

SWIG and Python

9

This chapter describes SWIG's support of Python. Many of the example presented here will have a scientific bias given Python's increasing use in scientific applications (this is how I primarily use Python), but the techniques are widely applicable to other areas.

Preliminaries

SWIG 1.1 works with Python 1.3 and Python 1.4. Given the choice, you should use the latest version of Python. You should also determine if your system supports shared libraries and dynamic loading. SWIG will work with or without dynamic loading, but the compilation process will vary.

Running SWIG

To build a Python module, run SWIG using the `-python` option :

```
%swig -python example.i
```

This will produce 2 files. The file `example_wrap.c` contains all of the C code needed to build a Python module and a documentation file describes the resulting interface. To build a Python module, you will need to compile the file `example_wrap.c` and link it with the rest of your program (and possibly Python itself). When working with shadow classes, SWIG will also produce a `.py` file, but this is described later.

Getting the right header files

In order to compile, you need to locate the following directories that are part of the Python distribution :

For Python 1.3 :

```
/usr/local/include/Py  
/usr/local/lib/python/lib
```

For Python 1.4 :

```
/usr/local/include/python1.4  
/usr/local/lib/python1.4/config
```

The exact location may vary on your machine, but the above locations are typical.

Compiling a dynamic module

To build a shared object file, you need to compile your module in a manner similar to the following (shown for Irix):

```
% swig -python example.i
% gcc -c example.c
% gcc -c example_wrap.c -DHAVE_CONFIG_H -I/usr/local/include/python1.4 \
    -I/usr/local/lib/python1.4/config
% ld -shared example.o example_wrap.o -o examplemodule.so
```

Unfortunately, the process of building a shared object file varies on every single machine so you may need to read up on the man pages for your C compiler and linker.

When building a dynamic module, the name of the output file is important. If the name of your SWIG module is “example”, the name of the corresponding object file should be “examplemodule.so” (or equivalent depending on your machine). The name of the module is specified using the `%module` directive or `-module` command line option.

While dynamic loading is the preferred method for making SWIG modules, it is not foolproof and not supported on all machines. In these cases, you can rebuild the Python interpreter with your extensions added.

Rebuilding the Python interpreter (aka. static linking)

The normal procedure for adding a new module to Python involves finding the Python source, adding an entry to the `Modules/Setup` file, and rebuilding the interpreter using the Python Makefile. While it's possible to simplify the process by using the `VPATH` feature of ‘make’, I've always found the process to be a little too complicated.

SWIG provides an extremely easy, although somewhat unconventional, mechanism for rebuilding Python using SWIG's library feature. When you want to build a static version of Python, simply make an interface file like this :

```
%module example

extern int fact(int);
extern int mod(int, int);
extern double My_variable;

#include embed.i           // Include code for a static version of Python
```

The `embed.i` library file includes supporting code that contains everything needed to rebuild Python. To build your module, simply do the following :

```
% swig -python example.i
% gcc example.c example_wrap.c -DHAVE_CONFIG_H -I/usr/local/include/python1.4 \
    -I/usr/local/lib/python1.4/config \
    -L/usr/local/lib/python1.4/config -lModules -lPython -lObjects -lParser -lm \
    -o mypython
```

On some machines, you may need need to supply additional libraries on the link line. In particular, you may need to supply `-lsocket`, `-lnsl`, and `-ldl`.

It is also possible to add the `embed.i` library to an existing interface by running SWIG as follows :

```
% swig -python -lembed.i example.i
```

The `embed.i` file uses all of the modules that are currently being used in your installed version of Python. Thus, your new version of Python will be identical to the old one except with your new module added. If you have configured Python to use modules such as `tkinter`, you may need to supply linkage to the Tcl/Tk libraries and X11 libraries.

Python's `main()` program is rather unfriendly towards C++ code, but SWIG's `embed.i` module provides a replacement that can be compiled with the C++ compiler--making it easy to build C++ Python extensions.

The `embed.i` library should only be used with Python 1.4. If you are using Python 1.3, you should use the file `embed13.i` instead (this can be done by making a symbolic link in the SWIG library) or simply using the `-l` option.

Using your module

To use your module in Python, simply use Python's `import` command. The process is identical regardless of whether or not you used dynamic loading or rebuilt the Python interpreter :

```
% python
>>> import example
>>> example.fact(4)
24
>>>
```

Compilation problems and compiling with C++

For the most part, compiling a Python module is straightforward, but there are a number of potential problems :

- Dynamic loading is not supported on all machines. If you can't get a module to build, you might try building a new version of Python using static linking instead.
- In order to build C++ modules, you may need to link with the C++ compiler using a command like `'c++ -shared example_wrap.o example.o -o examplemodule.so'`
- If building a dynamic C++ module using `g++`, you may also need to link against `lib-gcc.a`, `libg++.a`, and `libstc++.a` libraries.
- Make sure you are using the correct header files and libraries. A module compiled with Python 1.3 headers probably won't work with Python 1.4.

Building Python Extensions under Windows 95/NT

Building a SWIG extension to Python under Windows 95/NT is roughly similar to the process used with Unix. Normally, you will want to produce a DLL that can be loaded into the Python interpreter. This section covers the process of using SWIG with Microsoft Visual C++ 4.x although the procedure may be similar with other compilers. SWIG currently supports both the basic Python release and `Pythonwin`. In order to build extensions, you will need to download the source distribution to these packages as you will need the Python header files.

Running SWIG from Developer Studio

If you are developing your application within Microsoft developer studio, SWIG can be invoked as a custom build option. The process roughly follows these steps :

- Open up a new workspace and use the AppWizard to select a DLL project.
- Add both the SWIG interface file (the .i file), any supporting C files, and the name of the wrapper file that will be created by SWIG (ie. `example_wrap.c`). Note : If using C++, choose a different suffix for the wrapper file such as `example_wrap.cxx`. Don't worry if the wrapper file doesn't exist yet--Developer Studio will keep a reference to it around.
- Select the SWIG interface file and go to the settings menu. Under settings, select the "Custom Build" option.
- Enter "SWIG" in the description field.
- Enter "`swig -python -o $(ProjDir)\$(InputName)_wrap.c $(InputPath)`" in the "Build command(s) field"
- Enter "`$(ProjDir)\$(InputName)_wrap.c`" in the "Output files(s) field".
- Next, select the settings for the entire project and go to "C++:Preprocessor". Add the include directories for your Python installation under "Additional include directories".
- Define the symbol `__WIN32__` under preprocessor options.
- Finally, select the settings for the entire project and go to "Link Options". Add the Python library file to your link libraries. For example "python14.lib". Also, set the name of the output file to match the name of your Python module (ie. `example.dll`).
- Build your project.

Now, assuming all went well, SWIG will be automatically invoked when you build your project. Any changes made to the interface file will result in SWIG being automatically invoked to produce a new version of the wrapper file. To run your new Python extension, simply run Python and use the `import` command as normal. For example :

```
MSDOS > python
>>> import example
>>> print example.fact(4)
24
>>>
```

Using NMAKE

Alternatively, SWIG extensions can be built by writing a Makefile for NMAKE. Make sure the environment variables for MSVC++ are available and the MSVC++ tools are in your path. Now, just write a short Makefile like this :

```
# Makefile for building a Python extension

SRCS      = example.c
IFILE     = example
INTERFACE = $(IFILE).i
WRAPFILE  = $(IFILE)_wrap.c

# Location of the Visual C++ tools (32 bit assumed)

TOOLS     = c:\msdev
TARGET    = example.dll
```

```

CC          = $(TOOLS)\bin\cl.exe
LINK        = $(TOOLS)\bin\link.exe
INCLUDE32   = -I$(TOOLS)\include
MACHINE     = IX86

# C Library needed to build a DLL

DLLIBC      = msvcrt.lib oldnames.lib

# Windows libraries that are apparently needed
WINLIB      = kernel32.lib advapi32.lib user32.lib gdi32.lib comdlg32.lib
winspool.lib

# Libraries common to all DLLs
LIBS        = $(DLLIBC) $(WINLIB)

# Linker options
LOPT        = -debug:full -debugtype:cv /NODEFAULTLIB /RELEASE /NOLOGO \
              /MACHINE:$(MACHINE) -entry:_DllMainCRTStartup@12 -dll

# C compiler flags

CFLAGS      = /Z7 /Od /c /nologo
PY_INCLUDE  = -Id:\python-1.4\Include -Id:\python-1.4 -Id:\python-1.4\PC
PY_LIB      = d:\python-1.4\vc40\python14.lib
PY_FLAGS    = /D__WIN32__

python::
    swig -python -o $(WRAPFILE) $(INTERFACE)
    $(CC) $(CFLAGS) $(PY_FLAGS) $(PY_INCLUDE) $(SRCS) $(WRAPFILE)
    set LIB=$(TOOLS)\lib
    $(LINK) $(LOPT) -out:example.dll $(LIBS) $(PY_LIB) example.obj example_wrap.obj

```

To build the extension, run NMAKE (you may need to run `vcvars32` first). This is a pretty simplistic Makefile, but hopefully its enough to get you started.

The low-level Python/C interface

The SWIG Python module is based upon a basic low-level interface that provides access to C functions, variables, constants, and C++ classes. This low-level interface is often used to create more sophisticated interfaces (such as shadow classes) so it may be hidden in practice.

Modules

The SWIG `%module` directive specifies the name of the Python module. If you specified `'%module example'`, then everything found in a SWIG interface file will be contained within the Python `'example'` module. Make sure you don't use the same name as a built-in Python command or standard module or your results may be unpredictable.

Functions

C/C++ functions are mapped directly into a matching Python function. For example :

```
%module example
```

```
extern int fact(int n);
```

gets turned into the Python function `example.fact(n)` :

```
>>> import example
>>> print example.fact(4)
24
>>>
```

Variable Linking

SWIG provides access to C/C++ global variables, but the mechanism is slightly different than one might expect due to the object model used in Python. When you type the following in Python :

```
a = 3.4
```

“a” becomes a name for an object containing the value 3.4. If you later type

```
b = a
```

Then “a” and “b” are both names for the object containing the value 3.4. In other words, there is only one object containing 3.4 and “a” and “b” are both names that refer to it. This is a very different model than that used in C. For this reason, there is no mechanism for mapping “assignment” in Python onto C global variables (because assignment in Python is really a naming operation).

To provide access to C global variables, SWIG creates a special Python object called ‘cvar’ that is added to each SWIG generated module. This object is used to access global variables as follows :

```
// SWIG interface file with global variables
%module example
...
extern int My_variable;
extern double density;
...
```

Now in Python :

```
>>> import example
>>> # Print out value of a C global variable
>>> print example.cvar.My_variable
4
>>> # Set the value of a C global variable
>>> example.cvar.density = 0.8442
>>> # Use in a math operation
>>> example.cvar.density = example.cvar.density*1.10
```

Just remember, all C globals need to be prefixed with a “cvar.” and you will be set. If you would like to use a name other than “cvar”, it can be changed using the `-globals` option :

```
% swig -python -globals myvar example.i
```

Some care is in order when importing multiple SWIG modules. If you use the “`from <file>`”

`import *` style of importing, you will get a name clash on the variable `cvar` and will only be able to access global variables from the last module loaded. SWIG does not create `cvar` if there are no global variables in a module.

Constants

C/C++ constants are installed as new Python objects containing the appropriate value. These constants are given the same name as the corresponding C constant. “Constants” are not guaranteed to be constants in Python---in other words, you are free to change them and suffer the consequences!

