systems was the suite of routines in ODEPACK. Of particular note was LSODA which in the 1990s was very popular and is still a valuable set of software (currently used in COPASI).[2] The original stiff differential equation solver was developed by Gear [51] in the 1970s and is still used in Matlab in the form of ode15s.

5.3 Matlab Solvers

Although this isn't a book about Matlab, it is worth mentioning how Matlab can be used to solve differential equations. Matlab offers a range of solvers with the two most commonly used being `ode45` and `ode15s`.

The `ode45` solver implements a variable step size Runge-Kutta method by using the Dormand-Prince method. The basic syntax for `ode45` is:

```
[t,y] = ode45(@myModel, [t0, tend], yo, [], p);
```

where

`myModel` is the function containing the differential equations.

`t0, tend` are the initial and final values for the independent variable, t.

`yo` is a vector of initial conditions.

`p` is the set of parameters for the model, and can be any size.

The empty vector in the call is where additional options can be placed.

For example, to solve the set of ODEs:

$$\frac{dy_1}{dt} = v_o - k_1 y_1$$

$$\frac{dy_2}{dt} = k_1 y_1 - k_2 y_2$$

We would write the following .m file and load it into Matlab:

```
function dy = myModel(t, y, p)
dy = zeros (2,1);
vo = p(1);
k1 = p(2);
k2 = p(3);
dy(1) = vo - k1*y(1);
dy(2) = k1*y(1) - k2*y(2);
```

We would then call the solver as follows:

[2] In some of our own work, we have noticed that LSODA can be much faster then CVODE.

```
p = [10, 0.5, 0.35]
y0 = [0, 0]
[t, y] = ode45 (@myModel, [0, 20], y0, [], p)
```

Although many problems can be solved using `ode45`, some stiff models will fail to give the correct solution using this method. In these cases `ode15s` is recommended. `ode15s` is a variable order solver and uses the well known Gear method [51]. Like `ode45`, `ode15s` is also a variable step size method. `ode45` might be faster than `ode15s` on simple problems, but with today's fast computers the difference is not great. Therefore `ode15s` is recommended for all problems unless computing time is critical.

5.4 Python Solvers

Like Matlab, Python is a general purpose computing language. However, unlike Matlab, Python is open source and freely available for anyone to use. Python offers a variety of ODE solvers via the scipy package[3]. These include LSODA [67], an implicit Adams method [59] (for non-stiff systems), 4th order adaptive step size Dormand-Prince and an eight order adaptive step size Dormand-Prince [35]. The code below shows the Matlab code shown in the previous section expressed using Python. The example uses the default LSODA integrator.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

vo = 10
k1 = 0.5
k2 = 0.35

# Declare the model
def myModel(y, t):

   dy0 = vo - k1*y[0]
   dy1 = k1*y[0] - k2*y[1]
   return [dy0, dy1]

time = linspace(0.0, 20.0, 100)
yinit = array([0.0, 0.0])
y = odeint (myModel, yinit, time)
```

[3] http://scipy.org/

```
plt.plot(time, y[:,0], time, y[:,1]) # y[:,0] is the first column of y
plt.xlabel('t')
plt.ylabel('y')
plt.show()
```

5.5 Other Software

Matlab and Python aren't the only software that can be used to solved differential equations. Mathematica is an example of commercial tool that can be used to solve differential equations.

For those who require more control or who are unable to purchase a commercial tool, there are many free applications and professionally developed open source software libraries that can be used very effectively. Octave (http://www.gnu.org/software/octave/) is an open source tool that is very similar to Matlab. SciLab (http://www.scilab.org/) is another free Matlab like application. If you like programming in Python then Sage (http://www.sagemath.org/index.html) is a good option. There are therefore many alternative and free options to using Matlab.

For those who require much more control and higher performance, it is possible to write your own code around the SUNDIALS C/C++ library which is available under the unrestricted BSD open source licence. Within SUNDIALS is the CVODE library used by many of the commercial tools. CVODE implements an advanced Gear like algorithm using a variable order and variable step size approach. It is well suited for stiff systems and is the preferred method for those who need to write their own code. One final library worth mentioning is the GPL (GNU General Public License) licensed GSL library (http://www.gnu.org/software/gsl/. Although very comprehensive, the GPL license unfortunately puts critical restrictions on how the library can be used. Unless one has a real need to use the GSL library, it is recommend that one employ the unrestricted SUNDIALS suite.

Specialized Software

Simulating biochemical networks has a long history dating back to the 1940s [25]. The earliest simulations relied on building either mechanical or electrical analogs of biochemical networks. It was only in the late 1950s, with the advent of digital computers and the development of specialized software tools [47], that the ability to simulate biochemical networks became more widely available. In the intervening years up to 1980, a handful of other software applications were developed [20, 21, 126] to help the small community of modelers. In more recent years, particularly since the early 1990s, there has been a significant increase in interest in modeling biochemical processes and a wider range of tools is now available to the budding systems biologist. Many open source tools have been developed by practicing

scientists and are therefore freely available.

In this book we will be using the author's modeling tool Tellurium [150]. Tellurium is well suited for our purpose. It is a script based modeling application which makes it easy to illustrate a modeling exercise.

Many tools do not offer readable text based renderings of models because they use either a visual approach to modeling, such as JDesigner [12] or CellDesigner [84], or have a graphical user interface such as COPASI [71] or iBiosim [117]. All these tools export and import the standard modeling language SBML (See section G.1). However, because SBML is written in XML, it is also difficult to display a model using SBML in a textbook.

Tellurium

Tellurium [150] is an integrated Python based environment for modeling in systems biology. The current version (July 2014) integrates a number of libraries including libRoadRunner (Simulator), libSBML (SBML support), libAntimony (Antimony support) and SBML2-Matlab (SBML to Matlab converter). In addition Tellurium distributes a number of standard Python packages such as Matplotlib (plotting) and NumPy (array support). All packages are integrated using spyder2 (https://code.google.com/p/spyderlib/) which offers a Matlab like experience for modelers.

Visually, Tellurium has two main windows (Figure H.1): a console where commands can be issued and results returned, and an editor where control scripts and models are written. The application also has a plotting window which is used when graphing commands are issued.

Tellurium uses Antimony to let users describe biochemical pathways and Python coupled with libRoadRunner to do simulations and other analyses. Models can also be imported or exported as SBML. Many other capabilities are offered through libRoadRunner including support for metabolic control analysis, structural analysis of networks, and stochastic simulation. It has no explicit support for fitting as of yet. A more detailed description of Tellurium is given in Appendix H. The following code shows the model we used previously expressed using Tellurium:

