SloppyCell Developer Documentation

Revision : 1.1

June 3, 2009

Contents

The primary reference for developers is the code itself. This document serves to outline design decisions or particularly tricky parts of the code.

1 Test Suite

First, the importance of a good test suite cannot be over-emphasized. When large changes are made to the codebase (for example, incorporating C right-hand-side functions) comprehensive tests make it straightforward to crush many bugs before they effect end users.

SloppyCell's test suite is based on the Python `unittest` framework. The idea of a unit-test is that each test is small and self-contained, testing only a single aspect of the code. This makes it easier to track failures to their source. The `Testing/test.py` file agglomerates all the tests and runs them both with and without C-compilation. Developers should run the test suite with verbose error messages (`python test.py -v`). Make sure your CVS commits don't break the tests! *Every* CVS commit should be preceded by a run of the test suite.

The test suite is, unfortunately, incomplete. We didn't begin automated testing until much of the code was written, particularly at the `Model` level. Nevertheless, all new features should have unit-tests to ensure correctness, and adding new tests to old features is a very valuable contribution.

`test_Misc.py` is a home for tests of bugs that aren't related to specific features. Ideally, if a bug is squashed, a test should be added in this file if it doesn't fit elsewhere.

2 Logging

Rather than littering the code with `print` statements, the Python `logging` module should be used, as it allows for much finer control of messaging.

Every module begins by defining a `logger` object which is used for output from that module. Debugging messages should be logged by `logger.debug("My message")`. These will become visible when that module's logger is set to `DEBUG` mode:

```
module.logger.setLevel(module.logging.DEBUG).
```

The output of all loggers can be dumped to a file by starting any SloppyCell script as:

```
python my_scipy.py --debugSC=output_file.log.
```

Logging messages sent through `logger.warn` or `logger.critical` will be visible to the user. They should be used sparingly.

If a set of `logger.debug` statements is useful for tracking down bugs in general, it's fine to leave them in the final commit, as long as they aren't in an inner loop where they will impact performance.

3 Integrator

For integration, SloppyCell uses the differential-algebraic equation solver DASKR [1, 2]. DASKR is written in Fortran while the vast majority of SloppyCell is in Python. To interface with between the two languages, we use f2py, which has recently become part of NumPy. The Python interface to DASKR is specified in `ddaskr.pyf`.

One subtlety in the interface is the use of the variable `ipar`. When using DASKR from Fortran, `ipar` would be used to pass extra integer arguments to user-supplied functions. In our wrapper, we use `ipar` to store (1) the dimension of the system we're solving and (2) the dimension of `rpar`, the array used for passing extra double precision arguments to user-supplied functions. These entries in `ipar` must be set to the proper values by the user of the Python interface. This usage of `ipar` is necessary because the interface code f2py builds between Python and Fortran needs to have the dimension of all arrays specified.

4 ExprManip

Many important features in SloppyCell are enabled by the included `ExprManip` package, which includes a slew of methods for manipulating strings representing Python math expressions. These manipulations including extracting all the variables, making substitutions, and taking analytic derivatives. All of these operations begin by building an Abstract Syntax Tree (AST) using the `compiler` module. This tree (a list of lists of lists...) is then manipulated, often recursively.

One caveat with these methods is that they are relatively slow. Perhaps surprisingly, the bottleneck is not in recursively descending the AST, but rather in generating the AST itself. Given that the AST is generated by the standard `compiler` module, it may be difficult to speed this up. Thus it is important to minimize the number of redundant ASTs that are built. A first round of such optimization cut the time to compile large networks by a factor of five to ten.

5 Dynamic Functions

To integrate or otherwise analyze a specific `Network`, SloppyCell dynamically generates a number of functions. When possible, C versions of these functions are built and compiled, with Python versions used as a fall-back.

