

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 5/25/2008 (rev. 1)

PyDSTool compilation Project Report

1. Introduction

This report documents the findings of the project described in the "PyDSTool compilation - Project Plan". Each deliverable is described in section 2. Section 3 contains notes taken during the project and section 4 is a high-level diagram of the design for dynamic reloading. Given time constraints, we tried to cover all the aspects of the project, knowing that some details could be ironed out during a real implementation. Areas to be improved are noted in the code.

The code was tested on the following platforms:

1. Windows XP with python 2.5.1 and GCC 3.4.5 mingw32
2. Ubuntu 7.10 with python 2.5.1 and GCC 4.1.3.

2. Deliverables

2.1 Design #1 and #2

Design #1 provides a solution to dynamically compile, reload and use C extension modules. Design #2 is meant to do the same thing while allowing control over temporary compilation directories and verbose output. We decided to provide one design that satisfies both sets of requirements. An overview of the design is provided a diagram in section 4.

2.1.1 Design #1: dynamic reloading

While python doesn't prevent us from reloading a module, python modules are not designed to be initialized more than once. The design we found relies on the use of dynamic C libraries from inside a python extension module that is loaded only once. To test the design, we added a method to the integrator module to be reloaded. The method is name `GetID()` and it returns an ID number for the module. Each time a new module is created, a unique ID number is written in a source file of the module. When the module is recompiled and reloaded, the ID number returned by `GetID()` should match the number in the generated source file. If it does, the module was properly recompiled and reloaded. If it does not match, the module was not recompiled and reloaded properly.

The design relies on a separation of the general integrator code, which doesn't change, from the functions implemented by the source code generated by PyDSTool. Those functions will change, but their interface remains the same. To make those functions dynamically reloadable, we will create a proxy function for each of them. Each generated function now becomes a pair of sister functions. The first will never change and the second can be modified by the user and loaded dynamically by the first. The function that never changes therefore needs to be loaded only once and will be part of the main integrator python module. It will call its sister function, found in a dynamically loaded C library. Here are the main parts of the added code:

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 5/25/2008 (rev. 1)

A python function to be called by PyDSTool is defined in interface.c:

```
PyObject* GetID() { return Py_BuildValue("f", HH_get_ID()); }
```

The HH_get_ID() function is implemented in the user-generated file HHtest2_vf.c:

```
double HH_get_ID() { return loader_get_ID(); }
```

In the original design of PyDSTool, this function would have a hard-coded ID number, which would prevent it from being reloadable.

Instead, HH_get_ID() calls a function implemented in loader.c that will find the appropriate function (in this case get_ID) in a dynamic library (see loader.c for more details on loading a dynamic library):

```
double loader_get_ID() {
    double id;
    void *lib_handle;
    double (*fn)();

    lib_handle = get_library();
    if (lib_handle==NULL) {
        return -1.0;
    }

    fn = dlsym(lib_handle, "get_ID");

    id = (*fn)();
    return id;
}
```

The get_ID function is the only file that needs to be recompiled after the user generates new code. In our example, it is defined in dyn_HHtest2_vf.c:

```
double get_ID() {
    return 0.51089;
}
```

Loading and reloading of a dynamic module can be done using the following python methods (implemented in interface.c):

```
UnloadLibrary()
LoadLibrary( "path to file to load" )
```

Those functions call loading and unloading functions from loader.c.

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 5/25/2008 (rev. 1)

The following files have been added to the 'integrator' directory:

- loader.h and loader.c: those provide the load_library() and close_library() functions used to load and unload the dynamic library.
- loaded.h: define the functions that will be implemented in the dynamic library (get_ID in our example).

The following files were modified:

- Interface.h and interface.c: function GetID(), LoadLibrary() and UnloadLibrary() are now defined. They will become python methods of the compiled integrator module.

The code generator of PyDSTool now generates two files:

- HHtest2_vf.c: the original user-generated source file, which now calls to functions defined in loader.c instead of simply implementing the code.
- Dyn_HHtest2_vf.c: contains the hard-coded functions to be dynamically loaded by the functions of loader.c.

