PyDAS Documentation

Release 1.0.1

Reaction Mechanism Generator Team

August 19, 2015

1 Introduction **1**
1.1 Why PyDAS? . . . 1
1.2 Should I Use PyDAS? . . . 1

2 Installation **3**
2.1 Installing PyDAS binary package via Anaconda . . . 3
2.2 Installing PyDAS via Source . . . 3

3 Tutorial **7**

4 Using Cython with PyDAS **11**
4.1 The Problem . . . 11
4.2 Solution in Python . . . 11
4.3 Solution in Cython . . . 13

5 Using Observers or Listeners with PyDAS **17**
5.1 Creating a Listener Class . . . 17
5.2 Client Code . . . 17

Bibliography **19**

CHAPTER

ONE

INTRODUCTION

Differential equations are frequently encountered in a variety of technical fields. For example, scientists and engineers often express mathematical models of physical phenomena as a set of differential (and possibly some algebraic) equations. These models can quickly become complicated enough that direct analytical solutions are difficult or impossible to obtain. In these cases, computers can be used to numerically generate approximate solutions.

A system of differential algebraic equations (DAEs for short) is a set of equations that implicitly relate an independent variable t, a set of dependent variables $\mathbf{y}$, and the set of first derivatives $d\mathbf{y}/dt$. (Throughout this documentation, we use boldface to indicate a vector quantity.) The general form is

$$\mathbf{g}\left(t, \mathbf{y}, \frac{d\mathbf{y}}{dt}\right) = \mathbf{0}$$

Most DAE systems can be expressed in the above form using the appropriate transformations (e.g. finite differencing).

In 1982 the first version of DASSL, a Fortran code for efficient solving of DAE systems of the above form, was released by Petzold *[Petzold1982]*. In 1989 and 2002 the first versions of DASPK and DASKR, descendants of the DASSL code with a number of advanced features, were released by Brown, Hindmarsh, Petzold, and Ulrich.

1.1 Why PyDAS?

DASSL, DASPK, and DASKR are written in Fortran 77, which is not much fun to code in, especially for novice programmers. Even after the vast improvements in the language in Fortran 90/95, the task can seem daunting. (In particular, getting data into and out of a Fortran program via file input and output is still quite difficult and awkward.) However, the strength of Fortran is that it produces code that is very efficient to execute, which is often important when solving DAEs.

Meanwhile, the Python programming language is much easier to program in. In particular, Python comes with a large library of free, open source packages that provide a wide range of functionality, limiting the amount of work the programmer needs to do from scratch. A number of packages, including NumPy, SciPy, and matplotlib, replace much of the functionality of numerical computing environments such as MATLAB. However, the differential equation functionality within SciPy is insufficient for many complex DAE systems.

PyDAS provides Python programmers with access to a much more robust DAE solver by providing an interface to the DASSL code from Python.

1.2 Should I Use PyDAS?

If you have a small set of differential equations, you should consider trying the differential equation solver contained within SciPy first, as it may be suitable for your task. However, if you have a large set of differential (and algebraic) equations and/or know that you need a more robust solver, then PyDAS may be just the tool for you.

Read on to learn how to install and use PyDAS, and tips for getting the most out of PyDAS.

CHAPTER

TWO

INSTALLATION

2.1 Installing PyDAS binary package via Anaconda

Currently only available for Unix-based systems: Linux and MacOSX

If you wish to use PyDAS out of the box, you can now install it as a binary package using the Anaconda platform. PyDAS can now be installed in binary format using the Anaconda Python Platform. This is recommended for a basic user who will not be altering the source code of PyDAS or who is less familiar with Unix-based systems.

- Download and install the Anaconda Python Platform. When prompted to append Anaconda to your PATH, select or type Yes.
- Install PyDAS binary. Dependencies will be installed automatically. Type the following into your Terminal to do so

```
conda install -c rmg -y pydas
```

- You may now run a PyDAS test job. Save the Batch Reactor Reaction Series Script to a local directory. Use the Terminal to run the script inside that folder using the following command

```
python rxnSeries.py
```

You can now import any functions inside the PyDAS package directly from Python or your own scripts.

2.2 Installing PyDAS via Source

Installing PyDAS by compiling the source is recommended for developers and users who expect to modify the PyDAS code.