5.1 Compilation

The Python and C codes for all the dynamic functions are generated upon a call to `net.compile`. To ensure that the dynamic functions are up-to-date with the structure of the `Network`, `compile` is called before every integration. Generating the code for the dynamic functions is, however, quite slow, so `compile` first checks whether the structure of the `Network` has changed (as defined by `net._get_structure()`) before generating new code.

If `compile` decides it is necessary to regenerate the dynamic functions, all the methods in `Network._dynamic_structure_methods` are called in order. These methods should fill the `KeyedList net._dynamic_funcs_python` with the Python code for all the desired dynamic functions. Similarly, these methods can fill `self._prototypes_c` and `self._dynamic_funcs_c` with the function prototypes and code for all the functions that should be compiled in C. Thus to add a new dynamic function to the `Network` class, one simply writes a method in `Network` that will add entries to the appropriate `_dyamic_funcs` lists and adds that method to `Network._dynamic_structure_methods`. If the added function has a C version, the file `f2py_signatures.pyf` will need to be updated with an appropriate interface signature. (Note, because `Network`s are often modified solely by adding or removing events, generating the dynamic functions relating to events is handled separately but similarly.)

5.2 Execution

The final step in `net.compile` is to call `net.exec_dynamic_functions`, which will execute all the dynamic function code into callable functions.

First `net.exec_dynamic_functions` checks a cache to see whether it really needs to re-create all the functions. If it does, it first constructs Python functions from all the code in `net._dynamic_funcs_python`. To construct the functions, we `exec` the code for each function and attach the resulting object to the `Network` `self`. Note that these are functions, not methods; they don't take an implicit `self` argument and can't access attributes of the `Network` they are attached to. This is primarily for consistency with the C versions, which cannot be sensibly made into methods.

One additional wrinkle exists for large models. It turns out that Python has an in-built limit to the complexity of a module it can `import` (or, equivalently, string it can `exec`). The resulting error is `SystemError: com_backpatch: offset too large`. Including logical control statements (`if`, `for`, `while`) in our dynamic functions is thus dangerous [3]. This is why the current versions generates individual functions for $\partial y/\partial p$ rather than using a large `if` statement as in previous versions.

To construct C functions `net.exec_dynamic_functions` writes `.c` and `.pyf` files containing the code and interface for all the C dynamic functions. To help ensure uniqueness the base filenames are assigned based on current system time and the MPI rank of the current node. These files are compiled into an python module by spawning an external f2py process. The resulting module is imported and stored in the cache, and the dynamic functions in that module are assigned to the appropriate attributes of the `Network`, overwriting the Python versions. (Generating the module with a unique name circumvents the fact that C extension modules cannot be reloaded. If a network is changed many times during program execution, however, the import of all these modules may cause excessive memory usage, as the garbage collector cannot free unused imported C modules.)

In general the C code for a dynamic function is a straightforward translation of the Python code. One important difference is that functions that are passed to DASKR should take in the full argument list expected by DASKR, even if the f2py wrapper hides some

of them. By passing a `._cpointer` from the function to DASKR we can then get direct C to Fortran communication, avoiding any Python-induced overhead. The other subtlety is that ostensibly two-dimensional arrays are passed in from Fortran functions as flat one-dimensional arrays, so indexing is more complicated. (One can cast a one-dimensional array to a two-dimensional array in C via `double (*arr2d)[N] = (double(*)[N])(&arr1d);`. In testing this seemed to cause problems when interfacing with DASKR.)

6 Sensitivity Integration

Much of our research revolves around how changes in parameter values affect the dynamics $y(t;p)$ of a network, thus we are often interested in the *sensitivities* $\frac{dy(t;p)}{dp}$ of those dynamics. Such sensitivities can be obtained via finite-difference derivatives as

$$\frac{dy(t;p)}{dp} = \frac{y(t;p+\Delta p) - y(t;p)}{\Delta p}. \tag{1}$$

This procedure is, however, not very well-behaved numerically. We can do much better using SloppyCell's ability to take analytical derivatives of Python math expressions.