Pointers

Pointers to C/C++ objects are represented as character strings such as the following :

```
_100f8e2_Vector_p
```

A NULL pointer is represented by the string “NULL”. You can also explicitly create a NULL pointer consisting of the value 0 and a type such as :

```
_0_Vector_p
```

To some Python users, the idea of representing pointers as strings may seem strange, but keep in mind that pointers are meant to be opaque objects. In practice, you may never notice that pointers are character strings. There is also a certain efficiency in using this representation as it is easy to pass pointers around between modules and it is unnecessary to rely on a new Python datatype. Eventually, pointers may be represented as special Python objects, but the string representation works remarkably well so there has been little need to replace it.

Structures

The low-level SWIG interface only provides a simple interface to C structures. For example :

```
struct Vector {
    double x,y,z;
};
```

gets mapped into the following collection of C functions :

```
double Vector_x_get(Vector *obj)
double Vector_x_set(Vector *obj, double x)
double Vector_y_get(Vector *obj)
double Vector_y_set(Vector *obj, double y)
double Vector_z_get(Vector *obj)
double Vector_z_set(Vector *obj, double z)
```

These functions are then used in the resulting Python interface. For example :

```
# v is a Vector that got created somehow
>>> Vector_x_get(v)
3.5
>>> Vector_x_set(v,7.8)           # Change x component
>>> print Vector_x_get(v), Vector_y_get(v), Vector_z_get(v)
7.8 -4.5 0.0
>>>
```

Similar access is provided for unions and the data members of C++ classes.

C++ Classes

C++ classes are handled by building a set of low level accessor functions. Consider the following class :

```
class List {
public:
    List();
    ~List();
    int  search(char *item);
    void insert(char *item);
    void remove(char *item);
    char *get(int n);
    int  length;
    static void print(List *l);
};
```

When wrapped by SWIG, the following functions will be created :

```
List      *new_List();
void      delete_List(List *l);
int       List_search(List *l, char *item);
void      List_insert(List *l, char *item);
void      List_remove(List *l, char *item);
char      *List_get(List *l, int n);
int       List_length_get(List *l);
int       List_length_set(List *l, int n);
void      List_print(List *l);
```

Within Python, these functions used to access the C++ class :

```
>>> l = new_List()
>>> List_insert(l,"Ale")
>>> List_insert(l,"Stout")
>>> List_insert(l,"Lager")
>>> List_print(l)
Lager
Stout
Ale
>>> print List_length_get(l)
3
>>> print l
_1008560_List_p
>>>
```

C++ objects are really just pointers. Member functions and data are accessed by simply passing a pointer into a collection of accessor functions that take the pointer as the first argument.

While somewhat primitive, the low-level SWIG interface provides direct and flexible access to C++ objects. As it turns out, a more elegant method of accessing structures and classes is available using shadow classes.

Python shadow classes

The low-level interface generated by SWIG provides access to C structures and C++ classes, but it doesn't look much like a class that might be created in Python. However, it is possible to use the low-level C interface to write a Python class that looks like the original C++ class. In this case, the Python class is said to “shadow” the C++ class. That is, it behaves like the original class, but is really just a wrapper around a C++ class.

A simple example

For our earlier List class, a Python shadow class could be written by hand like this :

```
class List:
    def __init__(self):
        self.this = new_List()
    def __del__(self):
        delete_List(self.this)
    def search(self,item):
        return List_search(self.this,item)
    def insert(self,item):
        List_insert(self.this,item)
    def remove(self,item):
        List_remove(self.this,item)
    def get(self,n):
        return List_get(self.this,n)
    def __getattr__(self,name):
        if name == "length": return List_length_get(self.this)
        else :               return self.__dict__[name]
    def __setattr__(self,name,value):
        if name == "length": List_length_set(self.this,value)
        else :               self.__dict__[name] = value
```

When used in a Python script, we can use the class as follows :

```
>>> l = List()
>>> l.insert("Ale")
>>> l.insert("Stout")
>>> l.insert("Lager")
>>> List_print(l.this)
Lager
Stout
Ale
>>> l.length
3
```

Obviously, this is a much nicer interface than before--and it only required a small amount of Python coding.

Why write shadow classes in Python?

While one could wrap C/C++ objects directly into Python as new Python types, this approach has a number of problems. First, as the C/C++ code gets complicated, the resulting wrapper code starts to become extremely ugly. It also becomes hard to handle inheritance and more advanced language features. A second, and more serious problem, is that Python “types” created

in C can not be subclassed or used in the same way as one might use a real Python class. As a result, it is not possible to do interesting things like create Python classes that inherit from C++ classes.

By writing shadow classes in Python instead of C, the classes become real Python classes that can be used as base-classes in an inheritance hierarchy or for other applications. Writing the shadow classes in Python also greatly simplifies coding complexity as writing in Python is much easier than trying to accomplish the same thing in C. Finally, by writing shadow classes in Python, they are easy to modify and can be changed without ever recompiling any of the C code. The downside to this approach is worse performance--a concern for some users.

The problems of combining C++ and Python have been of great interest to the Python community. SWIG is primarily concerned with accessing C++ from Python. Readers who are interested in more than this (and the idea of accessing Python classes from C++) are encouraged to look into the MESS extension which aims to provide a tighter integration between C++ and Python. The recently announced GRAD package also shows much promise and provides very comprehensive C++/Python interface.

Automated shadow class generation

SWIG can automatically generate shadow classes if you use the `-shadow` option :

```
swig -python -shadow interface.i
```

This will create the following two files :

```
interface_wrap.c
module.py
```

The file `interface_wrap.c` contains the normal SWIG C/C++ wrappers. The file `module.py` contains the Python code corresponding to shadow classes. The name of this file will be the same as specified by the `%module` directive in the SWIG interface file.

Associated with the two files are TWO Python modules. The C module `'modulec'` contains the low-level C interface that would have been created without the `-shadow` option. The Python module `'module'` contains the Python shadow classes that have been built around the low-level interface. To use the module, simply use `'import module'`. For all practical purposes, the `'modulec'` module is completely hidden although you can certainly use it if you want to.

Compiling modules with shadow classes

To compile a module involving shadow classes, you can use the same procedure as before except that the module name now has an extra `'c'` appended to the name. Thus, an interface file like this

```
%module example
... a bunch of declarations ...
```

might be compiled as follows :

```
% swig -python -shadow example.i
% gcc -c example.c example_wrap.c -I/usr/local/include/python1.4 \
-I/usr/local/lib/python1.4/config -DHAVE_CONFIG_H
```

```
% ld -shared example.o example_wrap.o -o examplecmodule.so
```

Notice the naming of ‘examplecmodule.so’ as opposed to ‘examplemodule.so’ that would have been created without shadow classes.

When using static linking, no changes need to be made to the compilation process.

Where to go for more information

Shadow classes turn out to be so useful that they are used almost all of the time with SWIG. All of the examples presented here will assume that shadow classes have been enabled. The precise implementation of shadow classes is described at the end of this chapter and is not necessary to effectively use SWIG.

About the Examples

The next few sections will go through a series of Python examples of varying complexity. These examples are designed to illustrate how SWIG can be used to integrate C/C++ and Python in a variety of ways. Some of the things that will be covered include :

- Controlling a simple C++ program with Python
- Wrapping a C library.
- Adding Python methods to existing C++ classes
- Accessing arrays and other common data structures.
- Building reusable components.
- Writing C/C++ callback functions in Python.

Solving a simple heat-equation

In this example, we will show how Python can be used to control a simple physics application-- in this case, some C++ code for solving a 2D heat equation. This example is probably overly simplistic, but hopefully it’s enough to give you some ideas.

The C++ code

Our simple application consists of the following two files :

```
// File : pde.h
// Header file for Heat equation solver

#include <math.h>
#include <stdio.h>

// A simple 2D Grid structure

// A simple structure for holding a 2D grid of values
struct Grid2d {
    Grid2d(int ni, int nj);
    ~Grid2d();
    double **data;
    int     xpoints;
    int     ypoints;
};
```

```
// Simple class for solving a heat equation */
class Heat2d {
private:
    Grid2d    *work;                // Temporary grid, needed for solver
    double    h,k;                 // Grid spacing
public:
    Heat2d(int ni, int nj);
    ~Heat2d();
    Grid2d    *grid;               // Data
    double    dt;                 // Timestep
    double    time;               // Elapsed time
    void      solve(int nsteps);   // Run for nsteps
    void      set_temp(double temp); // Set temperature
};
```

The supporting C++ code implements a simple partial differential equation solver and some operations on the grid data structure. The precise implementation isn't important here, but all of the code can be found in the "Examples/python/manual" directory of the SWIG distribution.

Making a quick and dirty Python module

Given our simple application, making a Python module is easy. Simply use the following SWIG interface file :

```
// File : pde.i
%module pde
%{
#include "pde.h"
%}

#include pde.h
```

Since `pde.h` is fairly simple, we can simply include it directly into our interface file using `%include`. However, we also need to make sure we also include it in the `%{ , %}` block--otherwise we'll get a huge number of compiler errors when we compile the resulting wrapper file.

To build the module simply run SWIG with the following options

```
swig -python -shadow pde.i
```

and compile using the techniques described in the beginning of this chapter.

Using our new module

We are now ready to use our new module. To do this, we can simply write a Python script like this :

```
# A fairly uninteresting example

from pde import *

h = Heat2d(50,50)          # Creates a new "problem"
```

```
h.set_temp(1.0)
print "Dt = ", h.dt

# Solve something

for i in range(0,25):
    h.solve(100)
    print "time = ", h.time
```

When run, we get rather exciting output such as the following :

```
Dt = 2.5e-05
time = 0.0025
time = 0.005
time = 0.0075
...
time = 0.06
time = 0.0625
```

(okay, it's not that exciting--well, maybe it is if you don't get out much).

While this has only been a simple example it is important to note that we could have just as easily written the same thing in C++. For example :

```
// Python example written in C++

#include "pde.h"
#include <stdio.h>

int main(int argc, char **argv) {

    Heat2d *h;

    h = new Heat2d(50,50);
    printf("Dt = %g\n", h->dt);

    h->set_temp(1.0);

    for (int i = 0; i < 25; i++) {
        h->solve(100);
        printf("time = %g\n", h->time);
    }
}
```

For the most part, the code looks identical (although the Python version is simpler). As for performance, the Python version runs less than 1% slower than the C++ version on my machine. Given that most of the computational work is written in C++, there is very little performance penalty for writing the outer loop of our calculation in Python in this case.

Unfortunately, our Python version suffers a number of drawbacks. Most notably, there is no way for us to access any of the grid data (which is easily accomplished in C++). However, there are ways to fix this :

Accessing array data

Let's modify our heat equation problem so that we can access grid data directly from Python. This can be done by modifying our interface file as follows :

```
%module pde
%{
#include "pde.h"
%}

#include pde.h

// Add a few "helper" functions to extract grid data
%inline %{
double Grid2d_get(Grid2d *g, int i, int j) {
    return g->data[i][j];
}
void Grid2d_set(Grid2d *g, int i, int j, double val) {
    g->data[i][j] = val;
}
%}
```

Rather than modifying our C++ code, it is easy enough to supply a few accessor functions directly in our interface file. These function may only be used from Python so this approach makes sense and it helps us keep our C++ code free from unnecessary clutter. The `%inline` directive is a convenient method for adding helper functions since the functions you declare show up in the interface automatically.

We can now use our accessor functions to write a more sophisticated Python script :

```
# An example using our set/get functions

from pde import *

# Set up an initial condition
def initcond(h):
    h.set_temp(0.0)
    nx = h.grid.xpoints
    for i in range(0,nx):
        Grid2d_set(h.grid,i,0,1.0) # Set grid values

# Dump out to a file
def dump(h,filename):
    f = open(filename,"w")
    nx = h.grid.xpoints
    ny = h.grid.ypoints
    for i in range(0,nx):
        for j in range(0,ny):
            f.write(str(Grid2d_get(h.grid,i,j))+"\n") # Get grid value
    f.close()

# Set up a problem and run it

h = Heat2d(50,50)
initcond(h)
fileno = 1
for i in range(0,25):
```

```

h.solve(100)
dump(h,"Dat"+str(fileno))
print "time = ", h.time
fileno = fileno+1

```

We now have a Python script that can create a grid, set up an initial condition, run a simulation, and dump a collection of datafiles. So, with just a little supporting code in our interface file, we can start to do useful work from Python.

