A CmePy Tutorial

Reuben Fletcher-Costin

October 30, 2009

Contents

1 Overview

`CmePy` is a `Python` package to assist in the numerical solution of the Chemical Master Equation (CME).

2 Installation

2.1 Dependencies

`CmePy` was developed with `Python` 2.5.2, `SciPy` 0.7.0, & `NumPy` 1.2.1 . In addition, `matplotlib` is used to display graphical output in some examples, but is not otherwise a dependency.

2.2 Testing & Installation

Once `CmePy` has been obtained, the package can be tested and installed by running the `setup.py` script via `Python` as follows:

```
python setup.py test
python setup.py install
```

2.2.1 Local package installation (Linux)

If it is not possible to install `CmePy` to the `Python`'s global `site-packages` directory, `CmePy` may be installed locally to a user's `${HOME}` directory. First, ensure the path `${HOME}/lib/python` exists, and that the `${PYTHONPATH}` environment variable includes `${HOME}/lib/python`. Then, `CmePy` may be installed locally via

```
python setup.py install --home=${HOME}
```

A (possibly better) alternative to this method of local installation is to use the `virtualenv` package to establish isolated (local) `Python` environments. See `http://pypi.python.org/pypi/virtualenv`.

3 Examples

3.1 Basic `CmeSolver` usage

We begin with a simple example illustrating the basic usage of `CmePy`'s `CmeSolver` class. The code for this example is shown below in Listing 1.

Listing 1: Example: solving a model

```
import numpy
from cmepy.solver import CmeSolver
from cmepy.models import MM_simple as model

solver = CmeSolver(model)

time_steps = numpy.linspace(0, 2.5, 51)
for t in time_steps:
    solver.step(t)

solution = solver.get_p()
```

In this first example, we load `MM_simple`, one of the pre-defined models supplied with `CmePy`. This model is for a simple Michaelis-Menten system, consisting of three reactions. The model definition format will be explained later, in subsection 3.4.

After loading the model for the Michaelis-Menten system under the name `model`, we create a `CmeSolver` instance for this model, then step the solver forward, advancing the solution of the chemical master equation, from time $t = 0.0$ to time $t = 2.5$ seconds. Finally, we extract p, the (reaction-count) probability distribution for $t = 2.5$, and store it under the name `solution`. Since the system we are modelling has three reactions, the resulting probability distribution returned by `solver.get_p()` is a three-dimensional `NumPy` array.

3.2 Combining `CmeSolver`, `CmeRecorder` & `PyLab` for graphical output

We can extend the previous example to record and plot the expected species counts. `CmePy` includes a convenience class, `CmeRecorder`, which is able to compute and store a number of common measurements. In this example, shown in Listing 2, we use `CmeRecorder` to compute the expected count of each species in the system, at each time step, and then plot these measurements using `PyLab`. The resulting plot is displayed in Figure 1

Listing 2: Example: solving a model and plotting species counts

```
import numpy
import pylab
from cmepy.solver import CmeSolver
from cmepy.recorder import CmeRecorder
from cmepy.models import MM_simple as model

solver = CmeSolver(model)

recorder = CmeRecorder(solver)
recorder.add_target(output = ['expectation'],
                    species = model['species'])

time_steps = numpy.linspace(0, 2.5, 51)
for t in time_steps:
    solver.step(t)
    recorder.take_measurements()

pylab.figure()
for s_info in recorder.measurements('species'):
    pylab.plot(s_info.times,
               s_info.expectation,
               label = s_info.name)
```

```
pylab.legend()
pylab.title('Expectation values of species counts')
pylab.show()
```

3.3 Setting `CmeSolver` initial conditions

By default, when a `CmeSolver` instance is initialised for a model, default values for the initial time t_0 and initial probability distribution p_0 are used. The default initial time is 0, while the default initial probability distribution is probability 1 at the origin of the reaction count state space – that is, the state where all reaction counts are zero – and probability 0 for all other states.