In a normal integration, we're evaluating:

$$y(t;p) = \int_0^t \frac{dy}{dt'}\, dt'. \tag{2}$$

If we take d/dp of this equation, we obtain

$$dy(t;p)/dp = \int_0^t \left[\frac{\partial}{\partial p}\frac{dy}{dt'} + \sum_{y'} \frac{\partial}{\partial y'}\frac{dy}{dt'}\frac{dy'}{dp} \right] dt'. \tag{3}$$

Essential to this procedure is the fact that analytic Python math expressions for $\frac{\partial}{\partial p}\frac{dy}{dt'}$ and $\frac{\partial}{\partial y'}\frac{dy}{dt'}$ can be obtained using the analytic differentiation capabilities of the `ExprManip` module (Section 4). This set of equations must be integrated simultaneously with normal right-hand-side (Equation 2), so our system now has twice as many equations. This does slow down the integration somewhat, but our experience suggests that calculating sensitivities this way is not much slower than calculating them via finite differences and is much better behaved.

In SloppyCell, the right-hand-side function for the sensitivity integration for a `Network` object is `net.sens_rhs`. The optimizable variable to return derivatives with respect to is specified by the last entry in the `constants` argument to `sens_rhs`.

Warning: As of 9/17/2007, we're using DDASKR to calculate initial conditions for the sensitivity integration. One side effect of this is that, at the initial timepoint or when an event fires, $\frac{d}{dt}\frac{dy}{dp}$ is quite probably wrong. We don't currently use this quantity for anything, but if we do, this will need to be fixed.

6.1 Handling Events

Perhaps the trickiest part of the sensitivity integration is dealing correctly with the SBML event model. The SBML event model is relatively complex and perhaps not intuitive. An event *fires* when the function defining its triggering function T transitions from False to True. At that *firing time* t_f new values are calculated for all variables with event assignments. The effects of the event may *delayed* by some time t_d which may be a function D of the variables at the firing time. The event thus *executes* at a time $t_e = t_f + t_d$, and the values calculated when the event fired are assigned to the appropriate variables.

6.1.1 Time Sensitivities

First we calculate the derivative of the event firing time with respect to p. The event fires when $\mathrm{T}(y(t_f, p), p, t_f) = 0$, and taking the derivative yields:

$$\frac{d}{dp}\,\mathrm{T}(y(t_f,p),p,t_f) = \frac{\partial\,\mathrm{T}}{\partial p} + \sum_y \frac{\partial\,\mathrm{T}}{\partial y}\frac{dy}{dp} + \frac{\partial\,\mathrm{T}}{\partial t}\frac{dt_f}{dp} + \sum_y \frac{\partial\,\mathrm{T}}{\partial y}\frac{dy}{dt}\frac{dt_f}{dp} = 0. \tag{4}$$

Solving for $\frac{dt_f}{dp}$ we obtain

$$\frac{dt_f}{dp} = \frac{\frac{\partial\,\mathrm{T}}{\partial p} + \sum_y \frac{\partial\,\mathrm{T}}{\partial y}\frac{dy}{dp}}{\frac{\partial\,\mathrm{T}}{dt} + \sum_y \frac{\partial\,\mathrm{T}}{\partial y}\frac{dy}{dt}}. \tag{5}$$

All quantities in this derivative are, of course, evaluated at the time the event fires. One subtlety with $\frac{dt_f}{dp}$ is that events may be *chained*; the execution of one event may cause the firing of another event. In that case $\frac{dt_f}{dp}$ of the fired event is equal to $\frac{dt_e}{dp}$ of the event whose execution caused the current event to fire.