Use Python for control, C for performance

Now that it is possible to access grid data from Python, it is possible to quickly write code for all sorts of operations. However, Python may not provide enough performance for certain operations. For example, the `dump()` function in the previous example may become quite slow as problem sizes increase. Thus, we might consider writing it in C++ such as the follows:

```

void dump(Heat2d *h, char *filename) {
    FILE *f;
    int i,j;

    f = fopen(filename,"w");
    for (i = 0; i < h->grid->xpoints; i++)
        for (j = 0; j < h->grid->yypoints; j++)
            fprintf(f,"%0.17f\n",h->grid->data[i][j]);
    fclose(f);
}

```

To use this new function, simply put its declaration in the SWIG interface file and get rid of the old Python version. The Python script won't know that you changed the implementation.

Getting even more serious about array access

We have provided access to grid data using a pair of get/set functions. However, using these functions is a little clumsy because they always have to be called as a separate function like this :

```
Grid2d_set(grid,i,j,1.0)
```

It might make more sense to make the get/set functions appear like member functions of the `Grid2D` class. That way we could use them like this :

```
grid.set(i,j,1.0)
grid.get(i,j)
```

SWIG provides a simple technique for doing this as illustrated in the following interface file :

```

%module pde
%{
#include "pde.h"
%}
#include pde.h

// Add a few "helper" functions to extract grid data
%{
    double Grid2d_get(Grid2d *g, int i, int j) {

```

```

        return g->data[i][j];
    }
    void Grid2d_set(Grid2d *g, int i, int j, double val) {
        g->data[i][j] = val;
    }
}

// Now add these helper functions as methods of Grid2d

%addmethods Grid2d {
    double get(int i, int j);           // Gets expanded to Grid2d_get()
    void set(int i, int j, double val); // Gets expanded to Grid2d_set()
}

```

The `%addmethods` directive tells SWIG that you want to add new functions to an existing C++ class or C structure for the purposes of building an interface. In reality, SWIG leaves the original C++ class unchanged, but the resulting Python interface will have some new functions that appear to be class members.

SWIG uses a naming convention for adding methods to a class. If you have a class `Foo` and you add a member function `bar(args)`, SWIG will look for a function called `Foo_bar(this, args)` that implements the desired functionality. You can write this function yourself, as in the previous interface file, but you can also just supply the code immediately after a declaration like this :

```

%module pde
%{
#include "pde.h"
%}
#include pde.h

// Add some new accessor methods to the Grid2D class
%addmethods Grid2d {
    double get(int i, int j) {
        return self->data[i][j];
    };
    void set(int i, int j, double val) {
        self->data[i][j] = val;
    };
};

```

In this case, SWIG will take the supplied code, and automatically generate a function for the method. The special variable “`self`” is used to hold a pointer to the corresponding object. The `self` pointer is exactly like the C++ “`this`” pointer, except that the name has been changed in order to remind you that you aren’t really writing a real class member function. (Actually, the real reason we can’t use “`this`” is because the C++ compiler will start complaining!)

Finally, it is worth noting that the `%addmethods` directive may also be used inside a class definition like this :

```

struct Grid2d {
    Grid2d(int ni, int nj);
    ~Grid2d();
    double **data;
};

```

```

int    xpoints;
int    ypoints;
%addmethods {
    double get(int i, int j);
    void    set(int i, int j, double value);
}
};

```

This latter case is really only useful if the C++ class definition is included in the SWIG interface file itself. If you are pulling the class definition out of a separate file or a C++ header file, using a separate `%addmethods` directive is preferable. It doesn't matter if the `%addmethods` directive appears before or after the real class definition--SWIG will correctly associate the two definitions.

Okay, enough talk. By adding the set/get functions as methods, we can now change our Python script to look like this (changes are underlined) :

```

# An example using our new set/get functions

from pde import *

# Set up an initial condition

def initcond(h):
    h.set_temp(0.0)
    nx = h.grid.xpoints
    for i in range(0,nx):
        h.grid.set(i,0,1.0)           # Note changed interface

# Dump out to a file
def dump(h,filename):
    f = open(filename,"w")
    nx = h.grid.xpoints
    ny = h.grid.ypoints
    for i in range(0,nx):
        for j in range(0,ny):
            f.write(str(h.grid.get(i,j))+"\n")
    f.close()

# Set up a problem and run it

h = Heat2d(50,50)
initcond(h)
fileno = 1

for i in range(0,25):
    h.solve(100)
    h.dump("Dat"+str(fileno))
    print "time = ", h.time
    fileno = fileno+1

```

Now it's starting to look a little better, but we can do even better...

Implementing special Python methods in C

Now that you're getting into the spirit of things, let's make it so that we can access our `Grid2D` data like a Python array. As it turns out, we can do this with a little trickery in the SWIG inter-

face file. Don't forget to put on your Python wizard cap...

```
// SWIG interface file with Python array methods added
%module pde
%{
#include "pde.h"
%}

#include pde.h

%inline %{
// Define a new Grid2d row class
struct Grid2dRow {
    Grid2d *g;      // Grid
    int    row;     // Row number
    // These functions are used by Python to access sequence types (lists, tuples, ...)
    double __getitem__(int i) {
        return g->data[row][i];
    };
    void __setitem__(int i, double val) {
        g->data[row][i] = val;
    };
};
%}

// Now add a __getitem__ method to Grid2D to return a row
%addmethods Grid2d {
    Grid2dRow __getitem__(int i) {
        Grid2dRow r;
        r.g = self;
        r.row = i;
        return r;
    };
};
```

We have now replaced our get/set functions with the `__getitem__` and `__setitem__` functions that Python needs to access arrays. We have also added a special `Grid2dRow` class. This is needed to allow us to make a funny kind of “multidimensional” array in Python (this may take a few minutes of thought to figure out). Using this new interface file, we can now write a Python script like this :

```
# An example script using our array access functions

from pde import *

# Set up an initial condition

def initcond(h):
    h.set_temp(0.0)
    nx = h.grid.xpoints
    for i in range(0,nx):
        h.grid[i][0] = 1.0          # Note nice array access

# Set up a problem and run it

h = Heat2d(50,50)
initcond(h)
```

```
fileno = 1

for i in range(0,25):
    h.solve(100)
    h.dump("Dat"+str(fileno))
    print "time = ", h.time
    fileno = fileno+1

# Calculate average temperature over the region

sum = 0.0
for i in range(0,h.grid.xpoints):
    for j in range(0,h.grid.ypoints):
        sum = sum + h.grid[i][j]           # Note nice array access

avg = sum/(h.grid.xpoints*h.grid.ypoints)

print "Avg temperature = ",avg
```

Summary (so far)

In our first example, we have taken a very simple C++ problem and wrapped it into a Python module. With a little extra work, we have been able to provide array type access to our C++ data from Python and to write some computationally intensive operations in C++. At this point, it would be easy to write all sorts of Python scripts to set up problems, run simulations, look at the data, and to debug new operations implemented in C++.

Wrapping a C library

In this next example, we focus on wrapping the gd-1.2 library. gd is a public domain library for fast GIF image creation written by Thomas Boutell and available on the internet. gd-1.2 is copyright 1994,1995, Quest Protein Database Center, Cold Spring Harbor Labs. This example assumes that you have gd-1.2 available, but you can use the ideas here to wrap other kinds of C libraries.

Preparing a module

Wrapping a C library into a Python module usually involves working with the C header files associated with a particular library. In some cases, a header file can be used directly (without modification) with SWIG. Other times, it may be necessary to copy the header file into a SWIG interface file and make a few touch-ups and modifications. In either case, it's usually not too difficult.

To make a module, you can use the following checklist :

- Locate the header files associated with a package
- Look at the contents of the header files to see if SWIG can handle them. In particular, SWIG can not handle excessive use of C preprocessor macros, or non-ANSI C syntax. The best way to identify problems is to simply run SWIG on the file and see what errors (if any) get reported.
- Make a SWIG interface file for your module specifying the name of the module, the appropriate header files, and any supporting documentation that you would like to provide.

- If the header file is clean, simply use SWIG's `%include` directive. If not, paste the header file into your interface file and edit it until SWIG can handle it.
- Clean up the interface by possibly adding supporting code, deleting unnecessary functions, and eliminating clutter.
- Run SWIG and compile.

In the case of the `gd` library, we can simply use the following SWIG interface file :

```
%module gd
%{
#include "gd.h"
%}

%section "gd-1.2",ignore
#include "gd.h"

// These will come in handy later
FILE *fopen(char *, char *);
void fclose(FILE *f);
```

In this file, we first tell SWIG to put all of the `gd` functions in a separate documentation section and to ignore all comments. This usually helps clean up the documentation when working with raw header files. Next, we simply include the contents of "gd.h" directly. Finally, we provide wrappers to `fopen()` and `fclose()` since these will come in handy in our Python interface.

If we give this interface file to SWIG, we will get the following output :

```
% swig -python -shadow -I/usr/local/include gd.i
Generating wrappers for Python
/usr/local/include/gd.h : Line 32. Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 33. Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 34. Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 35. Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 41. Arrays not currently supported (ignored).
/usr/local/include/gd.h : Line 42. Arrays not currently supported (ignored).
%
```

While SWIG was able to handle most of the header file, it also ran into a few unsupported declarations---in this case, a few data structures with array members. However, the warning messages also tell us that these declarations have simply been ignored. Thus, we can choose to continue and build our interface anyways. As it turns out in this case, the ignored declarations are of little or no consequence so we can ignore the warnings.

If SWIG is unable to process a raw header file or if you would like to eliminate the warning messages, you can structure your interface file as follows :

```
%module gd
%{
#include "gd.h"
%}

%section "gd-1.2",ignore
```

```
... paste the contents of gd.h here and remove problems ...

// A few extra support functions

FILE *fopen(char *, char *);
void fclose(FILE *f);
```

This latter option requires a little more work (since you need to paste the contents of `gd.h` into the file and edit it), but is otherwise not much more difficult to do. For highly complex C libraries or header files that go overboard with the C preprocessor, you may need to do this more often.

Using the *gd* module

Now, that we have created a module from the `gd` library, we can use it in Python scripts. The following script makes a simple image of a black background with a white line drawn on it. Notice how we have used our wrapped versions of `fopen()` and `fclose()` to create a `FILE` handle for use in the `gd` library (there are also ways to use Python file objects, but this is described later).

```
# Simple gd program

from gd import *

im = gdImageCreate(64,64)
black = gdImageColorAllocate(im,0,0,0)
white = gdImageColorAllocate(im,255,255,255)
gdImageLine(im,0,0,63,63,white)
out = fopen("test.gif","w")
gdImageGif(im,out)
fclose(out)
gdImageDestroy(im)
```

That was simple enough--and it only required about 5 minutes of work. Unfortunately, our `gd` module still has a few problems...