2.1.2 Implementing this design in PyDSTool:

Here is a list of steps to implement this design in PyDSTool:

1. Modify the dopri and radau generators to write model source files and dynamic library source files (see the makeLibSource methods in the PyDSTool_reloader module). The model source file should call a proxy function to be implemented in loader.c

For example HHtest2_vf.c would contain:

```
void massMatrix(...) {
    loader_massMatrix(...);
}
```

Dyn_HHtest2_vf.c would contain:

```
void dyn_massMatrix(...) {
    [some code here];
}
```

2. Convert all hard-coded constants in the generated code (like N_AUXVARS) to a function like get_ID.

For example HHtest2_vf.c would contain:

```
int get_n_auxvars() { return loader_get_n_auxvars(); }
```

Dyn_HHtest2_vf.c would contain:

```
int dyn_get_n_auxvars() { return 0; }
```

3. Modify the integrator C source files that use the generated code to use proxy functions for constants and the loading and unloading functions of loader.c.

For example interface.c would contain:

```
assert(nAux == get_n_auxvars());
```

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 5/25/2008 (rev. 1)

4. Implement the loader.c functions to load the dynamic library and find the generated functions.

```
void loader_massMatrix(...) {
    void *lib_handle;
    void (*fn)();

    lib_handle = get_library();
    fn = dlsym(lib_handle, " dyn_massMatrix ");
    (*fn)();
}
```

5. Modify the dopri and radau compilation code to split it in two parts: the integrator module compilation and the dynamic library compilation. The integrator module can be loaded once and will reload user-modified versions of the dynamic library.

2.1.3 Design #2: compilation outputs

Design #2 was meant to add functionality to clean up the compilation outputs. The part of the code that steers the compilation can be found in the compileLib methods of the classes defined in PyDSTool_reloader.py. Two main features were added to solve this problem. One is to run the compilation in a sub-process and the second is to separate the build directory from the install directory. We did the following:

- Execute the compilation in a sub-process and write the output to a file. This prevents the user from seeing the output, which is nonetheless available for debugging purposes. It is possible to turn off the writing of the output.
- Place all temporary compiled objects in the 'build' directory, which can be deleted after compilation. We could also move the files in dopri853_temp and radau_temp to another location by changing the 'tempdir' parameters of the compilation functions. The 'build' directory can also be changed by adding the proper flag to the distutils call.
- When possible, run the 'install' from distutils. That was not possible for numpy.distutils on Windows. That appears to be a bug in numpy.distutils. In that case, the files were moved by the compilation script using shutil. We did not have that problem on linux and used 'install' in that case.
- Install the output module in the 'lib/python' directory. This is done by using the '--home' flag of distutils. One could also choose to install the output module in 'dopri853_temp' or 'radau_temp'. Note that the 'lib/python' install directory should be created before running the test cases.

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 5/25/2008 (rev. 1)

2.2 Prototypes

The prototypes are found in the PyDSTool_reloader module. They are meant to emulate the Dopri_ODEsystem and Radau_ODEsystem classes of PyDSTool.

2.2.1 Prototype #0

PyDSTool_reloader.Integrator is a prototype based on distutils that will build, load, rebuild and reload a dopri C extension module. It uses the compilation code extracted from PyDSTool and corresponds to the so-called current design.

2.2.2 Prototype #1

Prototype #1 was defined in the project plan as a prototype based on distutils that will build, load, rebuild and reload both a dopri and a radau C extension module. The prototype will be based on Design #1 and Design #2. Since dopri and radau are compiled differently, they were implemented in two classes:

PyDSTool_reloader.DynIntegrator is a prototype based on distutils that will build, load, rebuild and reload a dopri C extension module. It is based on the combined design documented in the previous section.

PyDSTool_reloader.DynRadauIntegrator is a prototype based on distutils that will build, load, rebuild and reload a radau C extension module. It is based on the combined design documented in the previous section.

2.2.3 Prototype #2

Prototype #2 was defined in the project plan as a prototype based on setuptools that will build, load, rebuild and reload both a dopri and a radau C extension module. The prototype will be based on Design #1 and Design #2. Since this prototype is very similar to prototype #1, we used the same classes and made a "setuptools" flag available in the compileLib method.

Note that since setuptools is based on python's distutils, it can't compile fortran sources directly. Prototype #2 is therefore not available for Radau. An alternate solution would be to create a makefile to compile the fortran code outside the setup.py and link to the produced library.

2.3 Test cases

Test test cases are available in the 'tests' directory.

2.3.1 Test Case #1

Script that will use distutils to build, load, rebuild and reload a C extension module from files generated by one of the dopri and one of the radau test scripts. The test case will follow the current PyDSTool design. The test case is expected to fail.

See TestCase1.py. With the results of Test Case #1 for dopri, we don't expect to learn anything from writing a prototype and a test case for radau based on the current design. Although the compilation code for radau was extracted and placed in compile_radau.py, Test Case #1 is not provided for radau. One can be provided if that proves to be a problem.