However, it is possible to explicitly pass initial conditions to the solver, by calling the method `set_initial_values(t0, p0)`. Here, `t0` specifies the time of the initial probability distribution, while `p0` specified the initial probability distribution. `p0` must be a `NumPy` array, of the shape `model['np']`. Since `p0` is a probability distribution, all elements of the array should be non-negative, and sum to unity, although these are not hard-requirements [1] .

An example of specifying custom initial conditions is given below in Listing 3. Here, the initial probability distribution is set to be uniformly zero, except for the states where the count of the first reaction is 0 or 1, and the count of the second reaction is 0, where the probability is set to 0.5. The initial time is specified to be 10.0 seconds.

Listing 3: Example: specifying initial conditions

```
import numpy
from cmepy.solver import CmeSolver
from cmepy.models import A2B2C as model

solver = CmeSolver(model)

#compute & set custom initial conditions for solver
p_0 = numpy.zeros(models['np'])
p_0[0, 0] = 0.5
p_0[1, 0] = 0.5

time_steps = numpy.linspace(10.0, 20.0, 11)
t_0 = time_steps[0]

solver.set_initial_values(t_0, p_0)

#solve the model
for t in time_steps:
        solver.step(t)
```

3.4 Defining new models

It is reasonably straight-forward to define new models. We will introduce the model format, and some useful techniques, by considering a sequence of examples. To begin, we start with a very simple system, consisting of two species, A and B, and a single reaction $A \to B$. We assume that the initial conditions consist of 10 copies of the species A, and 0 copies of the species B. A minimal model for this system is defined below in Listing 4.

Listing 4: A minimal model definition for the system $A \to B$.

```
model = {'propensities' : (lambda x : 1.0*(10-x), ),
         'np' : (11, ),
         }
```

As can be seen in the example, models are simply defined as `Python` dictionaries. The minimal [2] information a model must supply are the fields `propensities` and `np`.

[1] In fact, due to the nature of the numerical computations, probability distributions *returned* by the solver typically contain some (very small) non-positive elements.

[2] Here, we mean *minimal* in the sense that the presence of both these fields is both necessary and sufficient for this model to be solvable via a `cmepy.solver.CmeSolver` instance.

The field `propensities` should be a sequence (i.e. tuple, list, etc.) of propensity functions, one for each reaction in the system, mapping reaction counts to reaction propensities.

In this example we have used `Python`'s `lambda` shorthand to succinctly define the propensity function as an unnamed function, mapping `x`, the number of times the reaction has occurred, to the propensity of the reaction, `1.0 * (10-x)`. Observe that if x is the number of times that the reaction $A \to B$ has occurred, and that initially there are 10 copies of species A, then there must currently be $10 - x$ copies of species A.

The field `np` is a sequence of positive integers, specifying the dimensions of the reaction count state space. In this example, there is only one reaction in the system, hence `np` only contains one element, and the corresponding reaction count may range over the 11 possible values: $0, 1, 2, 3, \ldots, 10$.

For the second example, we extend the minimal model from Listing 4 to define a full model for the same system, shown below in Listing 5.

Listing 5: A full model definition for the system $A \to B$.

```
model = {'doc' : 'example model for reaction A -> B',
         'propensities' : (lambda x : 1.0*(10-x), ),
         'reactions' : ('A->B', ),
         'species counts' : (lambda x : 10-x,
                             lambda x : x, ),
         'species' : ('A', 'B', ),
         'np' : (11, ),
         }
```

The `propensities` and `np` fields remain unchanged, but there are a number of additional fields:

- `doc` : a brief description of the model;
- `reactions` : a sequence of reaction names, using the same ordering as the sequence of propensity functions;
- `species counts` : a sequence of functions, each mapping reaction counts to a species count, for each species in the model
- `species` : a sequence of species names, using the same ordering as the sequence of species count functions.

While a system consisting of only a single reaction is a good introduction to the model format, most systems of interest will feature two or more reactions. For the next example, we shall consider a system of two reactions, defined below in Listing 6.