Extending and fixing the *gd* module

While our first attempt at wrapping `gd` works for simple functions, there are a number of problems. For example, the `gd-1.2` library contains the following function for drawing polygons :

```
void gdImagePolygon(gdImagePtr im, gdPointPtr points, int pointsTotal, int color);
```

The `gdImagePtr` type is created by another function in our module and the parameters `pointsTotal` and `color` are simple integers. However, the 2nd argument is a pointer to an array of points as defined by the following data structure in the `gd-1.2` header file :

```
typedef struct {
    int x, y;
} gdPoint, *gdPointPtr;
```

Unfortunately, there is no way to create a `gdPoint` in Python and consequently no way to call the `gdImagePolygon` function. A temporary setback, but one that is not difficult to solve using the `%addmethods` directive as follows :

```

%module gd
%{
#include "gd.h"
%}

#include "gd.h"

// Fix up the gdPoint structure a little bit
%addmethods gdPoint {
  // Constructor to make an array of "Points"
  gdPoint(int npoints) {
    return (gdPoint *) malloc(npoints*sizeof(gdPoint));
  };
  // Destructor to destroy this array
  ~gdPoint() {
    free(self);
  };
  // Python method for array access
  gdPoint *__getitem__(int i) {
    return self+i;
  };
};

FILE *fopen(char *, char *);
void fclose(FILE *f);

```

With these simple additions, we can now create arrays of points and use the polygon function as follows :

```

# Simple gd program

from gd import *

im = gdImageCreate(64,64)
black = gdImageColorAllocate(im,0,0,0)
white = gdImageColorAllocate(im,255,255,255)

pts = gdPoint(3);                # Create an array of Points
pts[0].x,pts[0].y = (5,5)        # Assign a set of points
pts[1].x,pts[1].y = (60,25)
pts[2].x,pts[2].y = (16,60)

gdImagePolygon(im,pts,3,white)   # Draw a polygon from our array of points
out = fopen("test.gif","w")
gdImageGif(im,out)
fclose(out)
gdImageDestroy(im)

```

Building a simple 2D imaging class

Now it's time to get down to business. Using our gd-1.2 module, we can write a simple 2D imaging class that hides alot of the underlying details and provides scaling, translations, and a host of other operations. (It's a fair amount code, but an interesting example of how one can take a simple C library and turn it into something that looks completely different).

```
# image.py
# Generic 2D Image Class
#
# Built using the 'gd-1.2' library by Thomas Boutell
#

import gd

class Image:
    def __init__(self,width,height,xmin=0.0,ymin=0.0,xmax=1.0,ymax=1.0):
        self.im = gd.gdImageCreate(width,height)
        self.xmin = xmin
        self.ymin = ymin
        self.xmax = xmax
        self.ymax = ymax
        self.width = width
        self.height = height
        self.dx = 1.0*(xmax-xmin)
        self.dy = 1.0*(ymax-ymin)
        self.xtick = self.dx/10.0
        self.ytick = self.dy/10.0
        self.ticklen= 3
        self.name = "image.gif"
        gd.gdImageColorAllocate(self.im,0,0,0) # Black
        gd.gdImageColorAllocate(self.im,255,255,255) # White
        gd.gdImageColorAllocate(self.im,255,0,0) # Red
        gd.gdImageColorAllocate(self.im,0,255,0) # Green
        gd.gdImageColorAllocate(self.im,0,0,255) # Blue

    def __del__(self):
        print "Deleting"
        gd.gdImageDestroy(self.im)

    # Dump out this image to a file
    def write(self,name="NONE"):
        if name == "NONE":
            name = self.name
        f = gd.fopen(name,"w")
        gd.gdImageGif(self.im,f)
        gd.fclose(f)
        self.name = name

    # Virtual method that derived classes define
    def draw(self):
        print "No drawing method specified."

    # A combination of write and draw
    def show(self,filename="NONE"):
        self.draw()
        self.write(filename)

    # Load up a colormap from a Python array of (R,G,B) tuples
    def colormap(self, cmap):
        for i in range(0,255):
            gd.gdImageColorDeallocate(self.im,i)
        for c in cmap:
            gd.gdImageColorAllocate(self.im,c[0],c[1],c[2])

    # Change viewing region
```

```
def region(self,xmin,ymin,xmax,ymax):
    self.xmin = xmin
    self.ymin = ymin
    self.xmax = xmax
    self.ymax = ymax
    self.dx    = 1.0*(xmax-xmin)
    self.dy    = 1.0*(ymax-ymin)

# Transforms a 2D point into screen coordinates
def transform(self,x,y):
    npt = []
    ix = (x-self.xmin)/self.dx*self.width + 0.5
    iy = (self.ymax-y)/self.dy*self.height + 0.5
    return (ix,iy)

# A few graphics primitives
def clear(self,color):
    gd.gdImageFilledRectangle(self.im,0,0,self.width,self.height,color)

def plot(self,x,y,color):
    ix,iy = self.transform(x,y)
    gd.gdImageSetPixel(self.im,ix,iy,color)

def line(self,x1,y1,x2,y2,color):
    ix1,iy1 = self.transform(x1,y1)
    ix2,iy2 = self.transform(x2,y2)
    gd.gdImageLine(self.im,ix1,iy1,ix2,iy2,color)

def box(self,x1,y1,x2,y2,color):
    ix1,iy1 = self.transform(x1,y1)
    ix2,iy2 = self.transform(x2,y2)
    gd.gdImageRectangle(self.im,ix1,iy1,ix2,iy2,color)

def solidbox(self,x1,y1,x2,y2,color):
    ix1,iy1 = self.transform(x1,y1)
    ix2,iy2 = self.transform(x2,y2)
    gd.gdImageFilledRectangle(self.im,ix1,iy1,ix2,iy2,color)

def arc(self,cx,cy,w,h,s,e,color):
    ix,iy = self.transform(cx,cy)
    iw = (x - self.xmin)/self.dx * self.width
    ih = (y - self.ymin)/self.dy * self.height
    gd.gdImageArc(self.im,ix,iy,iw,ih,s,e,color)

def fill(self,x,y,color):
    ix,iy = self.transform(x,y)
    gd.gdImageFill(self,ix,iy,color)

def axis(self,color):
    self.line(self.xmin,0,self.xmax,0,color)
    self.line(0,self.ymin,0,self.ymax,color)
    x = -self.xtick*(int(-self.xmin/self.xtick)+1)
    while x <= self.xmax:
        ix,iy = self.transform(x,0)
        gd.gdImageLine(self.im,ix,iy-self.ticklen,ix,iy+self.ticklen,color)
        x = x + self.xtick
    y = -self.ytick*(int(-self.ymin/self.ytick)+1)
    while y <= self.ymax:
        ix,iy = self.transform(0,y)
```

```
        gd.gdImageLine(self.im,ix-self.ticklen,iy,ix+self.ticklen,iy,color)
        y = y + self.ytick

# scalex(s). Scales the x-axis. s is given as a scaling factor
def scalex(self,s):
    xc = self.xmin + self.dx/2.0
    dx = self.dx*s
    xmin = xc - dx/2.0
    xmax = xc + dx/2.0
    self.region(xmin,self.ymin,xmax,self.ymax)

# scaley(s). Scales the y-axis.
def scaley(self,s):
    yc = self.ymin + self.dy/2.0
    dy = self.dy*s
    ymin = yc - dy/2.0
    ymax = yc + dy/2.0
    self.region(self.xmin,ymin,self.xmax,ymax)

# Zooms a current image. s is given as a percent
def zoom(self,s):
    s = 100.0/s
    self.scalex(s)
    self.scaley(s)

# Move image left. s is given in range 0,100. 100 moves a full screen left
def left(self,s):
    dx = self.dx*s/100.0
    xmin = self.xmin + dx
    xmax = self.xmax + dx
    self.region(xmin,self.ymin,xmax,self.ymax)

# Move image right. s is given in range 0,100. 100 moves a full screen right
def right(self,s):
    self.left(-s)

# Move image down. s is given in range 0,100. 100 moves a full screen down
def down(self,s):
    dy = self.dy*s/100.0
    ymin = self.ymin + dy
    ymax = self.ymax + dy
    self.region(self.xmin,ymin,self.xmax,ymax)

# Move image up. s is given in range 0,100. 100 moves a full screen up
def up(self,s):
    self.down(-s)

# Center image
def center(self,x,y):
    self.right(50-x)
    self.up(50-y)
```

Our image class provides a number of methods for creating images, plotting points, making lines, and other graphical objects. We have also provided some methods for moving and scaling the image. Now, let's use this image class to do some interesting things :

A mathematical function plotter

Here's a simple class that can be used to plot mathematical functions :

```
# funcplot.py

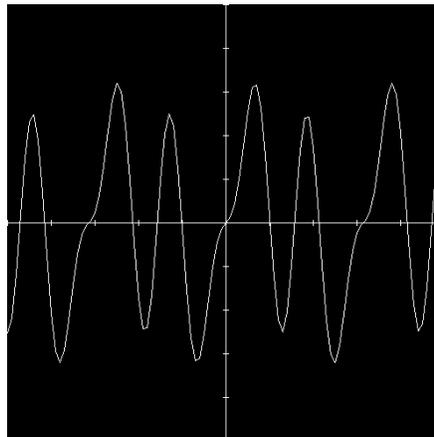
from image import *

class PlotFunc(Image):
    def __init__(self, func, xmin, ymin, xmax, ymax, width=500, height=500):
        Image.__init__(self, width, height, xmin, ymin, xmax, ymax)
        self.func = func          # The function being plotted
        self.npoints = 100       # Number of samples
        self.color = 1
    def draw(self):
        self.clear(0)
        lastx = self.xmin
        lasty = self.func(lastx)
        dx = 1.0*(self.xmax-self.xmin)/self.npoints
        x = lastx+dx
        for i in range(0, self.npoints):
            y = self.func(x)
            self.line(lastx, lasty, x, y, self.color)
            lastx = x
            lasty = y
            x = x + dx
        self.axis(1)
```

Most of the functionality is implemented in our base image class so this is pretty simple. However, if we wanted to make a GIF image of a mathematical function, we could just do this :

```
>>> from funcplot import *
>>> import math
>>> p = PlotFunc(lambda x: 0.5*math.sin(x)+0.75*math.sin(2*x)-0.6*math.sin(3*x),
                -10, -2, 10, 2)
>>> p.show("plot.gif")
```

Which would produce the following GIF image :



Plotting an unstructured mesh

Of course, perhaps we want to plot something a little more complicated like a mesh. Recently, a colleague came to me with some unstructured mesh data contained in a pair of ASCII formatted files. These files contained a collection of points, and a list of connectivities defining a mesh on these points. Reading and plotting this data in Python turned out to be relatively easy using the following script and our image base class :

```
# plotmesh.py
# Plots an unstructured mesh stored as an ASCII file
from image import *
import string

class PlotMesh(Image):
    def __init__(self,filename,xmin,ymin,xmax,ymax,width=500,height=500):
        Image.__init__(self,width,height,xmin,ymin,xmax,ymax)
        # read in a mesh file in pieces
        pts = []
        # Read in data points
        atoi = string.atoi
        atof = string.atof
        f = open(filename+".pts","r")
        npoints = atoi(f.readline())
        for i in range(0,npoints):
            l = string.split(f.readline())
            pts.append((atof(l[0]),atof(l[1])))
        f.close()

        # Read in mesh data
        f = open(filename+".tris","r")
        ntris = string.atoi(f.readline())
        tris = [ ]
        for i in range(0,ntris):
            l = string.split(f.readline())
            tris.append((atoi(l[0])-1,atoi(l[1])-1,atoi(l[2])-1,atoi(l[3])))
        f.close()

    # Set up local attributes
    self.pts = pts
    self.npoints = npoints
    self.tris = tris
    self.ntris = ntris

    # Draw mesh
    def draw(self):
        self.clear(0);
        i = 0
        while i < self.ntris:
            tri = self.tris[i]
            pt1 = self.pts[tri[0]]
            pt2 = self.pts[tri[1]]
            pt3 = self.pts[tri[2]]
            # Now draw the mesh
            self.triangle(pt1[0],pt1[1],pt2[0],pt2[1],pt3[0],pt3[1],tri[3]);
            i = i + 1

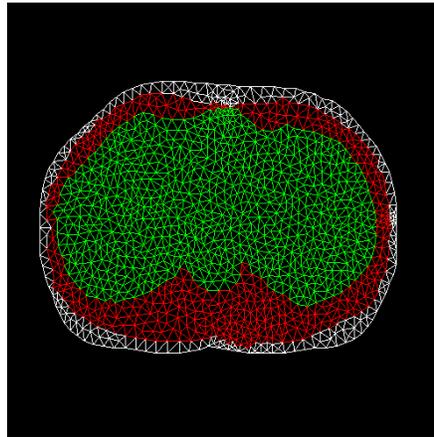
    # Draw a triangle
    def triangle(self,x1,y1,x2,y2,x3,y3,color):
```

```
self.line(x1,y1,x2,y2,color)
self.line(x2,y2,x3,y3,color)
self.line(x3,y3,x1,y1,color)
```