2.2.1 Prerequisites

PyDAS is currently available for the Python 2.x series. In particular, Python 2.5 and later are known to work. In addition to the packages in the Python standard library, PyDAS depends on the following packages:

- NumPy version 1.3.0 or later
- Cython version 0.12.1 or later

In addition, you will also need a Fortran compiler and a C compiler that produce object files that can interoperate [1]. The `gfortran` and `gcc` compilers from the GNU Compiler Collection are known to work, and are probably your

[1] The Fortran interfaces are exposed to Python via C, so the installer needs to be able to link object files from Fortran and C for this to work.

best bet no matter what operating system you are using. On Windows the MinGW compiler collection provides these compilers.

The DASSL and DASKR solvers are provided with the PyDAS package; you do not need to download them separately.

The DASPK3.1 solver is subject to copyright restrictions and therefore must be downloaded separately either using the `daspk31/download_daspk31.sh` script or by manually downloading the daspk31.tgz source code found on Linda Petzold's software page into the `daspk31` folder.

2.2.2 Compiling PyDAS

If you are running an operating system other than Windows, refer to the section directly below. Windows users get their own special installation procedure, described subsequently.

Unix-like Systems

A Makefile has been provided that can be used to compile all of the solvers – DASSL, DASPK3.1, and DASKR – and the PyDAS wrapper code. To use, invoke the following command from the root package directory:

```
$ make
```

This command will build PyDAS in-place, rather than installing it to your Python package directory. At this point you can optionally run the unit test script `test.py` and/or any of the provided examples to confirm that PyDAS was compiled successfully. The user will also be prompted to download and compile DASPK3.1. The DASPK3.1 solver and wrapper will be compiled automatically if the user agrees to the copyright restrictions.

If you wish to formally install PyDAS, run the following command from the root package directory after the `make` command (you may need root privileges for this):

```
$ make install
```

You may wish to write a file *make.inc* that sets certain variables used by the Makefiles (e.g. the Fortran compiler). An example of such a file, *make.inc.example*, has been provided.

We suggest running the unit tests to ensure the compile was successful by running the command:

```
$ make test
```

Windows

The easiest way to compile PyDAS on Windows is using the MinGW compiler in either cygwin or git bash. Simply use the commands:

```
$ mingw32-make
```

If errors arise in downloading the DASPK3.1 fortran code, you may need to download it manually into the `daspk31` folder from <http://www.cs.ucsb.edu/~cse/software.html>‘_ . Make sure to keep the daspk31.tgz file in the `daspk31` folder.

If you wish to formally install PyDAS, run the following command from the root package directory after the batch script completes successfully (you may need administrator privileges for this):

```
$ mingw32-make install
```

Warning: A regression in Cython 0.14 has resulted in it being unusable for PyDAS in Windows. Cython 0.13 is known to work, and more recent released of Cython should also correct for this regression. See this thread from the `cython-dev` mailing list for more information.

Alternatively, a batch script `make.bat` has been provided in the root package directory. Double-clicking this script will compile the DASSL solver into a static library and also compile the PyDAS wrapper code.

The batch script `make_daspk.bat` has been provided to compile the DASPK3.1 solver and the DASPK wrapper code. Make sure the source code for the DASPK3.1 have been manually downloaded and stored inside the daspk31 folder. See `daspk31/README` for details on where to download the files.

Note: The batch script presumes that you have the 32-bit version of the MinGW C and Fortran compilers installed. You may need to add the location of the MinGW `bin` directory to the `PATH` environment variable if this has not already been done.

At this point you can optionally run the unit test script `test.py` and/or any of the provided examples to confirm that PyDAS was compiled successfully.

Warning: When using MinGW with Cython on Windows, you may encounter the error "Unable to find vcvarsall.bat". A workaround for this issue is available from the Cython FAQ at this page. In particular the `pydistutils.cfg` file approach should work.

CHAPTER

THREE

TUTORIAL

In this section we will develop code to solve a very simple DAE system using DASSL.

The physical process we are modeling is a pair of first-order chemical reactions in series occurring in an ideal batch reactor:

$$\mathrm{A} \xrightarrow{k_1} \mathrm{B} \xrightarrow{k_2} \mathrm{C}$$

The governing equations for this process are the system of first-order ordinary differential equations

$$\begin{aligned}\frac{dA}{dt} &= -k_1 A \\ \frac{dB}{dt} &= k_1 A - k_2 B \\ \frac{dC}{dt} &= k_2 B\end{aligned}$$

where t is time, A, B, and C represent the concentration of each species, and k_1 and k_2 are rate coefficients for each chemical reaction. The initial condition is pure A, i.e.

$$A(0) = 1 \qquad B(0) = 0 \qquad C(0) = 0$$

For the purposes of this example we are not interested in units.