Listing 6: A minimal model definition for the system $A \to B$, $B \to C$.

```
model = {'propensities' : (lambda x1, x2 : 1.0*(10-x1),
                           lambda x1, x2 : 1.0*(x1-x2),),
         'np' : (11, 11, ),
         }
```

As the system consists of two reactions, both the `propensities` and `np` fields now contain two elements. Observe that each propensity function must now take two arguments, `x1` and `x2`, which are the reaction counts of the first and second reactions respectively. Apart from the addition of a second unused argument, the first propensity function has the same form as that defined in the first example, Listing 4.

Note, in the second propensity function, that the difference `x1 - x2` is the count of reaction $A \to B$ minus the count of reaction $B \to C$, so this yields the number of copies of the species B.

As before, we may expand this minimal two-reaction model into a full model, as shown in Listing 7.

Listing 7: A full model definition for the system $A \to B$, $B \to C$.

```
model = {'doc' : 'example model for reactions A -> B, B -> C',
         'propensities' : (lambda x1, x2 : 1.0*(10-x1),
                           lambda x1, x2 : 1.0*(x1-x2), ),
         'reactions' : ('A->B', 'B->C', ),
         'species counts' : (lambda x1, x2 : 10-x1,
                             lambda x1, x2 : x1-x2,
                              lambda x1, x2 : x2, ),
         'species' : ('A', 'B', 'C', ),
         'np' : (11, 11, ),
         }
```

For a final example, illustrating some useful techniques, consider the model below in Listing 8.

Listing 8: Automatic generation of models for the sustem $A \to B$, $B \to C$, using specified initial copy counts of the species A, B and C.

```
def generate_model(initial_count_a,
                   initial_count_b,
                   initial_count_c):
    """
    Generates and returns a complete model for the system of reactions
    A->B, B->C, using the specified initial counts of A, B & C.
    """

    # define maps from reaction counts to species counts first
    species_counts = (lambda *x : initial_count_a - x[0],
                      lambda *x : initial_count_b + x[0] - x[1],
                      lambda *x : initial_count_c + x[1], )

    # then define the reaction propensities in terms of the species counts
    propensities = (lambda *x : 1.0*species_counts[0](*x),
                    lambda *x : 1.0*species_counts[1](*x), )

    model = {'doc' : 'example model for reactions A -> B, B -> C',
             'propensities' : propensities,
             'reactions' : ('A->B', 'B->C', ),
             'species counts' : species_counts,
             'species' : ('A', 'B', 'C', ),
             'np' : (initial_count_a + 1,
                     initial_count_a + initial_count_b + 1, ),
             }

    return model

model = generate_model(10, 0, 0)
```

Listing 8 is a re-formulation of the same model from Listing 7, employing a number of techniques:

- Instead of hard-wiring the initial copy counts of species A, B and C into the model itself, a function `generate_model` has been defined, which accepts the initial species counts as arguments, and returns the corresponding model.
- Since the reaction propensities depend upon the species counts, the species count functions have been defined first, which are then used in turn during the definition of the reaction propensity functions.
- Lastly, instead of explicitly specifying the reaction count arguments for the species count and propensity functions, we use `Python`'s variable argument list syntax to pack the arguments to each function into a list `x` of reaction count arguments. Each reaction count is then addressed as an element of the list, that is, `x[0]` for the first reaction count, `x[1]` for the second reaction count.

3.5 Further Examples

A number of example scripts are distributed with `CmePy` inside the `examples` directory. The example model definitions, as discussed in the previous subsection, are included in `model_example_1.py` and `model_example_2.py` , with the latter script also containing a unit test to verify the equivalence of a number of different formulations of the one model.

Various examples of `CmeSolver` and `CmeRecorder` usage, including graphical output via `matplotlib`, are given in the scripts `example_1.py`, `recorder_example_1.py`, `solver_example_1.py`, `solver_example_2.py` and solver_example_3.py . Lastly, catalytic_example.py demonstrates the solution of a catalytic reaction and displays a table of computed expected species counts.

