

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 6/21/2008 (rev. 1)

# PyDSTool: implementation of dynamic loading

## Project Report

### 1. Introduction

This report documents the work of the project described in the document "PyDSTool: implementation of dynamic loading - Project Plan". The implementation of dynamic loading followed the design described in the report "PyDSTool compilation: Project Report".

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.
3. Mac OS X 10.4.11 with python 2.51 and GCC 4.0.1

### 2. Implementation details

The dynamic loading was implemented according to the findings of the previous PyDSTool compilation project. The compilation was pulled out of the *DopriODEsystem* code and is now executed in a separate process. This allows more control over the location of the installation of the new modules.

All the functions previously defined in generated source code and called from the integrator code have been replaced by functions to be part of a dynamically loaded library. The constants defined in the user generated code have been replaced by functions that return their value.

It was found that there are several ways to create a *DopriODEsystem* object. Although we gave that class the flexibility to be instantiated with an option to use or not to use the dynamic library functionality, the dynamic option is now forced ON. All objects that can create a *DopriODEsystem* should have an option to turn the dynamic functionality ON or OFF. Once that is done, the line of the `__init__` function forcing the dynamic option to be chosen can be removed. When turned OFF, the old functionality is recovered.

Usage note:

When a new *dopri* object is instantiated, it tries to unload any dynamic library that might already have been loaded by having previously used the same user-generated integrator module. To prevent older data in a *DopriODEsystem* object to corrupt the new computations, the *cleanup()* method of the *dopri* class should be called. That will call the *CleanUp()* method of the user-generated integrator module in addition to unloading any previously loaded dynamic library.

#### 2.1 Files added

The following is a list of files that were added:

- PyDSTool/cleanup.py

Script to execute between tests. Although not exhaustive, it will remove most of the temporary files that can be created by test scripts.

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 6/21/2008 (rev. 1)

- PyDSTool/Generator/compile\_dopri.py  
Utilities to write a setup script for the new module. Also provides a utility method to return the absolute path of a dynamic library. It is also where we define the installation directory.
- PyDSTool/Generator/compile\_util.py  
Since the integrator code never changes, it will only be compiled once. The compilation code uses `__DOPRI__`, `__RADAU__`, and `__PYDYNAMIC__` compiler flags. This means that the integrator code should be recompiled when those flags change. This module provides utility methods to read and write a configuration file to log which compiler flags were set the last time the code was compiled. These will allow us to know when we should recompile the integrator code after switching from radau to dopri or from static modules to dynamic modules. See notes in section 3 about the directory structure.
- PyDSTool/Generator/compile\_dyn\_dopri.py  
This script compiles and installs the dynamic library. Once dynamic loading is extended to radau, some of the methods in this file might be better placed in `compile_util.py`.
- PyDSTool/Generator/relpath.py  
Small third-party module to find the common prefix of two file paths.
- PyDSTool/integrator/loader.h  
Header file defining the dynamic loader functions.
- PyDSTool/integrator/loader.c  
Implementation of the dynamic loader functions.
- PyDSTool/tests/comparator.py  
Utility module to save the output data of a test and compare it to a previously saved data set.
- PyDSTool/tests/CIN\_reload\_test.py  
Slightly modified version of the `CIN_reload_test.py`. Note the `CB.cleanup()` line.

## 2.2 Files modified

The following is a list of files that were modified:

- PyDSTool/FuncSpec.py  
Added `'dynamic'` as an optional instantiation parameter. Modified the output of `_genAuxFnC` and `_genSpecC` to create source code for the dynamic library when the `'dynamic'` flag is `True`.
- PyDSTool/integrator.py  
Added a `'_dynamic_lib'` data member and a `cleanup()` method to unload the dynamic library and call the `CleanUp()` method of the integrator module.
- PyDSTool/Model.py  
Added `cleanup()` method.

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 6/21/2008 (rev. 1)

- PyDSTool/Generator/Dopri\_ODEsystem.py
 

Modified the *dopri* class to load a dynamic library when the *dynamic\_lib* argument is set to *True* at instantiation. Heavily modified *makeLibSource()* and *compileLib()* to create and compile modules that use dynamic libraries when the option is turned ON. A *cleanup()* method was added. It should be called to unload the dynamic library between recompilations of user-generated modules.