1. **Rewrite your DAE system in general form.** The general form of a DAE system is $\mathbf{g}(t, \mathbf{y}, d\mathbf{y}/dt) = \mathbf{0}$. Since our governing equations are explicit first-order ODEs, this is very easy:

$$\begin{aligned}-k_1 A - \frac{dA}{dt} &= 0 \\ k_1 A - k_2 B - \frac{dB}{dt} &= 0 \\ k_2 B - \frac{dC}{dt} &= 0\end{aligned}$$

   Most DAE systems should be expressible in general form, though some manual transformations may be required.

2. **Import the solver class from the `pydas` module.** This is done with a single import statement:

```
from pydas import DASSL
```

3. **Create a new class that derives from the chosen solver class.** The use of a class for this purpose enables you to store variables needed by the residual and/or Jacobian functions as attributes of the class. In this example we will need to store the rate coefficients k_1 and k_2. Here we provide an `__init__()` method to easily set these values. A docstring describing the purpose of the class is always a good idea.

```
class Model(DASSL):
    """
    A model of first-order irreversible reactions in series

        A -> B -> C

    occurring in a batch reactor. In such a system the concentration
    of the intermediate B has a maximum that depends on the relative
    rate constants `k1` and `k2`. These are stored as data members of
    the class so that they are available to the residual function.
    """

    def __init__(self, k1=0.0, k2=0.0):
        self.k1 = k1
        self.k2 = k2
```

4. **Write the residual method for that class.** The `residual()` method is automatically called by the solver, and is used to compute the value of.

$$\mathbf{g}\left(t, \mathbf{y}, \frac{d\mathbf{y}}{dt}\right) = \mathbf{0}$$

The method provides three parameters in addition to the `self` object: the independent variable *t*, a float; the vector of dependent variables *y*, a numpy array of floats; and the first derivatives *dydt*, a numpy array of floats. The method expects two return variables: a numpy array of floats containing the value of $\mathbf{g}(t, \mathbf{y}, d\mathbf{y}/dt)$ corresponding to the input parameters, and a integer status flag, having a value of 0 if okay or -2 to terminate the simulation. Our `residual()` method reflects the general form of the DAE system:

```
def residual(self, t, y, dydt):
    delta = numpy.zeros(y.shape[0], numpy.float64)
    delta[0] = -self.k1 * y[0] - dydt[0]
    delta[1] =  self.k1 * y[0] - self.k2 * y[1] - dydt[1]
    delta[2] =  self.k2 * y[1] - dydt[2]
    return delta, 0
```

Note that you need an `import numpy` statement in your module for the above to run.

5. **(Optional) Write the jacobian method for that class.** The `jacobian()` method is used to specify the analytical Jacobian corresponding to the residual function. The Jacobian is the matrix of partial derivatives with elements given by

$$J_{mn} = \frac{\partial g_m}{\partial y_n} + C_J \frac{\partial g_m}{\partial (dy_n/dt)}$$

In addition to the same *t*, *y*, and *dydt* parameters as the `residual()` method, the `jacobian()` method also has a fourth parameter: a float *cj* to be used to scale the derivative components in the Jacobian matrix. For our system the Jacobian is not too difficult to generate:

```
def jacobian(self, t, y, dydt, cj):
    pd = -cj * numpy.identity(y.shape[0], numpy.float64)
    pd[0,0] += -self.k1
    pd[1,0] += self.k1
    pd[1,1] += -self.k2
    pd[2,1] += self.k2
    return pd
```

If not specified, the solver will evaluate an approximate Jacobian numerically. For many problems this is enough, which is good because it can be time-consuming to generate an analytical Jacobian. The use of analytical or numerical Jacobian is detected and configured automatically by PyDAS; all you need to do is to provide or omit the `jacobian()` method.