This class simply reads the data into a few Python lists, has a drawing function for making a plot, and adds a special method for making triangles. Making a plot is now easy, just do this :

```
>>> from plotmesh.py import *
>>> mesh = PlotMesh("mesh",5,0,35,25)
>>> mesh.show("mesh.gif")
```

This produces the following GIF image :



When run interactively, we can also use simple commands to zoom in and move the image around. For example :

```
>>> mesh = PlotMesh("mesh",5,0,35,25)
>>> mesh.zoom(200)           # Enlarge by 200%
>>> mesh.left(50)           # Move image half a screen to left
>>> mesh.show()
>>>
```

While a Python-only implementation would be unsuitable for huge datasets, performance critical operations could be moved to C and used in conjunction with our Image base class.

From C to SWIG to Python

This example illustrates a number of things that are possible with SWIG and Python. First, it is usually relatively easy to build a Python interface to an existing C library. With a little extra work, it is possible to improve the interface by adding a few support functions such as our Point extensions. Finally, once in Python, it is possible to encapsulate C libraries in new kinds of Python objects and classes. We built a simple Image base class and used it to plot mathematical functions and unstructured 2D mesh data---two entirely different tasks, yet easily accomplished with a small amount of Python code. If we later decided to use a different C library such as OpenGL, we could wrap it in SWIG, change the Image base class appropriately, and use the function and mesh plotting examples without modification. I think this is pretty cool.

Putting it all together

Finally, let's combine our heat equation solver and graphics module into a single application. To do this, we first need to know how to combine two different SWIG generated modules. When different SWIG modules need to be combined, there are a number of things you can do.

Merging modules

Two SWIG modules can be combined into a single module if you make an interface file like this :

```
%module package
#include pde.i
#include gd.i
```

This will combine everything in both interface files into a single super-module called "package". The advantage to this approach is that it is extremely quick and easy. The disadvantage is that the module names of "pde" and "gd" will be lost. If you had a bunch of scripts that relied on those names, they would no longer work. Thus, combining modules in this way is probably only a good idea if the modules are closely related.

Using dynamic loading

If your system supports dynamic loading, you can build each SWIG module into a separate dynamically loadable module and load each one individually into Python. This is the preferred approach if it is supported on your system. SWIG wrapper files declare virtually everything as "static" so using dynamic loading with multiple SWIG generated modules will not usually cause any namespace clashes.

Use static linking

As an alternative to dynamic loading, you can use a special form of the `%module` directive as follows :

```
%module package, pdec, gdc
#include embed.i
```

This will build a static version of Python with 3 C extension modules added (`package`, `pdec`, and `gdc`). When using this technique, the names of the modules refer to the low-level SWIG generated C/C++ modules. Since shadow classes are being used, these modules must have an extra 'c' appended to the name (thus, "pdec" and "gdc" instead of "pde" and "gd"). The extra modules specified with the `%modules` directive do not necessarily have to be SWIG-generated modules. In practice, almost any kind of Python module can be listed here. It should also be noted that extra modules names are completely ignored if the `embed.i` library file is not used.

Building large multi-module systems

By default, SWIG includes the C code for the SWIG type-checker and variable linking into every module. However, when, building systems involving large numbers of SWIG modules, common code such as the SWIG pointer type-checker and variable linking extensions can be shared if you run SWIG with the `-c` option. For example :

```
% swig -c -python graphics.i
% swig -c -python network.i
```

```
% swig -c -python analysis.i
% swig -c -python math.i
% gcc -c graphics_wrap.c network_wrap.c analysis_wrap.c math_wrap.c
% ld -shared graphics_wrap.o -lswigpy -o graphicsmodule.so
% ld -shared network_wrap.o -lswigpy -o networkmodule.so
% ld -shared analysis_wrap.o -lswigpy -o analysismodule.so
% ld -shared math_wrap.o -o -lswigpy mymathmodule.so
```

swigpy is a special purpose library that contains the SWIG pointer type checker and other support code (see the `Misc` subdirectory of the SWIG distribution). When used in this manner, the same support code will be used for all of the modules. The `swigpy` library can also be applied when static linking is being used. See the `Advanced Topics` chapter for more information about using SWIG with multiple modules.

A complete application

The following Python script shows an application that combines our C++ heat equation solver, our `gd` library, and our `Image` base class that we developed.

```
# Solve the heat equation.
# Make a series of data files
# Make a movie of GIF images

from pde import *
from image import *
import string

# Image class
class HeatImg(Image):
    def __init__(self,h,width=300,height=300):
        Image.__init__(self,width,height,0.0,0.0,1.0,1.0)
        self.h = h
        # Create a greyscale colormap
        cmap = []
        for i in range(0,255):
            cmap.append((i,i,i))
        self.colormap(cmap)
        self.cmin = 0.0
        self.cmax = 1.0
        self.imgno = 1
    def draw(self):
        self.clear(0)
        dx = 1.0/(self.h.grid.xpoints-2)
        dy = 1.0/(self.h.grid.ypoints-2)
        i = 1
        x = 0.0
        while i < self.h.grid.xpoints:
            j = 1;
            y = 0.0
            while j < self.h.grid.ypoints:
                c = int((self.h.grid[i][j]-self.cmin)/(self.cmax-
                    self.cmin)*255)
                self.solidbox(x,y+dy,x+dx,y,c)
                j = j + 1
                y = y + dy
            i = i + 1
            x = x + dx
```

```
        self.name = "image"+string.zfill(self.imgno,4)+".gif"
        self.imgno = self.imgno+1

# Set up an initial condition
def initcond(h):
    h.set_temp(0.0)
    nx = h.grid.xpoints
    for i in range(0,nx):
        h.grid[i][0] = 1.0

# Set up a problem and run it
h = Heat2d(50,50)

# Make an image object
img = HeatImg(h)

initcond(h)
fileno = 1

# Run it
for i in range(0,25):
    h.solve(100)
    h.dump("Dat"+str(fileno))
    img.show()
    print "time = ", h.time
    fileno = fileno+1

# Calculate average temperature and exit
sum = 0.0
for i in range(0,h.grid.xpoints):
    for j in range(0,h.grid.ypoints):
        sum = sum + h.grid[i][j]
avg = sum/(h.grid.xpoints*h.grid.ypoints)
print "Avg temperature = ",avg
```

When run, we now get a collection of datafiles and series of images like this :



Thus, we have a simple physics application that only takes about 1 page of Python code, runs a simulation, creates data files, and a movie of images. We can easily change any aspect of the simulation, interactively query variables and examine data. New procedures can be written and tested in Python and later implemented in C++ if needed. More importantly, we have an application that is actually fun to use and modify (well, at least I think so).

Exception handling

The SWIG `%except` directive can be used to create a user-definable exception handler in charge

of converting exceptions in your C/C++ program into Python exceptions. The chapter on exception handling contains more details, but suppose you have a C++ class like the following :

```
class RangeError {};     // Used for an exception

class DoubleArray {
private:
    int n;
    double *ptr;
public:
    // Create a new array of fixed size
    DoubleArray(int size) {
        ptr = new double[size];
        n = size;
    }
    // Destroy an array
    ~DoubleArray() {
        delete ptr;
    }
    // Return the length of the array
    int length() {
        return n;
    }

    // Get an item from the array and perform bounds checking.
    double getitem(int i) {
        if ((i >= 0) && (i < n))
            return ptr[i];
        else
            throw RangeError();
    }

    // Set an item in the array and perform bounds checking.
    void setitem(int i, double val) {
        if ((i >= 0) && (i < n))
            ptr[i] = val;
        else {
            throw RangeError();
        }
    }
};
```

The functions associated with this class can throw a range exception for an out-of-bounds array access. We can catch this in our Python extension by specifying the following in an interface file :

```
%except(python) {
    try {
        $function
    }
    catch (RangeError) {
        PyErr_SetString(PyExc_IndexError, "index out-of-bounds");
        return NULL;
    }
}
```

When the C++ class throws a RangeError exception, our wrapper functions will catch it, turn it

into a Python exception, and allow a graceful death as opposed to just having some sort of mysterious program crash. Since SWIG's exception handling is user-definable, we are not limited to C++ exception handling. Please see the chapter on exception handling for more details and using the `exception.i` library for writing language-independent exception handlers.

Python exceptions can be raised using the `PyErr_SetString()` function as shown above. The following table provides a list of the different Python exceptions available.

Built-in Python Exceptions

<code>PyExc_AttributeError</code>	Raised when an attribute reference or assignment fails. Usually raised when an invalid class attribute is accessed.
<code>PyExc_EOFError</code>	Indicates the end-of-file condition for I/O operations.
<code>PyExc_IOError</code>	Raised when an I/O occurs, e.g. 'file not found', 'permission denied', etc...
<code>PyExc_ImportError</code>	Raised when an 'import' statement fails.
<code>PyExc_IndexError</code>	Indicates a subscript out of range--usually for array and list indexing.
<code>PyExc_KeyError</code>	Raised when a dictionary key is not found in the set of existing keys. Could be used for hash tables and similar objects.
<code>PyExc_KeyboardInterrupt</code>	Raised when the user hits the interrupt key.
<code>PyExc_MemoryError</code>	Indicates a recoverable out of memory error.
<code>PyExc_NameError</code>	Raised to indicate a name not found.
<code>PyExc_OverflowError</code>	Raised when results of arithmetic computation is too large to be represented.
<code>PyExc_RuntimeError</code>	A generic exception for "everything else". Errors that don't fit into other categories.
<code>PyExc_SyntaxError</code>	Normally raised when the Python parser encounters a syntax error.
<code>PyExc_SystemError</code>	Used to indicate various system errors.
<code>PyExc_SystemExit</code>	A serious error, abandon all hope now.
<code>PyExc_TypeError</code>	Raised when an object is of invalid type.
<code>PyExc_ValueError</code>	Raised when an object has the right type, but an inappropriate value.
<code>PyExc_ZeroDivisionError</code>	Division by zero error.

Remapping C datatypes with typemaps

This section describes how SWIG's treatment of various C/C++ datatypes can be remapped using the SWIG `%typemap` directive. While not required, this section assumes some familiarity with Python's C API. The reader is advised to consult the Python reference manual or one of the books on Python. A glance at the chapter on SWIG typemaps will also be useful.

What is a typemap?

A typemap is mechanism by which SWIG's processing of a particular C datatype can be overridden. A simple typemap might look like this :

```
%module example

%typemap(python,in) int {
    $target = (int) PyLong_AsLong($source);
    printf("Received an integer : %d\n",$target);
}
extern int fact(int n);
```

Typemaps require a language name, method name, datatype, and conversion code. For Python, "python" should be used as the language name. The "in" method in this example refers to an input argument of a function. The datatype 'int' tells SWIG that we are remapping integers. The supplied code is used to convert from a `PyObject *` to the corresponding C datatype. Within the supporting C code, the variable `$source` contains the source data (the `PyObject` in this case) and `$target` contains the destination of a conversion.