*Note: It seems that more than one object in the code can create a Dopri\_ODEsystem object. I did not attempt to find them all and add the 'dynamic' flag to their API. An optional 'dynamic' instantiation parameter was added to Dopri\_ODEsystem. That parameter is forced to be True. That should be removed once the dynamic option is propagated to the whole system.*
- PyDSTool/Generator/ODEsystem.py
  1. Added *'dynamic'* as an optional instantiation parameter. Added an empty *cleanup()* method to be filled by all *ODEsystem* subclasses that use dynamic libraries.
- PyDSTool/integrator/dop853.i
  2. Defined *LoadLibrary* and *UnloadLibrary* as python methods of the newly created modules.
- PyDSTool/integrator/dop853mod.c
 

Use the `__PYDYNAMIC__` compiler flag to know when to use a function to get a constant defined by the user-generated code.
- PyDSTool/integrator/interface.h
 

Defined *LoadLibrary* and *UnloadLibrary* as python methods of the newly created modules.
- PyDSTool/integrator/interface.c
  3. Implemented the *LoadLibrary* and *UnloadLibrary* python methods. Use the `__PYDYNAMIC__` compiler flag to know when to use a function to get a constant defined by the user-generated code.

## 2.3 Implementing dynamic libraries for radau

Here is a list of steps to implement this design for radau:

4. Modify the *makeLibSource()* and *compileLib()* methods of *Radau\_ODEsystem* and make sure that the integrator is recompiled if the user switches from dopri to radau.
5. If needed, modify the *\_genAuxFnC()* and *\_genSpecC()* methods of *FuncSpec.py*
6. Check that all constants and functions are dynamically loaded. If not, modify *integrator.h*, *integrator.c*, *loader.h* and *loader.c*.
7. Modify *radau5.i* and *radau5mod.c* to recognize the `__PYDYNAMIC__` flag and use loader functions to access user-generated constants. See modifications to *dop853.i* and *dop853mod.c*.

PyDSTool compilation	Author: Mathieu Doucet
Project Report	Date: 6/21/2008 (rev. 1)

### 3. Notes:

This section contains various notes taken during the project.

- I suggest not keeping global variables that need access on the dynamic side. Pass the parameters with the function arguments instead. At least make it a struct. As it is the dynamic code must be able to access global data filled on the static side. I didn't want to modify your code on that one, so I implemented a function to give the dynamic library pointers to those globals. It works fine, but I would strongly suggest to refactor.
- In *makeLibSource*: I removed the definitions of *auxvars*, *jacobian* and *jacobianParam* since they are already defined in *vfield.h*.
- I did not want to change the API, but a good change would be to always have a return value to the functions: for instance, *void vfield()* becomes *int vfield()*. That way we can return a flag for success or failure.
- Added *\_dyn\_header*, *\_global\_defs*, and *\_constant\_func* to *DopriODEsystem*, but this should be transferred to *ODEsystem* once it is confirmed that they should be the same for *dopri* and *radau*.
- Might want to think about using something like the 'logging' module to log errors.
- One constraint on the system is that if the user wants to re-compile a module with new compile flags (for instance by change from dynamic to non-dynamic), that application has to be restarted. This is the same problem we originally had, that an extension python module is not meant to be reloaded. Note that new modules with different names can be compiled and loaded, even with different compile flags.
- I recommend keeping a directory structure that separates the generated source from the compilation outputs. Source generation (*makeLibSource*) and code compilation (*compileLib*) are in two methods. Since we might have to delete the build directory at compile time after a change of compilation flags, we want to make sure that the user-generated code is not deleted during that process. Having a separate directory for source and compilation outputs will ensure that the generation and compilation processes are truly independent of each other.
- As it is, the code is not protected against attempts to reload a module in non-dynamic mode. For instance, *CIN\_reload\_test.py* will crash if the dynamic option is *False*. Unless user-generated modules are cleaned when starting the application, it is not sufficient to check for the existence of those modules on the file system to decide whether the user is attempting to reload. If at all needed, a better protection mechanism should be written.
- The *distutils* "install" command looks for a directory named 'build'. While it is possible to install the code anywhere with the *-home* flag and the generated source can be placed in any directory by changing the *\_compilation\_tempdir* data member of *DopriODEsystem*, the build directory should be kept unchanged to benefit from the installation functionality of *distutils*.
- Bug: in *DopriODEsystem.compute()*, if the integration fails, the line

*info(self.diagnostics.outputStats, "Output statistics")*

will raise an exception.