6. **Create an object of your derived class.** Since we designed our `__init__()` method to accept the rate coefficients, we can easily set them to the values we want. For this tutorial we will use $k_1 = 1.0$ and $k_2 = 0.25$.

```
model = Model(k1=1.0, k2=0.25)
```

7. **Initialize the model with the initial conditions and solver options.** The initial conditions follow directly from the equation above:

```
t0 = 0.0; y0 = numpy.array([1.0, 0.0, 0.0], numpy.float64)
```

Since we are using a DAE solver, we also need initial values for the first derivatives $d\mathbf{y}/dt$. These initial values must be *consistent* with those of t and $\mathbf{y}$. It is best if you can determine the initial values of $d\mathbf{y}/dt$ manually, as we can in this example:

```
dydt0 = - model.residual(t0, y0, numpy.zeros(3, numpy.float64))[0]
model.initialize(t0, y0, dydt0)
```

If you cannot do this manually (or simply have no idea how), you can simply omit it; DASSL will then try to estimate it for you. Note that this is not always successful.

```
model.initialize(t0, y0)
```

You can also use the `initialize()` method to specify absolute and relative tolerances for the solver to use, either as scalars or numpy arrays. Below we use scalars:

```
model.initialize(t0, y0, dydt0, atol=1e-16, rtol=1e-8)
```

Default values will be used if not specified.

The `initialize()` method of the solver class must always be called before attempting to integrate.

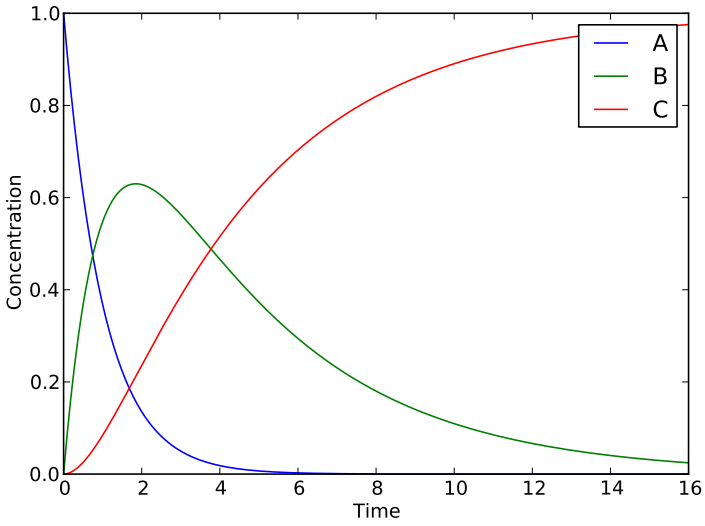
8. **Integrate forward using advance or step.** After initialization, the current values of t and $\mathbf{y}(t)$ are available from the *t* and *y* attributes of the solver object. In order to conduct the integration, call the `advance()` or `step()` methods of the solver object. The `advance()` method integrates forward until the time specified as the parameter is reached; the *t* and *y* attributes then will contain the solution at that time. The `step()` method integrates such that one automatically-determined step is taken towards the specified parameter, which is usually the end time of the simulation; when finished, the *t* and *y* attributes will contain the solution at the end time of that step. Here we will use the `step()` method:

```
# Initialize solution vectors
t = []
y = []

# Set maximum simulation time and maximum number of simulation steps to allow
tmax = 16
maxiter = 1000

# Generate the solution by stepping until tmax is reached
# This will give you the solution at time points automatically selected
# by the solver
iter = 0
while iter < maxiter and model.t < tmax:
    model.step(tmax)
    t.append(model.t)
    # You must make a copy of y because it is overwritten by DASSL at
    # each call to step()
    y.append(model.y.copy())

# Convert the solution vectors to numpy arrays
```

```
    t = numpy.array(t, numpy.float64)
    y = numpy.array(y, numpy.float64)
```

Note that we must keep track of the solution ourselves, as each call to `advance()` or `step()` causes the the *t* and *y* attributes to be overwritten. (In particular, we must copy *y* because it is a numpy array.) We must also provide the test that the end time of the simulation is reached, as this is not done for you. Finally, we convert the solution data to numpy arrays for easier postprocessing.

The complete code for this tutorial can be found in the `rxnSeries` DASSL example. Running this code should produce the following plot (assuming you have matplotlib installed):



CHAPTER

FOUR

USING CYTHON WITH PYDAS

The reactions in series example from the previous section was useful as an introductory tool. However, the model was very simple, so much that using PyDAS is almost certainly overkill. In this section we will develop and solve a more complicated problem. The problem is complex enough that solving it using pure Python code is time consuming, so we will show how to accelerate the solve using Cython (which you already have installed if you are using PyDAS).

