mechanics of complex signaling networks: nerve growth factor signaling. Phys Biol 1:184–195. back

[5] Frederiksen SL, Jacobsen KW, Brown KS, Sethna JP (2004) Bayesian ensemble approach to error estimation of interatomic potentials. Phys Rev Lett 93:165501. back

[6] Waterfall JJ, Casey FP, Gutenkunst RN, Brown KS, Myers CR, et al. (2006) Sloppy-model universality class and the Vandermonde matrix. Phys Rev Lett 97:150601. back

[7] Casey FP, Baird D, Feng Q, Gutenkunst RN, Waterfall JJ, et al. (2007) Optimal experimental design in an epidermal growth factor receptor signalling and down-regulation model. IET Syst Biol 1:190–202. arXiv:q-bio/0610024. back

[8] Gutenkunst RN, Waterfall JJ, Casey FP, Brown KS, Myers CR, et al. (2007) Universally sloppy parameter sensitivities in systems biology. PLoS Comput Biol In press, arXiv:q-bio.QM/0701039. back

[9] Gutenkunst RN, Casey FP, Waterfall JJ, Myers CR, Sethna JP (2007) Extracting falsifiable predictions from sloppy models. In: Stolovitsky G, Califano A, Collins J, editors, Reverse Engineering Biological Networks: Opportunities and Challenges in Computational Methods for Pathway Inference. New York Academy of Sciences. In press, arXiv:0704.3049. back

[10] Gutenkunst RN (2007) Sloppiness, Modeling, and Evolution in Biochemical Networks. Ph.D. thesis, Cornell University. back

[11] Swameye I, Muller TG, Timmer J, Sandra O, Klingmuller U (2003) Identification of nucleocytoplasmic cycling as a remote sensor in cellular signaling by databased modeling. Proc Natl Acad Sci USA 100:1028–1033. back

[12] Zi Z, Klipp E (2006) SBML-PET: a Systems Biology Markup Language-based parameter estimation tool. Bioinformatics 22:2704–2705. back

[13] Gutenkunst RN, Atlas JC, Kuczenski RS, Casey FP, Waterfall JJ, et al. Falsifiable modeling of biochemical networks with SloppyCell. In preparation. back

[14] Pérez F, Granger BE (2007) IPython: a system for interactive scientific computing. Comput Sci Eng 9:21–29. back

[15] Waterfall JJ (2006) Universality in Multiparameter Fitting: Sloppy Models. Ph.D. thesis, Cornell University. back

[16] Chib S, Greenberg E (1989) Understanding the Metropolis-Hastings algorithm. Amer Statistician 49:327–335. back

[17] Hunter JD (2007) Matplotlib: a 2D graphics environment. Comput Sci Eng 9:90–95. back