When this example is compiled into a Python module, it will operate as follows :

```
>>> from example import *
>>> fact(6)
Received an integer : 6
720
```

A full discussion of typemaps can be found in the main SWIG users reference. We will primarily be concerned with Python typemaps here.

Python typemaps

The following typemap methods are available to Python modules :

<code>%typemap(python,in)</code>	Converts Python objects to input function arguments
<code>%typemap(python,out)</code>	Converts return value of a C function to a Python object
<code>%typemap(python,varin)</code>	Assigns a global variable from a Python object
<code>%typemap(python,varout)</code>	Returns a global variable as a Python object
<code>%typemap(python,freearg)</code>	Cleans up a function argument (if necessary)
<code>%typemap(python,argout)</code>	Output argument processing
<code>%typemap(python,ret)</code>	Cleanup of function return values
<code>%typemap(python,const)</code>	Creation of Python constants
<code>%typemap(memberin)</code>	Setting of C++ member data
<code>%typemap(memberout)</code>	Return of C++ member data
<code>%typemap(python,check)</code>	Checks function input values.

Typemap variables

The following variables may be used within the C code used in a typemap:

<code>\$source</code>	Source value of a conversion
<code>\$target</code>	Target of conversion (where the result should be stored)
<code>\$type</code>	C datatype being remapped
<code>\$mangle</code>	Mangled version of data (used for pointer type-checking)
<code>\$value</code>	Value of a constant (const typemap only)

Name based type conversion

Typemaps are based both on the datatype and an optional name attached to a datatype. For example :

```
%module foo

// This typemap will be applied to all char ** function arguments
%typemap(python,in) char ** { ... }

// This typemap is applied only to char ** arguments named 'argv'
%typemap(python,in) char **argv { ... }
```

In this example, two typemaps are applied to the `char **` datatype. However, the second typemap will only be applied to arguments named `'argv'`. A named typemap will always override an unnamed typemap.

Due to the name-based nature of typemaps, it is important to note that typemaps are independent of typedef declarations. For example :

```
%typemap(python, in) double {
    ... get a double ...
}
void foo(double);           // Uses the above typemap
typedef double Real;
void bar(Real);           // Does not use the above typemap (double != Real)
```

To get around this problem, the `%apply` directive can be used as follows :

```
%typemap(python,in) double {
    ... get a double ...
}
void foo(double);

typedef double Real;           // Uses typemap
%apply double { Real };       // Applies all "double" typemaps to Real.
void bar(Real);               // Now uses the same typemap.
```

Converting Python list to a char **

A common problem in many C programs is the processing of command line arguments, which are usually passed in an array of NULL terminated strings. The following SWIG interface file allows a Python list object to be used as a `char **` object.

```
%module argv
```

```
// This tells SWIG to treat char ** as a special case
%typemap(python,in) char ** {
  /* Check if is a list */
  if (PyList_Check($source)) {
    int size = PyList_Size($source);
    int i = 0;
    $target = (char **) malloc((size+1)*sizeof(char *));
    for (i = 0; i < size; i++) {
      PyObject *o = PyList_GetItem($source,i);
      if (PyString_Check(o))
        $target[i] = PyString_AsString(PyList_GetItem($source,i));
      else {
        PyErr_SetString(PyExc_TypeError,"list must contain strings");
        free($target);
        return NULL;
      }
    }
    $target[i] = 0;
  } else {
    PyErr_SetString(PyExc_TypeError,"not a list");
    return NULL;
  }
}

// This cleans up the char ** array we malloc'd before the function call
%typemap(python,freearg) char ** {
  free((char *) $source);
}

// This allows a C function to return a char ** as a Python list
%typemap(python,out) char ** {
  int len,i;
  len = 0;
  while ($source[len]) len++;
  $target = PyList_New(len);
  for (i = 0; i < len; i++) {
    PyList_SetItem($target,i,PyString_FromString($source[i]));
  }
}

// Now a few test functions
%inline %{
int print_args(char **argv) {
  int i = 0;
  while (argv[i]) {
    printf("argv[%d] = %s\n", i,argv[i]);
    i++;
  }
  return i;
}

// Returns a char ** list

char **get_args() {
  static char *values[] = { "Dave", "Mike", "Susan", "John", "Michelle", 0};
  return &values[0];
}
%}
```

When this module is compiled, our wrapped C functions now operate as follows :

```
>>> from argv import *
>>> print_args(["Dave", "Mike", "Mary", "Jane", "John"])
argv[0] = Dave
argv[1] = Mike
argv[2] = Mary
argv[3] = Jane
argv[4] = John
5
>>> get_args()
['Dave', 'Mike', 'Susan', 'John', 'Michelle']
>>>
```

Our type-mapping makes the Python interface to these functions more natural and easy to use.

Converting a Python file object to a FILE *

In our previous example involving gd-1.2, we had to write wrappers around `fopen()` and `fclose()` so that we could provide gd with a `FILE *` pointer. However, we could have used a `typemap` like this instead :

```
// Type mapping for grabbing a FILE * from Python

%typemap(python,in) FILE * {
  if (!PyFile_Check($source)) {
    PyErr_SetString(PyExc_TypeError, "Need a file!");
    return NULL;
  }
  $target = PyFile_AsFile($source);
}
```

Now, we can rewrite one of our earlier examples like this :

```
# Simple gd program

from gd import *

im = gdImageCreate(64,64)
black = gdImageColorAllocate(im,0,0,0)
white = gdImageColorAllocate(im,255,255,255)
gdImageLine(im,0,0,63,63,white)
f = open("test.gif","w")           # Create a Python file object
gdImageGif(im,f)                   # Pass to a C function as FILE *
f.close()
gdImageDestroy(im)
```

Using typemaps to return arguments

A common problem in some C programs is that values may be returned in arguments rather than in the return value of a function. For example :

```
/* Returns a status value and two values in out1 and out2 */
int spam(double a, double b, double *out1, double *out2) {
    ... Do a bunch of stuff ...
}
```

```

        *out1 = result1;
        *out2 = result2;
        return status;
};

```

A named typemap can be used to handle this case as follows :

```

%module outarg

// This tells SWIG to treat an double * argument with name 'OutValue' as
// an output value. We'll append the value to the current result which
// is guaranteed to be a List object by SWIG.

%typemap(python,argout) double *OutValue {
    PyObject *o;
    o = PyFloat_FromDouble(*$source);
    if ((!$target) || ($target == Py_None)) {
        $target = o;
    } else {
        if (!PyList_Check($target)) {
            PyObject *o2 = $target;
            $target = PyList_New(0);
            PyList_Append($target,o2);
            Py_XDECREF(o2);
        }
        PyList_Append($target,o);
        Py_XDECREF(o);
    }
}

int spam(double a, double b, double *OutValue, double *OutValue);

```

With this typemap, we first check to see if any result exists. If so, we turn it into a list and append our new output value to it. If this is the only result, we simply return it normally. For our sample function, there are three output values so the function will return a list of 3 elements. As written, our function needs to take 4 arguments, the last two being pointers to doubles. We may not want to pass anything into these arguments if they are only used to hold output values so we could change this as follows :

```

%typemap(python,ignore) double *OutValue(double temp) {
    $target = &temp; /* Assign the pointer to a local variable */
}

```

Now, in a Python script, we could do this :

```

>>> a = spam(4,5)
>>> print a
[0, 2.45, 5.0]
>>>

```

Mapping Python tuples into small arrays

In some applications, it is sometimes desirable to pass small arrays of numbers as arguments. For example :

```

extern void set_direction(double a[4]); /* Set direction vector

```

This too, can be handled used typemaps as follows :

```
// Grab a 4 element array as a Python 4-tuple
%typemap(python,in) double[4](double temp[4]) { // temp[4] becomes a local variable
    int i;
    if (PyTuple_Check($source)) {
        if (!PyArg_ParseTuple($source,"dddd",temp,temp+1,temp+2,temp+3)) {
            PyErr_SetString(PyExc_TypeError,"tuple must have 4 elements");
            return NULL;
        }
        $target = &temp[0];
    } else {
        PyErr_SetString(PyExc_TypeError,"expected a tuple.");
        return NULL;
    }
}
```

This allows our `set_direction` function to be called from Python as follows :

```
>>> set_direction((0.5,0.0,1.0,-0.25))
```

Since our mapping copies the contents of a Python tuple into a C array, such an approach would not be recommended for huge arrays, but for small structures, this kind of scheme works fine.

Accessing array structure members

Consider the following data structure :

```
#define NAMELEN 32
typedef struct {
    char name[NAMELEN];
    ...
} Person;
```

By default, SWIG doesn't know how to handle the name structure since it's an array, not a pointer. In this case, SWIG will make the array member readonly. However, member typemaps can be used to make this member writable from Python as follows :

```
%typemap(memberin) char[NAMELEN] {
    /* Copy at most NAMELEN characters into $target */
    strncpy($target,$source,NAMELEN);
}
```

Whenever a `char[NAMELEN]` type is encountered in a structure or class, this typemap provides a safe mechanism for setting its value. An alternative implementation might choose to print an error message if the name was too long to fit into the field.

It should be noted that the `[NAMELEN]` array size is attached to the typemap. A datatype involving some other kind of array would not be affected. However, you can write a typemap to match any sized array using the `ANY` keyword as follows :

```
%typemap(memberin) char [ANY] {
    strncpy($target,$source,$dim0);
}
```

During code generation, `$dim0` will be filled in with the real array dimension.

Useful Functions

When writing typemaps, it is often necessary to work directly with Python objects instead of using the conventional `PyArg_ParseTuple()` function that is usually used when writing Python extensions. However, there are a number of useful Python functions available for you to use.

Python Integer Conversion Functions

<code>PyObject *PyInt_FromLong(long l)</code>	Convert long to Python integer
<code>long PyInt_AsLong(PyObject *)</code>	Convert Python integer to long
<code>int PyInt_Check(PyObject *)</code>	Check if Python object is an integer

Python Floating Point Conversion Functions

<code>PyObject *PyFloat_FromDouble(double)</code>	Convert a double to a Python float
<code>double PyFloat_AsDouble(PyObject *)</code>	Convert Python float to a double
<code>int PyFloat_Check(PyObject *)</code>	Check if Python object is a float

Python String Conversion Functions

<code>PyObject *PyString_FromString(char *)</code>	Convert NULL terminated ASCII string to a Python string
<code>PyObject *PyString_FromStringAndSize(char *, int len)</code>	Convert a string and length into a Python string. May contain NULL bytes.
<code>int PyString_Size(PyObject *)</code>	Return length of a Python string
<code>char *PyString_AsString(PyObject *)</code>	Return Python string as a NULL terminated ASCII string.
<code>int PyString_Check(PyObject *)</code>	Check if Python object is a string

Python List Conversion Functions

<code>PyObject *PyList_New(int size)</code>	Create a new list object
<code>int PyList_Size(PyObject *list)</code>	Get size of a list
<code>PyObject *PyList_GetItem(PyObject *list, int i)</code>	Get item <code>i</code> from list
<code>int PyList_SetItem(PyObject *list, int i, PyObject *item)</code>	Set <code>list[i]</code> to item.
<code>int PyList_Insert(PyObject *list, int i, PyObject *item)</code>	Inserts item at <code>list[i]</code> .