4.1 The Problem

The problem of interest in this section is a classic problem from fluid dynamics/transport phenomena: diffusion through a one-dimensional thin membrane. The governing differential equation, called the *diffusion equation*, is (in dimensionless form)

$$\frac{\partial \theta}{\partial t} = \frac{\partial^2 \theta}{\partial x^2}$$

where $\theta(x, t)$ is the "concentration" of the diffusing substance, t is time, and x is position in the membrane. Since we are working in dimensionless units we will take the membrane width to be unity, giving us a domain of $0 \leq x \leq 1$.

The diffusion equation requires one initial condition and two boundary conditions. We will start with none of the substance in the membrane, so

$$\theta(x, 0) = 0$$

is the initial condition. We will fix the concentration at both boundaries using the boundary conditions

$$\theta(0, t) = 1$$
$$\theta(1, t) = 0$$

For completeness we note that this problem has an analytical solution:

$$\theta(x, t) = 1 - x - 2 \sum_{n=1}^{\infty} \exp\left[-(n\pi)^2 t\right] \frac{\sin n\pi x}{n\pi}$$

4.2 Solution in Python

Note: The code for this implementation can be found in the `examples/diffusion/python` directory. To run, use the command `python examples/diffusion/python/diffusion.py`.

In general we recommend that you begin by drafting your solution in pure Python. This lets you more quickly establish that your code is working as expected. If the resulting code is running fast enough for your purposes, then you haven't

wasted any time with unnecessary optimization. However, if the resulting code does run very slowly, it is not difficult or time consuming to convert it to Cython. We will utilize this paradigm for this example, which will also enable us to assess the sort of speed increase one might expect from Cythonization.

As with the previous example, the first step is to rewrite the problem as a DAE system in general form. We can only have one independent variable in our DAE system; for this we will choose time because it only has one initial condition. For the spatial coordinate we discretize into a regular grid of N points, with constant spacing $\Delta x = 1/(N-1)$ between the grid points. Thus the function $\theta(x,t)$ is replaced with a set of variables $\theta_i(t)$, where the subscript indicates a unique grid point.

We use the finite difference method to approximate the spatial second derivative at each grid point.

$$\frac{\partial^2 \theta_i}{\partial x^2} \approx \frac{\theta_{i+1} - 2\theta_i + \theta_{i-1}}{(\Delta x)^2}$$

The result is a system of coupled ordinary differential equations

$$\begin{aligned} \theta_1(t) &= 1 \\ \frac{d\theta_i}{dt} &= \frac{\theta_{i+1} - 2\theta_i + \theta_{i-1}}{(\Delta x)^2} \qquad i = 2, 3, \ldots, N-1 \\ \theta_N(t) &= 0 \end{aligned}$$

with corresponding initial condition

$$\theta_i(0) = 0 \qquad i = 1, 2, \ldots, N$$

(This approach is called the *method of lines*.) Finally we convert the above into general DAE form:

$$\begin{aligned} \theta_1(t) - 1 &= 0 \\ \frac{\theta_{i+1} - 2\theta_i + \theta_{i-1}}{(\Delta x)^2} - \frac{d\theta_i}{dt} &= 0 \qquad i = 2, 3, \ldots, N-1 \\ \theta_N(t) &= 0 \end{aligned}$$

Now that we have the DAE system in general form, we can draft the code that solves it. We will again use DASSL to solve this problem. As before, we develop a class that derives from the `DASSL` class in the `pydas` module, implementing a residual function based on the equations above. The complete class – only 25 lines! – is shown below:

```
import numpy
from pydas import DASSL

class DiffusionModel(DASSL):
    """
    A class for solving the diffusion equation in a 1D thin membrane. The
    attribute `N` is the number of grid points to use to discretize the
    spatial direction.
    """

    def __init__(self, N=10):
        self.N = N

    def residual(self, t, y, dydt):
        delta = numpy.zeros(y.shape[0], numpy.float64)
        # The grid point spacing
        dx = 1.0 / (self.N - 1)
        # Internal nodes
        for i in range(1, self.N-1):
            delta[i] = (y[i+1] - 2 * y[i] + y[i-1]) / (dx * dx) - dydt[i]
        # Left boundary (x = 0)
```

```
        i = 0
        delta[i] = y[i] - 1.0
        # Right boundary (x = 1)
        i = self.N - 1
        delta[i] = y[i]

        return delta, 0
```

What remains is to write code that instantiates an object of the above class, initializes the model with the appropriate initial conditions, and generates the solution. This code is given below. Unlike the simple example from the previous section, here we use the `advance()` method (instead of the `step()` method) to get solution data only at a set of desired time points. The solver will take as many steps as necessary to reach the desired time, then interpolate if it oversteps that time.