```
import tellurium as te

rr = te.loada ('''
    $Xo -> y1; vo;
     y1 -> y2; k1*y1;
     y2 -> $waste; k2*y2;

    vo = 10; k1 = 0.5; k2 = 0.35;
    y1 = 0; y2 = 0;
''')
```

```
m = rr.simulate (0, 20, 100);
rr.plot (m);
```

The first part of the code shows the model expressed using Antimony and loaded into roadrunner while the second part show two commands to simulate and plot the results via libRoadRunner.

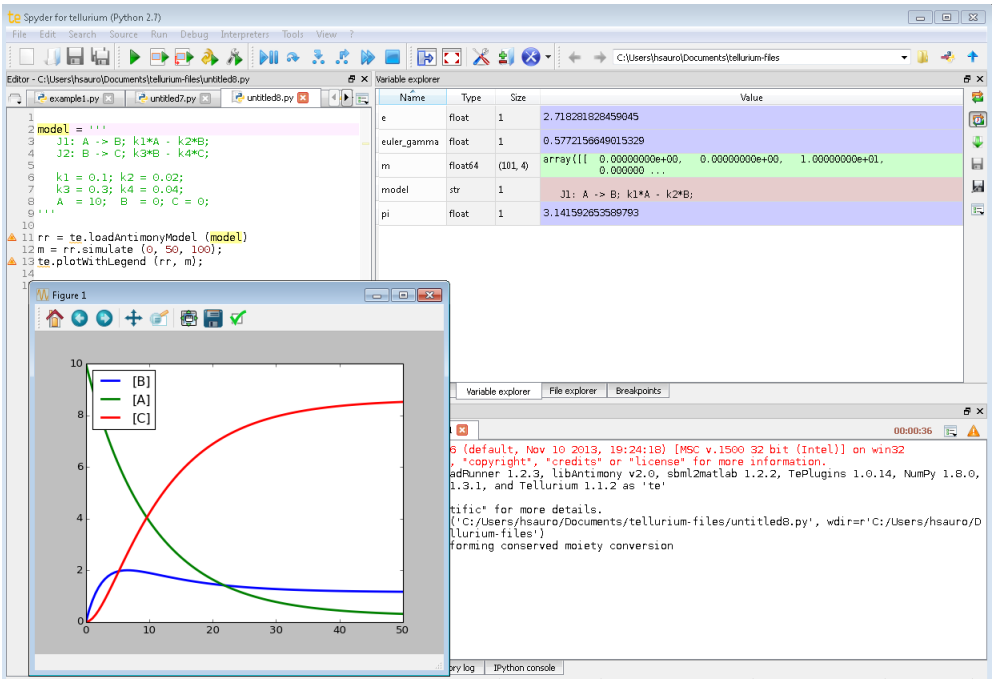

Figure 5.6 Screen shot of Tellurium with simulation results.

5.6 Moiety Conserved Cycles

Any chemical group that is preserved during a cyclic series of interconversions is called a conserved moiety (See section 3.8), Figure 5.7. Examples of conserved moiety subgroups include species such as phosphate, acyl, nucleoside groups, or covalently modifiable proteins. As a moiety gets redistributed through a network, the *total amount* of the moiety is constant and does not change during the time evolution of the system. For any particular subgroup, the total amount is determined solely by the initial conditions imposed on the model.

There are rare cases when a ‘conservation’ relationship arises out of a non-moiety cycle. This does not affect the mathematical analysis, but only the physical interpretation of the