4 Some common errors, & possible solutions

4.1 Propensity functions not defined for vector arguments

Each propensity function defined in the 'propensities' entry of a model definition must handle the case where the reaction count arguments are not integers, but NumPy arrays of integers. Consider the following incorrectly-defined model shown in Listing 9:

Listing 9: A broken model

```
model = {'doc' : 'a broken model, unable to handle vector arguments',
         'np' : (10, ),
         'propensities' : (lambda *x : max(x[0], 0.0),)
         }
```

Attempting to solve this model using a CmeSolver instance will raise a rather cryptic exception during the call to solve:

```
Traceback (most recent call last):

[ some output omitted here . . . ]

  File "solver_failure_example.py", line 12, in <lambda>
    'propensities' : (lambda *x : max(x[0], 0.0),)
ValueError: The truth value of an array with more than one element
is ambiguous. Use a.any() or a.all()
```

This error is due to the use of Python's built-in max function, which is unable to perform a vectorised element-wise maximum operation when one of the arguments, in this case x[0], is a NumPy array. The correct solution is to replace the use of max in the definition of the propensity function with a call to numpy.maximum, which performs a vectorised maximum operation if passed array arguments, and reduces to the usual scalar max behaviour otherwise. A corrected model is show below in Listing 10:

Listing 10: A corrected model

```
import numpy
model = {'doc' : 'corrected model',
         'np' : (10, ),
         'propensities' : (lambda *x : numpy.maximum(x[0], 0.0),)
         }
```

4.2 Integration failure due to large time steps

Symptom: a CmeSolver instance raises a RuntimeError during a call to step, with similar output to the following:

```
 DVODE--  At current T (=R1), MXSTEP (=I1) steps
       taken on this call before reaching TOUT
      In above message,  I1 =       500
      In above message,  R1 =  0.5801764171096D+00
vode: Excess work done on this call. (Perhaps wrong MF.)
Traceback (most recent call last):

[ some output omitted here . . . ]

RuntimeError: CME integration failure (look for messages from DVODE / vode)
```

Possible Solution: attempt reducing the size of the time steps. For very stiff models, this might involve using very small time steps.

4.3 Integration failure due to singular propensity functions

Symptom: a `CmeSolver` instance raises a RuntimeError during a call to `step`, with similar output to the following:

```
 DVODE--  At T (=R1) and step size H (=R2), the
       corrector convergence failed repeatedly
       or with abs(H) = HMIN
      In above,  R1 =  0.0000000000000D+00   R2 =  0.9965007081277D-17
vode: Repeated convergence failures. (Perhaps bad Jacobian supplied or wrong
choice of MF or tolerances.)
Traceback (most recent call last):

[ some output omitted here . . . ]

RuntimeError: CME integration failure (look for messages from DVODE / vode)
```

Possible Solution: replace singular propensity functions with non-singular ones!

4.4 State space is too large

Interesting things can occur if the size of a model's state space, `model['np']`, is too large. One of the following errors may be raised:

```
ValueError: dimensions too large.

MemoryError

ValueError: negative dimensions are not allowed
```

This last error is likely due to integer overflow while attempting to compute the number of states in the state space.

Finally, if the size of the state space causes the machine to run out of memory during a call to `SciPy`'s underlying 'vode' ode integrator implementation, this may cause a segmentation fault, which can be difficult to diagnose, as it will not raise an exception in `Python`.

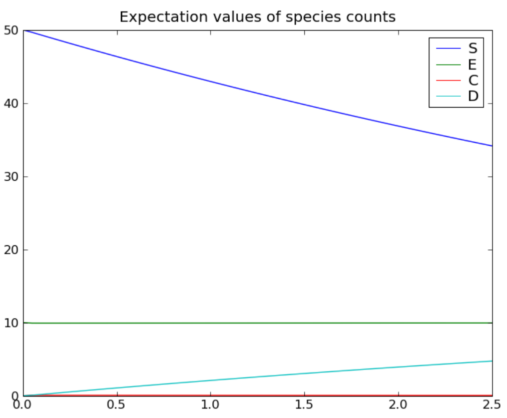

Figure 1: Output produced by Listing 2.