Next we need the derivative of the delay time t_d which may be calculated by a function $\mathrm{D}(y(t_f), p, t_f)$. This is straightforward to calculate as:

$$\frac{dt_d}{dp} = \frac{\partial\,\mathrm{D}}{\partial p} + \sum_y \frac{\partial\,\mathrm{D}}{\partial y}\frac{dy}{dp} + \frac{\partial\,\mathrm{D}}{\partial t}\frac{dt_f}{dp} + \sum_y \frac{\partial\,\mathrm{D}}{\partial y}\frac{dy}{dt}\frac{dt_f}{dp}, \tag{6}$$

where again all variables are evaluated at the firing time.

Finally, the sensitivity of the event execution time is just

$$\frac{dt_e}{dp} = \frac{dt_f}{dp} + \frac{dt_d}{dp}. \tag{7}$$

6.1.2 Variable Sensitivities

Now let us calculate the sensitivities of variables after event execution. First consider a variable y whose value is not changed by the event. Note that $\frac{dy}{dt}$ may be different before

and after the event executes because of changes to other variables. Then the perturbation to its sensitivity $\frac{dy}{dp}$ is given by

$$\left.\frac{dy}{dp}\right|_{>t_e} = \left.\frac{dy}{dp}\right|_{<t_e} + \left.\frac{dy}{dt}\right|_{<t_e} \frac{dt_e}{dp} + \left.\frac{dy}{dt}\right|_{>t_e} \frac{dt_e}{dp}, \tag{8}$$

where $|_{<t_e}$ denotes values prior to event execution and $|_{>t_e}$ denotes values after event execution. The additional terms involving $\frac{dy}{dt}$ can be thought of as changes in y that do or do not happen because of the shift execution time.

Now let y be a variable whose value is changed by the event, as determined by the function A. The sensitivity of y after event execution is:

$$\left.\frac{dy}{dp}\right|_{>t_e} = \left.\frac{d\,\mathrm{A}}{dp}\right|_{t_f} - \left.\frac{dy}{dt}\right|_{>t_e} \frac{dt_e}{dp}, \tag{9}$$

where $|_{t_f}$ is a reminder of the fact that A is calculated at the firing time, so only the variable values at that time can matter for that term. The sensitivity of the assigned value is

$$\frac{d\,\mathrm{A}}{dp} = \frac{\partial\,\mathrm{A}}{\partial p} + \sum_y \frac{\partial\,\mathrm{A}}{\partial y}\frac{dy}{dp} + \frac{\partial\,\mathrm{A}}{\partial t}\frac{dt_f}{dp} + \sum_y \frac{\partial\,\mathrm{A}}{\partial y}\frac{dy}{dt}\frac{dt_f}{dp}, \tag{10}$$

where all variables are evaluated at the time the event fires.

6.1.3 Implementation

To calculate all the sensitivities, we need $\left.\frac{dy}{dt}\right|_{t_f}$, $\left.\frac{dy}{dt}\right|_{<t_e}$, $\left.\frac{dy}{dt}\right|_{>t_e}$, $\left.\frac{dy}{dp}\right|_{t_f}$, and $\left.\frac{dy}{dp}\right|_{<t_e}$. In particular, note that we need $\left.\frac{dy}{dt}\right|_{>t_e}$ which is a quantity only available *after* the event executes. To deal with this, each sensitivity integration begins with a normal integration of the network. The resulting trajectory stores a list of `event_info` objects under `traj.events_occurred`, and the relevant quantities are available as attributes of these objects. For example, if `e` is an `event_info` object, then `e.yp_pre_exec` holds $\left.\frac{dy}{dt}\right|_{<t_e}$.

During sensitivity integration a copy of normal trajectory's `events_occurred` list is made, and the `event_info` objects are updated with the necessary sensitivity information (e.g. $\left.\frac{dy}{dp}\right|_{t_f}$) as the network is integrated. All computations related to events and sensitivities outlined above are performed in `Network_mod.executeEventAndUpdateSens`. Chained events are handled by storing a reference to the `event_info` object prior to the current event in the chain.