Python List Conversion Functions

<code>int PyList_Append(PyObject *list, PyObject *item)</code>	Appends item to list
<code>PyObject *PyList_GetSlice(PyObject *list, int i, int j)</code>	Returns list[i:j]
<code>int PyList_SetSlice(PyObject *list, int i, int j, PyObject *list2)</code>	Sets list[i:j] = list2
<code>int PyList_Sort(PyObject *list)</code>	Sorts a list
<code>int PyList_Reverse(PyObject *list)</code>	Reverses a list
<code>PyObject *PyList_AsTuple(PyObject *list)</code>	Converts a list to a tuple
<code>int PyList_Check(PyObject *)</code>	Checks if an object is a list

Python Tuple Functions

<code>PyObject *PyTuple_New(int size)</code>	Create a new tuple
<code>int PyTuple_Size(PyObject *t)</code>	Get size of a tuple
<code>PyObject *PyTuple_GetItem(PyObject *t, int i)</code>	Get object t[i]
<code>int PyTuple_SetItem(PyObject *t, int i, PyObject *item)</code>	Set t[i] = item
<code>PyObject *PyTuple_GetSlice(PyObject *t, int i int j)</code>	Get slice t[i:j]
<code>int PyTuple_Check(PyObject *)</code>	Check if an object is a tuple

Python File Conversions

<code>PyObject *PyFile_FromFile(FILE *f)</code>	Convert a FILE * to a Python file object
<code>FILE *PyFile_AsFile(PyObject *)</code>	Return FILE * from a Python object
<code>int PyFile_Check(PyObject *)</code>	Check if an object is a file

Standard typemaps

The following typemaps show how to convert a few common kinds of objects between Python and C (and to give a better idea of how typemaps work)

Function argument typemaps

int, short, long	<pre>%typemap(python,in) int,short,long { if (!PyInt_Check(\$source)) { PyErr_SetString(PyExc_TypeError,"not an integer"); return NULL; } \$target = (\$type) PyInt_AsLong(\$source); }</pre>
float, double	<pre>%typemap(python,in) float,double { if (!PyFloat_Check(\$source)) { PyErr_SetString(PyExc_TypeError,"not a float"); return NULL; } \$target = (\$type) PyFloat_AsDouble(\$source); }</pre>
char *	<pre>%typemap(python,in) char * { if (!PyString_Check(\$source)) { PyErr_SetString(PyExc_TypeError,"not a string"); return NULL; } \$target = PyString_AsString(\$source); }</pre>
FILE *	<pre>%typemap(python,in) FILE * { if (!PyFile_Check(\$source)) { PyErr_SetString(PyExc_TypeError,"not a file"); return NULL; } \$target = PyFile_AsFile(\$source); }</pre>

Function return typemaps

int, short	<pre>%typemap(python,out) int,short { \$target = PyBuild_Value("i", (\$type) \$source); }</pre>
long	<pre>%typemap(python,out) long { \$target = PyBuild_Value("l", \$source); }</pre>
float, double	<pre>%typemap(python,out) float,double { \$target = PyBuild_Value("d", (\$type) \$source); }</pre>

Function return typemaps

char *	<pre>%typemap(python,out) char * { \$target = PyBuild_Value("s",\$source); }</pre>
FILE *	<pre>%typemap(python,out) FILE * { \$target = PyFile_FromFile(\$source); }</pre>

Pointer handling

SWIG pointers are mapped into Python strings containing the hexadecimal value and type. The following functions can be used to create and read pointer values.

SWIG Pointer Conversion Functions

<pre>void SWIG_MakePtr(char *str, void *ptr, char *type)</pre>	Makes a pointer string and saves it in <code>str</code> , which must be large enough to hold the result. <code>ptr</code> contains the pointer value and <code>type</code> is the string representation of the type.
<pre>char *SWIG_GetPtr(char *str, void **ptr, char *type)</pre>	Attempts to read a pointer from the string <code>str</code> . <code>ptr</code> is the address of the pointer to be created and <code>type</code> is the expected type. If <code>type</code> is NULL, then any pointer value will be accepted. On success, this function returns NULL. On failure, it returns the pointer to the invalid portion of the pointer string.

These functions can be used in typemaps. For example, the following typemap makes an argument of "char *buffer" accept a pointer instead of a NULL-terminated ASCII string.

```
%typemap(python,in) char *buffer {
    PyObject *o;
    char *str;
    if (!PyString_Check(o)) {
        PyErr_SetString(PyExc_TypeError,"not a string");
        return NULL;
    }
    str = PyString_AsString(o);
    if (SWIG_GetPtr(str, (void **) &$target, "$mangle")) {
        PyErr_SetString(PyExc_TypeError,"not a pointer");
        return NULL;
    }
}
```

Note that the `$mangle` variable generates the type string associated with the datatype used in the typemap.

By now you hopefully have the idea that typemaps are a powerful mechanism for building more specialized applications. While writing typemaps can be technical, many have already been written for you. See the Typemaps chapter for more information about using library files.

Implementing C callback functions in Python

Now that you're an expert, we will implement simple C callback functions in Python and use them in a C++ code.

Let's say that we wanted to write a simple C++ 2D plotting widget layered on top of the gd-1.2 library. A class definition might look like this :

```
// -----
// Create a C++ plotting "widget" using the gd-1.2 library by Thomas Boutell
//
// This example primarily illustrates how callback functions can be
// implemented in Python.
// -----

#include <stdio.h>
extern "C" {
#include "gd.h"
}

typedef double (*PLOTFUNC)(double, void *);

class PlotWidget {
private:
    double      xmin,ymin,xmax,ymax;          // Plotting range
    PLOTFUNC    callback;                    // Callback function
    void      *clientdata;                   // Client data for callback
    int       npoints;                       // Number of points to plot
    int       width;                         // Image width
    int       height;                        // Image height
    int       black,white;                   // Some colors
    gdImagePtr im;                           // Image pointer
    void      transform(double,double,int&,int&);
public:
    PlotWidget(int w, int h,double,double,double,double);
    ~PlotWidget();
    void set_method(PLOTFUNC func, void *clientdata); // Set callback method
    void set_range(double,double,double,double);     // Set plot range
    void set_points(int np) {npoints = np;}         // Set number of points
    void plot();                                     // Make a plot
    void save(FILE *f);                              // Save a plot to disk
};
```

The widget class hides all of the underlying implementation details so this could have just as easily been implemented on top of OpenGL, X11 or some other kind of library. When used in C++, the widget works like this :

```
// Simple main program to test out our widget
#include <stdio.h>
#include "widget.h"
#include <math.h>
```

```

// Callback function
double my_func(double a, void *clientdata) {
    return sin(a);
}

int main(int argc, char **argv) {
    PlotWidget *w;
    FILE *f;

    w = new PlotWidget(500,500,-6.3,-1.5,6.3,1.5);
    w->set_method(my_func,0);           // Set callback function
    w->plot();                          // Make plot
    f = fopen("plot.gif","w");
    w->save(f);
    fclose(f);
    printf("wrote plot.gif\n");
}

```

Now suppose that we wanted to use our widget interactively from Python. While possible, it is going to be difficult because we would really like to implement the callback function in Python, not C++. We also don't want to go in and hack our C++ code to support this. Fortunately, you can do it with SWIG using the following interface file :

```

// SWIG interface to our PlotWidget
%module plotwidget
%{
#include "widget.h"
%}

// Grab a Python function object as a Python object.
%typemap(python,in) PyObject *pyfunc {
    if (!PyCallable_Check($source)) {
        PyErr_SetString(PyExc_TypeError, "Need a callable object!");
        return NULL;
    }
    $target = $source;
}

// Type mapping for grabbing a FILE * from Python
%typemap(python,in) FILE * {
    if (!PyFile_Check($source)) {
        PyErr_SetString(PyExc_TypeError, "Need a file!");
        return NULL;
    }
    $target = PyFile_AsFile($source);
}

// Grab the class definition
#include widget.h

%{
/* This function matches the prototype of the normal C callback
function for our widget. However, we use the clientdata pointer
for holding a reference to a Python callable object. */

static double PythonCallBack(double a, void *clientdata)
{

```

```

PyObject *func, *arglist;
PyObject *result;
double    dres = 0;

func = (PyObject *) clientdata;           // Get Python function
arglist = Py_BuildValue("(d)",a);         // Build argument list
result = PyEval_CallObject(func,arglist); // Call Python
Py_DECREF(arglist);                       // Trash arglist
if (result) {                              // If no errors, return double
    dres = PyFloat_AsDouble(result);
}
Py_XDECREF(result);
return dres;
}
%}

// Attach a new method to our plot widget for adding Python functions
%addmethods PlotWidget {
    // Set a Python function object as a callback function
    // Note : PyObject *pyfunc is remapped with a typemap
    void set_pymethod(PyObject *pyfunc) {
        self->set_method(PythonCallBack, (void *) pyfunc);
        Py_INCREF(pyfunc);
    }
}

```

While this is certainly not a trivial SWIG interface file, the results are quite cool. Let's try out our new Python module :

```

# Now use our plotting widget in variety of ways

from plotwidget import *
from math import *

# Make a plot using a normal Python function as a callback
def func1(x):
    return 0.5*sin(x)+0.25*sin(2*x)+0.125*cos(4*x)

print "Making plot1.gif..."
# Make a widget and set callback
w = PlotWidget(500,500,-10,-2,10,2)
w.set_pymethod(func1)           # Register our Python function
w.plot()
f = open("plot1.gif","w")
w.save(f)
f.close()

# Make a plot using an anonymous function

print "Making plot2.gif..."
w1 = PlotWidget(500,500,-4,-1,4,16)
w1.set_pymethod(lambda x: x*x)   # Register x^2 as a callback
w1.plot()
f = open("plot2.gif","w")
w1.save(f)
f.close()

# Make another plot using a built-in function

```

```

print "Making plot3.gif..."
w2 = PlotWidget(500,500,-7,-1.5,7,1.5)
w2.set_pymethod(sin)                # Register sin(x) as a callback
w2.plot()
f = open("plot3.gif","w")
w2.save(f)
f.close()

```

The “plot” method for each widget is written entirely in C++ and assumes that it is calling a callback function written in C/C++. Little does it know that we have actually implemented this function in Python. With a little more work, we can even write a simple function plotting tool :

```

# Plot a function and spawn xv

import posix
import sys
import string
from plotwidget import *
from math import *

line = raw_input("Enter a function of x : ")
ranges = string.split(raw_input("Enter xmin,ymin,xmax,ymax :"),",")

print "Making a plot..."
w = PlotWidget(500,500,string.atof(ranges[0]),string.atof(ranges[1]),
               string.atof(ranges[2]),string.atof(ranges[3]))

# Turn user input into a Python function
code = "def func(x): return " + line
exec(code)

w.set_pymethod(func)
w.plot()
f = open("plot.gif","w")
w.save(f)
f.close()
posix.system("xv plot.gif &")

```

Other odds and ends

Adding native Python functions to a SWIG module

Sometimes it is desirable to add a native Python method to a SWIG wrapper file. Suppose you have the following Python/C function :

```

PyObject *spam_system(PyObject *self, PyObject *args) {
    char *command;
    int sts;
    if (!PyArg_ParseTuple(args,"s",&command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i",sts);
}

```

```
}

```

This function can be added to a SWIG module using the following declaration :

```
%native(system) spam_system;           // Create a command called 'system'
```

Alternatively, you can use the full function declaration like this

```
%native(system) PyObject *spam_system(PyObject *self, PyObject *args);
```

or

```
%native(system) extern PyObject *spam_system(PyObject *self, PyObject *args);
```

The gory details of shadow classes

This section describes the process by which SWIG creates shadow classes and some of the more subtle aspects of using them.