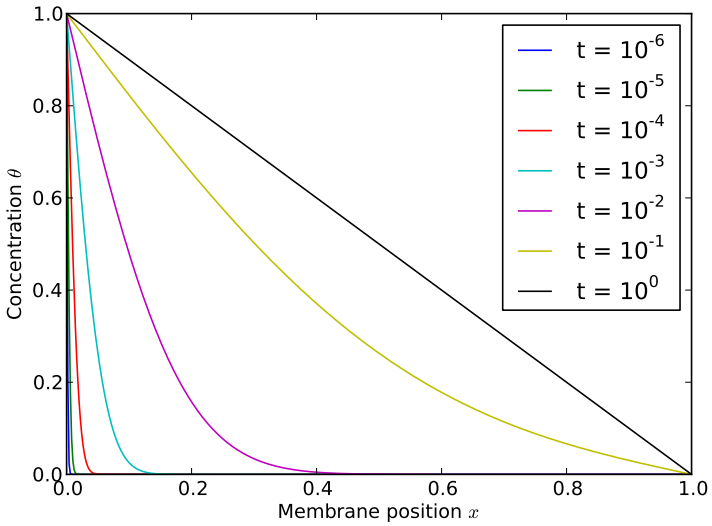
```
import numpy
from model import DiffusionModel

if __name__ == '__main__':

    # The times at which to obtain the solution
    tlist = numpy.array([10**i for i in range(-6, 1)], numpy.float64)
    # The number of grid points to use to discretize the spatial dimension
    N = 501

    # Set initial conditions
    t0 = 0.0
    y0 = numpy.zeros((N), numpy.float64)
    y0[0] = 1

    # Initialize the model
    model = DiffusionModel(N=N)
    dydt0 = - model.residual(t0, y0, numpy.zeros((N), numpy.float64))[0]
    model.initialize(t0, y0, dydt0)

    # Integrate to get the solution at each time point
    t = []; y = []
    for t1 in tlist:
        model.advance(t1)
        t.append(model.t)
        # You must make a copy of y because it is overwritten by DASSL at
        # each call to advance()
        y.append(model.y.copy())

    # Convert the solution vectors to numpy arrays
    t = numpy.array(t, numpy.float64)
    y = numpy.array(y, numpy.float64)
```

Note that we are once again able to provide consistent initial values for the derivatives $d\mathbf{y}/dt$ using the residual function.

Running the above code and plotting the results gives the following series of concentration profiles that are consistent with the analytical solution:

4.3 Solution in Cython

Note: The code for this implementation can be found in the `examples/diffusion/cython` directory. To run,



use the command `python examples/diffusion/cython/diffusion.py`.

Running the Python implementation takes some time. Profiling reveals that our `residual()` method clocks in at 110 s on this particular computer. This may or may not be considered "slow" depending on your target application, but let's see how fast we can make it by switching to Cython.

Cython is a language very similar to Python, but with added syntax to enable compilation to efficient C code. In particular, much of the speed boost comes from simply declaring the type of the variables being used. For numpy arrays, this enables fast array access using pointer arithmetic. Since we have a lot of array lookups in our residual function, we anticipate the potential for a significant speed boost. But first we have to code it.

Fortunately, our Python code serves as an excellent starting point. (It is a stated goal of the Cython project that they be able to compile as much unmodified Python code as possible, and as of Cython 0.14 they are doing a pretty good job of this.) From our profiling analysis we know that the `residual()` method is by far the bottleneck of the code. It's not so much that the function is slow (though it is), but that it is called more than 35000 times by DASSL. Thus we only need to Cythonize the `DiffusionModel` class.