6.2 Adjoint Method

The above sensitivity analysis requires solving $2N_e$ equations for each parameter, where N_e is the number of equations in a normal integration of the model. For calculating a Jacobian this many integrations is probably be unavoidable. In optimization, however, we're often

interested in the gradient of a single function of the model variables, the cost. So-called 'adjoint' sensitivity methods are designed for just this case, where we're interested in the derivative of one or a few quantities with respect to many variables [4, 5]. Essentially, an adjoint calculation involves two integrations, one forward and one backward, of an augmented systems of equations. Importantly, the size of this augmented system does not depend on the number of parameters one is considering.

An early version of SloppyCell included adjoint calculation of the gradient of the cost. Performance was somewhat disappointing; the method was not faster than calculating the forward sensitivities as above. This was not, however, a failure of the method itself, but rather our implementation. As mentioned above, the adjoint method requires an integration backwards in time which must refer to values calculated on the forward integration. To access those values we used the SciPy's spline interpolation routines, and it was calls to these routines that killed performance. An implementation that had direct access to the forward integration's approximation to the trajectory could be much faster.

Additionally, we did not work out how to propagate the adjoint system across events. As seen above, this can be quite complicated even in forward sensitivity analysis.

7 Parallel Execution

For communication between nodes in a parallel job, SloppyCell uses `PyPar`, a relatively Pythonic wrapper for MPI. To minimize code complexity (particularly in exception handling), SloppyCell uses a simple master/slave architecture enabled by Python's object serialization and dynamic code execution capabilities [6].

Upon import of `RunInParallel.py`, all nodes with `rank != 0` enter a `while` loop and wait for commands. The master node sends commands of the form `(command, args)`. `command` is a string specifying the command to execute, while `args` is a dictionary mapping names of arguments to the appropriate objects. The slave `eval`s the command and sends the return value back to the master. If an exception is raised during the function evaluation, the slave instead sends back the exception object. This architecture ensures that the execution path is very simple on the slaves, minimizing difficulties in synchronization. It does markedly increase the communication overhead, but many of our common calculations can be parallelized into large, coarse operations.

The arguments and return values sent between nodes can be any Python object that can be "pickled", serialized into a string representation. This requires some finesse for our `Network` objects, as dynamically generated functions cannot themselves be pickled. When pickled, an object returns its state as a dictionary from `self.__getstate__`. When unpickled that dictionary is used by `self.__setstate__` to restore the object. To enable pickling (and copying) of `Network` objects, they overload the default `__getstate__` and `__setstate__` methods. In `Network.__getstate__` all dynamic functions are removed from the return dictionary, and in `Network.__setstate__` `self.exec_dynamic_functions` is called to recreate them. (The code for the dynamic functions is pickled with the Network, so it need not be regenerated.)

One source of danger in the current implementation is that it will fail if the slaves call some function that is itself parallelized, as all parallelized functions assume they are being called on the master. Bypassing this limitation may be difficult, as messages can only be sent to workers who are in the 'receive' state. As more parts of the code our parallelized, we may need options to choose a which level parallelization happens.

References

[1] Brown PN, Hindmarsh AC, Petzold LR (1994) Using Krylov methods in the solution of large-scale differential-algebraic systems. SIAM J Sci Comput 15:1467–1488. back

[2] Brown PN, Hindmarsh AC, Petzold LR (1998) Consistent initial condition calculation for differential-algebraic systems. SIAM J Sci Comput 19:1495–1512. back

[3] Reedy T (2006). Re: Too many if statements? comp.lang.python newsgroup, Feb 10. back

[4] Errico RM (1997) What is an adjoint model? B Am Meteorol Soc 78:2577–2591. back

[5] Cao Y, Li S, Petzold L, Serban R (2002) Adjoint sensitivity analysis for differential-algebraic equations: the adjoint DAE system and its numerical solution. SIAM J Sci Comput 24:1076–1089. back

[6] Myers CR, Gutenkunst RN, Sethna JP (2007) Python unleashed on systems biology. Comput Sci Eng 9:34–37. arXiv:0704.3259. back