A simple shadow class

Consider the following declaration from our previous example :

```
%module pdec
struct Grid2d {
    Grid2d(int ni, int nj);
    ~Grid2d();
    double **data;
    int     xpoints;
    int     ypoints;
};
```

The SWIG generated class for this structure looks like the following :

```
# This file was created automatically by SWIG.
import pdec
class Grid2dPtr :
    def __init__(self,this):
        self.this = this
        self.thisown = 0
    def __del__(self):
        if self.thisown == 1 :
            pdec.delete_Grid2d(self.this)
    def __setattr__(self,name,value):
        if name == "data" :
            pdec.Grid2d_data_set(self.this,value)
            return
        if name == "xpoints" :
            pdec.Grid2d_xpoints_set(self.this,value)
            return
        if name == "ypoints" :
            pdec.Grid2d_ypoints_set(self.this,value)
            return
        self.__dict__[name] = value
    def __getattr__(self,name):
```

```

    if name == "data" :
        return pdec.Grid2d_data_get(self.this)
    if name == "xpoints" :
        return pdec.Grid2d_xpoints_get(self.this)
    if name == "ypoints" :
        return pdec.Grid2d_ypoints_get(self.this)
    return self.__dict__[name]
def __repr__(self):
    return "<C Grid2d instance>"
class Grid2d(Grid2dPtr):
    def __init__(self, arg0, arg1) :
        self.this = pdec.new_Grid2d(arg0, arg1)
        self.thisown = 1

```

Module names

Shadow classes are built using the low-level SWIG generated C interface. This interface is named “modulec” where “module” is the name of the module specified in a SWIG interface file. The Python code for the shadow classes is created in a file “module.py”. This is the file that should be loaded when a user wants to use the module.

Two classes

For each structure or class found in an interface file, SWIG creates two Python classes. If a class is named “Grid2d”, one of these classes will be named “Grid2dPtr” and the other named “Grid2d”. The Grid2dPtr class is used to turn wrap a Python class around an already pre-existing Grid2d pointer. For example :

```

>>> gptr = create_grid2d()           # Returns a Grid2d from somewhere
>>> g = Grid2dPtr(gptr)              # Turn it into a Python class
>>> g.xpoints
50
>>>

```

The Grid2d class, on the other hand, is used when you want to create a new Grid2d object from Python. In reality, it inherits all of the attributes of a Grid2dPtr, except that its constructor calls the corresponding C++ constructor to create a new object. Thus, in Python, this would look something like the following :

```

>>> g = Grid2d(50,50)                # Create a new Grid2d
>>> g.xpoints
50
>>>

```

This two class model is a tradeoff. In order to support C/C++ properly, it is necessary to be able to create Python objects from both pre-existing C++ objects and to create entirely new C++ objects in Python. While this might be accomplished using a single class, it would complicate the handling of constructors considerably. The two class model, on the other hand, works, is consistent, and is relatively easy to use. In practice, you probably won’t even be aware that there are two classes working behind the scenes.

The this pointer

Within each shadow class, the member “this” contains the actual C/C++ pointer to the object.

You can check this out yourself by typing something like this :

```
>>> g = Grid2d(50,50)
>>> print g.this
_1008fe8_Grid2d_p
>>>
```

Direct manipulation of the “this” pointer is generally discouraged. In fact forget that you read this.

Object ownership

Ownership is a critical issue when mixing C++ and Python. For example, suppose I create a new object in C++, but later use it to create a Python object. If that object is being used elsewhere in the C++ code, we clearly don't want Python to delete the C++ object when the Python object is deleted. Similarly, what if I create a new object in Python, but C++ saves a pointer to it and starts using it repeatedly. Clearly, we need some notion of who owns what. Since sorting out all of the possibilities is probably impossible, SWIG shadow classes always have an attribute “thisown” that indicates whether or not Python owns an object. Whenever an object is created in Python, Python will be given ownership by setting `thisown` to 1. When a Python class is created from a pre-existing C/C++ pointer, ownership is assumed to belong to the C/C++ code and `thisown` will be set to 0.

Ownership of an object can be changed as necessary by changing the value of `thisown`. When set, Python will call the C/C++ destructor when the object is deleted. If it is zero, Python will never call the C/C++ destructor.

Constructors and Destructors

C++ constructors and destructors will be mapped into Python's `__init__` and `__del__` methods respectively. Shadow classes always contain these methods even if no constructors or destructors were available in the SWIG interface file. The Python destructor will only call a C/C++ destructor if `self.thisown` is set.

Member data

Member data of an object is accessed through Python's `__getattr__` and `__setattr__` methods.

Printing

SWIG automatically creates a Python `__repr__` method for each class. This forces the class to be relatively well-behaved when printing or being used interactively in the Python interpreter.

Shadow Functions

Suppose you have the following declarations in an interface file :

```
%module vector
struct Vector {
    Vector();
    ~Vector();
    double x,y,z;
};
```

```
Vector addv(Vector a, Vector b);
```

By default, the function `addv` will operate on `Vector` pointers, not Python classes. However, the SWIG Python module is smart enough to know that `Vector` has been wrapped into a Python class so it will create the following replacement for the `addv()` function.

```
def addv(a,b):
    result = VectorPtr(vectorc.addv(a.this,b.this))
    result.thisown = 1
    return result
```

Function arguments are modified to use the “this” pointer of a Python `Vector` object. The result is a pointer to the result which has been allocated by `malloc` or `new` (this behavior is described in the chapter on SWIG basics), so we simply create a new `VectorPtr` with the return value. Since the result involved an implicit `malloc`, we set the ownership to 1 indicating that the result is to be owned by Python and that it should be deleted when the Python object is deleted. As a result, operations like this are perfectly legal and result in no memory leaks :

```
>>> v = add(add(add(add(a,b),c),d),e)
```

Substitution of complex datatypes occurs for all functions and member functions involving structure or class definitions. It is rarely necessary to use the low-level C interface when working with shadow classes.

Nested objects

SWIG shadow classes support nesting of complex objects. For example, suppose you had the following interface file :

```
%module particle

typedef struct {
    Vector();
    double x,y,z;
} Vector;

typedef struct {
    Particle();
    ~Particle();
    Vector r;
    Vector v;
    Vector f;
    int type;
} Particle;
```

In this case you will be able to access members as follows :

```
>>> p = Particle()
>>> p.r.x = 0.0
>>> p.r.y = -1.5
>>> p.r.z = 2.0
>>> p.v = addv(v1,v2)
>>> ...
```

Nesting of objects is implemented using Python's `__setattr__` and `__getattr__` functions. In this case, they would look like this :

```
class ParticlePtr:
    ...
    def __getattr__(self,name):
        if name == "r":
            return particlec.VectorPtr(Particle_r_get(self.this))
        elif name == "v":
            return particlec.VectorPtr(Particle_v_get(self.this))
        ...

    def __setattr__(self,name,value):
        if name == "r":
            particlec.Particle_r_set(self.this,value.this)
        elif name == "v":
            particlec.Particle_v_set(self.this,value.this)
        ...
```

The attributes of any given object are only converted into a Python object when referenced. This approach is more memory efficient, faster if you have a large collection of objects that aren't examined very often, and works with recursive structure definitions such as :

```
struct Node {
    char *name;
    struct Node *next;
};
```

Nested structures such as the following are also supported by SWIG. These types of structures tend to arise frequently in database and information processing applications.

```
typedef struct {
    unsigned int dataType;
    union {
        int      intval;
        double   doubleval;
        char     *charval;
        void     *ptrvalue;
        long     longval;
        struct {
            int    i;
            double f;
            void   *v;
            char  name[32];
        } v;
    } u;
} ValueStruct;
```

Access is provided in an entirely natural manner,

```
>>> v = new_ValueStruct()      # Create a ValueStruct somehow
>>> v.dataType
1
>>> v.u.intval
45
```

```
>>> v.u.longval
45
>>> v.u.v.v = _0_void_p
>>>
```

To support the embedded structure definitions, SWIG has to extract the internal structure definitions and use them to create new Python classes. In this example, the following shadow classes are created :

```
# Class corresponding to union u member
class ValueStruct_u :
    ...
# Class corresponding to struct v member of union u
class ValueStruct_u_v :
    ...
```

The names of the new classes are formed by appending the member names of each embedded structure.

Inheritance and shadow classes

Since shadow classes are implemented in Python, you can use any of the automatically generated classes as a base class for more Python classes. However, you need to be extremely careful when using multiple inheritance. When multiple inheritance is used, at most ONE SWIG generated shadow class can be involved. If multiple SWIG generated classes are used in a multiple inheritance hierarchy, you will get name clashes on the `this` pointer, the `__getattr__` and `__setattr__` functions won't work properly and the whole thing will probably crash and burn. Perhaps it's best to think of multiple inheritance as a big hammer that can be used to solve a lot of problems, but it hurts quite a lot if you accidentally drop it on your foot....

Methods that return new objects

By default SWIG assumes that constructors are the only functions returning new objects to Python. However, you may have other functions that return new objects as well. For example :

```
Vector *cross_product(Vector *v1, Vector *v2) {
    Vector *result = new Vector();
    result = ... compute cross product ...
    return result;
}
```

When the value is returned to Python, we want Python to assume ownership. The brute force way to do this is to simply change the value of `thisown`. For example :

```
>>> v = cross_product(a,b)
>>> v.thisown = 1          # Now Python owns it
```

Unfortunately, this is ugly and it doesn't work if we use the result as a temporary value :

```
w = vector_add(cross_product(a,b),c)          # Results in a memory leak
```

However, you can provide a hint to SWIG when working with such a function as shown :

```
// C Function returning a new object
%new Vector *cross_product(Vector *v1, Vector *v2);
```

The `%new` directive only provides a hint that the function is returning a new object. The Python module will assign proper ownership of the object when this is used.

Performance concerns and hints

Shadow classing is primarily intended to be a convenient way of accessing C/C++ objects from Python. However, if you're directly manipulating huge arrays of complex objects from Python, performance may suffer greatly. In these cases, you should consider implementing the functions in C or thinking of ways to optimize the problem.

There are a number of ways to optimize programs that use shadow classes. Consider the following two code fragments involving the `Particle` data structure in a previous example :

```
def force1(p1,p2):
    dx = p2.r.x - p1.r.x
    dy = p2.r.y - p1.r.y
    dz = p2.r.z - p1.r.z
    r2 = dx*dx + dy*dy + dz*dz
    f = 1.0/(r2*math.sqrt(r2))
    p1.f.x = p1.f.x + f*dx
    p2.f.x = p2.f.x - f*dx
    p1.f.y = p1.f.y + f*dy
    p2.f.y = p2.f.y - f*dy
    p1.f.z = p1.f.z + f*dz
    p2.f.z = p2.f.z - f*dz

def force2(p1,p2):
    r1 = p1.r
    r2 = p2.r
    dx = r2.x - r1.x
    dy = r2.y - r1.y
    dz = r2.z - r1.z
    r2 = dx*dx + dy*dy + dz*dz
    f = 1.0/(r2*math.sqrt(r2))
    f1 = p1.f
    f2 = p2.f
    f1.x = f1.x + f*dx
    f2.x = f2.x - f*dx
    f1.y = f1.y + f*dy
    f2.y = f2.y - f*dy
    f1.z = f1.z + f*dz
    f2.z = f2.z - f*dz
```

The first calculation simply works with each `Particle` structure directly. Unfortunately, it performs a lot of dereferencing of objects. If the calculation is restructured to use temporary variables as shown in `force2`, it will run significantly faster--in fact, on my machine, the second code fragment runs more than twice as fast as the first one.

If performance is even more critical you can use the low-level C interface which eliminates all of the overhead of going through Python's class mechanism (at the expense of coding simplicity). When Python shadow classes are used, the low level C interface can still be used by importing the `'modulec'` module where `'module'` is the name of the module you used in the SWIG interface

file.