We will abstain from a detailed discussion of the Cython syntax here, and instead refer you to the generally excellent Cython documentation for this. Put briefly, we must do the following:

- Rename the file to have the `.pyx` suffix (instead of `.py`) used by Cython.
- Add `cimport` statements to access the Cython declarations for numpy and PyDAS.
- Mark the `DiffusionModel` class as a Cython extension type by prepending the `cdef` keyword.
- Declare the type of each attribute of the `DiffusionModel` class.
- Declare the type of each parameter of the `residual()` method.
- Declare the type of each local variable of the `residual()` method.
- Create a `setup.py` file to use to compile the `.pyx` file to machine code. The compiled file has the suffix `.pyd` on Windows and `.so` on non-Windows systems, and acts as a valid Python module (i.e. can be imported as usual).

The new DiffusionModel class is only slightly longer than before:

```
import numpy
cimport numpy

from pydas cimport DASSL

cdef class DiffusionModel(DASSL):
    """
    An extension type for solving the diffusion equation in a 1D thin membrane.
    The attribute `N` is the number of grid points to use to discretize the
    spatial direction.
    """

    cdef public int N

    def __init__(self, N=10):
        self.N = N

    def residual(self, double t, numpy.ndarray[numpy.float64_t, ndim=1] y, numpy.ndarray[numpy.float6

        cdef Py_ssize_t i
        cdef double dx
        cdef numpy.ndarray[numpy.float64_t, ndim=1] delta

        dx = 1.0 / (self.N - 1)

        delta = numpy.zeros(y.shape[0], numpy.float64)
        # Internal nodes
        for i in range(1, self.N-1):
            delta[i] = (y[i+1] - 2 * y[i] + y[i-1]) / (dx * dx) - dydt[i]
        # Left boundary (x = 0)
        i = 0
        delta[i] = y[i] - 1.0
        # Right boundary (x = 1)
        i = self.N - 1
        delta[i] = y[i]

        return delta, 0
```

The block of code used to initialize and solve is completely unchanged from before.

Lastly, we need a way to compile the `.pyx` file above. One way is to use Python's distutils functionality. Below is a valid `setup.py` file for this purpose:

```
import numpy

if __name__ == '__main__':

    from distutils.core import setup
    from distutils.extension import Extension
    from Cython.Distutils import build_ext

    # The Cython modules to setup
    ext_modules = [
        Extension('model', ['model.pyx'], include_dirs=[numpy.get_include()]),
    ]

    # Run the setup command
    setup(
        cmdclass = {'build_ext': build_ext},
```

```
        ext_modules = ext_modules
    )
```

To execute the above, issue a command like

```
$ python setup.py build_ext --inplace
```

from the directory containing all of the relevant files.

Running the Cythonized version of the model is much faster, and gives completely identical output as the pure Python implementation. Profiling analysis results in a time of 0.71 s for the `residual()` method, a speed increase of over 150x! Of course, the amount of speed boost you get depends heavily on how your residual function is constructed, but gains of an order of magnitude or more are fairly common.

CHAPTER

FIVE

USING OBSERVERS OR LISTENERS WITH PYDAS

One can easily extract and store variables from the PyDAS solver class by writing a Listener or Observer class. This is helpful if one wishes to store the y variable continuously while the solver integrates without having to write to IO. For more information on the general usage of Listeners, see Wikipedia: Observer Pattern.

The `DASPK` and `DASSL` solver classes are a subclass of the `Subject` class, which allows for easy listening of internal variables from the solver. To do so, one needs to create a Listener class and notify the Listener during the simulation's execution to store that data.

5.1 Creating a Listener Class

Write a separate Listener class that will store the data you are interested in, e.g. YListener. The Listener class must include an `update(self,subject)` function, which is customized to store the data of interest.

```
class YListener(object):
    def __init__(self):
        self.data = []

    def update(self, subject):
        self.data.append(subject.y)
```

5.2 Client Code

Next, in your client code, create an instance of the listener, attach the listener to a solver class which customizes either the `DASPK` or `DASSL` class, in this case we have a `reactionSystem`. When integrating, one can call the `notify()` function, which automatically updates the Listener and saves the data you are interested in.

You may detach the Listener when you have finished collecting the data. Note that multiple listeners can be attached or detached.

```
def client_code(...):
    #create a listener instance:
    listener = YListener()


    reactionSystem.attach(listener)
    while reactionSystem.t < t_end:
        reactionSystem.step()
        reactionSystem.notify()
```

```
    #unregister as a listener
    reactionSystem.detach(listener)
```

[Petzold1982] L. R. Petzold. “A Description of DASSL: A Differential/Algebraic System Solver.” Sandia National Laboratories report SAND82-8637 (1982).