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 5/25/2008 (rev. 1)

2.3.2 Test Case #2

Script that will use Prototype #1 to build, load, rebuild and reload a C extension module from files generated by one of the dopri and one of the radau test scripts. The test case is expected to pass.

See TestCase2a.py for dopri, and TestCase2b.py for radau. Those test cases were also written as unit tests in UnitTest2_dopri.py and UnitTest2_radau.py.

2.3.3 Test Case #3

Script that will use setuptools to build, load, rebuild and reload a C extension module from files generated by one of the dopri and one of the radau test scripts. The test case will follow the current PyDSTool design. The test case is expected to fail.

See TestCase3.py. Note that since setuptools is based on python's distutils, it can't compile fortran sources directly. Test Case #3 is therefore not available for Radau.

2.3.4 Test Case #4

Script that will use Prototype #2 to build, load, rebuild and reload a C extension module from files generated by one of the dopri and one of the radau test scripts. The test case is expected to pass.

See pre_TestCase4.py and TestCase4.py. Since the eggs generated by setuptools are meant to be installed before use, pre_TestCase4.py should be run before TestCase4.py. It will build and install the new module.

2.4 File list:

- cleanup.py: a file to clean up the directory tree before the execution of a test, to ensure that there is no interference between tests.
- compile_dopri.py: compilation code for dopri. Include the extracted PyDSTool compilation code and the modified version that generator setup_dopri_internal.py.
- compile_radau.py: compilation code for radau. Include the extracted PyDSTool compilation code and the modified version that generator setup_radau_internal.py.
- compile_dyn_dopri.py: compilation code for the dynamic library. It is used for both dopri and radau. A makefile could be used instead of the direct gcc calls.
- The 'tests' directory includes all the test cases from the previous section. It also includes the test files event_test_vf_original.c and HHtest_vf_original.c, which were generated by running the PyDSTool tests called HH_model_Cintegrator.py and radau_event_test.py.
- The integrator directory includes all the files from the integrator directory of PyDSTool. As detailed in this document, some interface.h and interface.c were modified and some loader.h, loader.c and loaded.h were added.

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 5/25/2008 (rev. 1)

3. Notes:

This section contains various notes taken during the project.

- The dynamic library needs to be placed at a location where it can be retrieved for loading. The install directory is the option we chose (the lib/python directory in our working directory).
- When running 'install' setuptools needs the install directory to be on the PYTHONPATH. For an alternate install, like we are doing here, we need write an altinstall.pth in the site-packages directory. See pre_TestCase4.py for details.
- Distutils compilations: for a cleaner location of the temp outputs, use relative paths in file names instead of absolute paths.
- Test Case #1 is dopri only, since radau will produce the same result.
- We compile everything from compileLib, with a flag to compile the whole thing or just the dynamic library. Ideally the module compilation would be done a general setup.py and only the dynamic library would be compiled.
- Note that if you compile the whole thing once and call setup again, the sources that were already compiled are not recompiled. You can test this by running PyDSTool_reloader and compile_dopri.py one after the other and looking at the time of the .o files.
- setuptools creates other temp directories that add to the messiness of the directory structure.
- numpy.distutils doesn't process 'install' correctly on Windows, where it wants to find a compiler for 'install' because of swig. Since 'install' does not allow for a '--compiler' flag, running 'install' crashes. This problem does not appear on linux.
- setup tools's eggs can't easily be imported if they are created by the same process that uses them. Create the main module first before trying to reload a new dynamic integrator (which does not require a new egg).
- Since pre_TestCase 4.py writes to the site-packages to declare an alternate install directory, it should run the first time as root. After it is done once, any user can compile new modules. The procedure should be: (1) run pre_TestCase4.py as root, (2) run cleanup.py as root, (3) run pre_TestCase4.py and TestCase4.py as a normal user.
- Make sure to always clean up the 'build' directory when installing new dopri/radau modules to avoid interference between dopri and radau when running install. To avoid this, one can also use a different build directory for radau and dopri.
- Depending on whether one wants to let the user have multiple integrator modules at the same time, the generator code that calls the dynamic library functions could be integrated with the general integrator code (i.e. in our example, there is no real need for HHtest2_vf.c).
- We still have the problem that distutils and numpy.distutils place the output python module from swig at different locations. We have to move it by hand when using numpy.

4. UML overview of the dynamic design:

