Chapter 5

Combining Python with Fortran, C, and C++

Most languages offer the possibility to call code written in other languages, but in Python this is a particularly simple and smooth process. One reason is that Python was initially designed for being integrated with C and extended with new C code. The support for C implicitly provides support for closely related languages like Fortran and C++. Another reason is that tools, such as F2PY and SWIG, have been developed in recent years to assist the integration and, in simpler cases, fully automate it. The present chapter is a first introduction to mixed language programming with Python, Fortran 77 (F77), C, and C++. The focus is on applying the tools F2PY and SWIG to automate the integration process.

Chapter 5.1.2 gives an introduction to the nature of mixed language programming. Chapter 5.2 applies a simple Scientific Hello World example to demonstrate how to call F77, C, and C++ from Python. The F77 simulator from Chapter 2.3 can be equipped with a Python interface. A case study on how to perform this integration of Python and F77 is presented in Chapter 5.3.

In scientific computing we often invoke compiled languages to perform numerical operations on large array structures. This topic is treated in detail in Chapters 9 and 10.

Readers interested in Python-Fortran integration only may skip reading the C and C++ material in Chapters 5.2.2 and 5.2.3. Conversely, those who want to avoid the Fortran material may skip Chapters 5.2.1 and 5.3.

5.1 About Mixed Language Programming

First, in Chapter 5.1.1, we briefly describe the contexts where mixed language programming is useful and some implications to numerical code design.

Integration of Python with Fortran 77 (F77), C, and C++ code requires a communication layer, called *wrapper code*. Chapter 5.1.2 outlines the need for wrapper code and how it looks like. Thereafter, in Chapter 5.1.3, some tools are mentioned for generating wrapper code or assisting the writing of such code.

5.1.1 Applications of Mixed Language Programming

Integration of Python with Fortran, C, or C++ code is of interest in two main contexts:

1. *Migration of slow code.* We write a new application in Python, but migrate numerical intensive calculations to Fortran or C/C++.
2. *Access to existing numerical code.* We want to call existing numerical libraries or applications in Fortran or C/C++ directly from Python.

In both cases we want to benefit from using Python for non-numerical tasks. This involves user interfaces, I/O, report generation, and management of the entire application. Having such components in Python makes it fast and convenient to modify code, test, glue with other packages, steer computations interactively, and perform similar tasks needed when exploring scientific or engineering problems. The syntax and usage can be made close to that of Matlab, indicating that such interfaces may greatly simplify the usage of the underlying compiled language code. A user may be productive in this type of environment with only some basic knowledge of Python.

The two types of mixed language programming pose different challenges. When interfacing a monolithic application in a compiled language, one often wants to interface only the computationally intensive functions. That is, one discards I/O, user interfaces, etc. and moves these parts to Python. The design of the monolithic application determines how easy it is to split the code into the desired components.

Writing a new scientific computing application in Python and moving CPU-time critical parts to a compiled language has certain significant advantages. First of all, the design of the application will often be better than what is accomplished in a compiled language. The reason is that the many powerful language features of Python make it easier to create abstractions that are close to the problem formulation and well suited for future extensions. The resulting code is usually compact and easy to read. The class and module concepts help organizing even very large applications. What we achieve is a *high-level design* of numerical applications. By careful profiling (see Chapter 8.10.2) one can identify bottlenecks and move these to Fortran, C, or C++. Existing Fortran, C, or C++ code may be reused for this purpose, but the interfaces might need adjustments to integrate well with high-level Python abstractions.

5.1.2 Calling C from Python

Interpreted languages differ a lot from compiled languages like C, C++, and Fortran as we have outlined in Chapter 1.1. Calling code written in a compiled language from Python is therefore not a trivial task. Fortran, C, and C++

Java have strong typing rules, which means that a variable is declared and allocated in memory with proper size before it is used. In Python, variables are typeless, at least in the sense that a variable can be an integer and then change to a string or a window button:

```
d = 3.2    # d holds a float
d = 'txt'  # d holds a string
d = Button(frame, text='push')  # d holds a Button instance
```

In a compiled language, `d` can only hold one type of variable, while in Python `d` just references an object of any defined type (like `void*` in C/C++). This is one of the reasons why we need a technically quite comprehensive interface between a language with static typing and a dynamically typed language.

Python is implemented in C and designed to be extended with C functions. Naturally, there are rules and C utilities available for sending variables from Python to C and back again. Let us look at a simple example to illustrate how wrapper code may look like.

Suppose we in a Python script want to call a C function that takes two `double`s as arguments and returns a `double`:

```
extern double hw1(double r1, double r2);
```

This C function will be available in a module (say) `hw`. In the Python script we can then write

```
from hw import hw1
r1 = 1.2; r2 = -1.2
s = hw1(r1, r2)
```

The Python code must call a wrapper function, written in C, where the contents of the arguments are analyzed, the double precision floating-point numbers are extracted and stored in straight C `double` variables. Then, the wrapper function can call our C function `hw1`. Since the `hw1` function returns a `double`, we need to convert this `double` to a Python object that can be returned to the calling Python code and referred by the object `s`. A wrapper function can in this case look as follows:

```
static PyObject *_wrap_hw1(PyObject *self, PyObject *args) {
    double arg1, arg2, result;

    if (!PyArg_ParseTuple(args, "dd:hw1", &arg1, &arg2)) {
        return NULL;  /* wrong arguments provided */
    }
    result = hw1(arg1, arg2);
    return Py_BuildValue("d", result);
}
```

All objects in Python are derived from the `PyObject` "class" (Python is coded in pure C, but the implementation simulates object-oriented programming). A wrapper function typically takes two arguments, `self` and `args`. The first is

of relevance only when dealing with instance methods, and `args` holds a tuple of the arguments sent from Python, here `r1` and `r2`, which we expect to be two doubles. (A third argument to the wrapper function may hold keyword arguments.) We may use the utility `PyArg_ParseTuple` in the Python C library for converting the `args` object to two `double` variables (specified as the string `dd`). The doubles are stored in the help variables `arg1` and `arg2`. Having these variables, we can call the `hw1` function. The `Py_BuildValue` function from the Python C library packs a C variable (here of type `double`) as a Python object, which is returned to the calling code and there appears as a standard Python `float` object.

The wrapper function must be compiled, here with a C compiler. We must also compile the file with the `hw1` function. The object code of the `hw1` function must then be linked with the wrapper code to form a shared library module. Such a shared library module is also often referred to as an *extension module* and can be loaded into Python using the standard `import` statement. From Python, it is impossible[1] to distinguish between a pure Python module or an extension module based on pure C code.

5.1.3 Automatic Generation of Wrapper Code

As we have tried to demonstrate, the writing of wrapper functions requires knowledge of how Python objects are manipulated in C code. In other words, one needs to know details of the C interface to Python, referred to as the Python C API (API stands for Application Programming Interface). The official electronic Python documentation (see link from `doc.html`) has a tutorial for the C API, called "Extending and Embedding the Python Interpreter" [36], and a reference manual for the API, called "Python/C API". The C API is also covered in numerous books [2,12,20,22].

The major problem with writing wrapper code is that it is a big job: each C function you want to call from Python must have an associated wrapper function. Such manual work is boring and error-prone. Luckily, tools have been developed to automate this manual work.

SWIG (Simplified Wrapper Interface Generator), originally developed by David Beazley, automates the generation of wrapper code for interfacing C and C++ software from dynamically typed languages. Lots of such languages are supported, including Guile, Java, Mzscheme, Ocaml, Perl, Pike, PHP, Python, Ruby, and Tcl. Sometimes SWIG may be a bit difficult to use beyond the getting-started examples in the SWIG manual. This is due to the flexibility of C and especially C++, and the different nature of dynamically typed languages and C/C++.

[1] This is not completely correct: the module's `__file__` attribute is the name of a `.py` file for a pure Python module and the name of a compiled shared library file for a C extension module. Also, C extension modules cannot be reimported with the `reload` function.

Making an interface between Fortran code and Python is very easy using the high-level tool F2PY, developed by Pearu Peterson. Very often F2PY is able to generate C wrapper code for Fortran libraries in a fully automatic way. Transferring NumPy arrays between Python and compiled code is much simpler with F2PY than with SWIG. Fortunately, F2PY can also be used with C code, though this requires some familiarity with Fortran. For C++ code it can be an idea to write a small C interface and use F2PY on this interface in order to pass arrays between Python and C++.

A tool called Instant can be used to put C or C++ code inline in Python code and get automatically compiled as an extension library, much in the same way as F2PY does. Instant has good support for NumPy arrays and is very easy to use. SWIG is invisibly applied to generate the wrapper code.

In this book we mainly concentrate on making Python interfaces to C, C++, and Fortran functions that do not use any of the features in the Python C API. However, sometimes one desires to manipulate Python data structures, like lists, dictionaries, and NumPy arrays, in C or C++ code. This requires the C or C++ code to make direct use of the Python and NumPy C API. One will then often wind the wrapper functionality and the data manipulation into one function. Examples on such programming appear in Chapters 10.2 and 10.3.

It should be mentioned that there is a Python interpreter, called Jython, implemented in 100% pure Java, which allows a seamless integration of Python and Java code. There is no need to write wrappers: any Java class can be used in a Jython script and vice versa.

Alternatives to F2PY, Instant, and SWIG. We will in this book mostly use F2PY, Instant, and SWIG to interface Fortran, C, and C++ from Python, but several other tools for assisting the generation of wrapper functions can be used. CXX, Boost.Python, and SCXX are C++ tools that simplify programming with the Python C API. With these tools, the C++ code becomes much closer to pure Python than C code operating on the C API directly. Another important application of the tools is to generate Python interfaces to C++ packages. However, the tools do not generate the interfaces automatically, and manual coding is necessary. The use of SCXX is exemplified in Chapter 10.3. SIP is a tool for wrapping C++ (and C) code, much like SWIG, but it is specialized for Python-C++ integration and has a potential for producing more efficient code than SWIG. The documentation of SIP is unfortunately still sparse at the time of this writing. Weave allows inline C++ code in Python scripts and is hence a tool much like Instant.

Psyco is a very simple-to-use tool for speeding up Python code. It works like a kind of just-in-time compiler, which analyzes the Python code at run time and moves time-critical parts to C. Pyrex is a small language for simplified writing of extension modules. The purpose is to reduce the normally quite comprehensive work of developing a C extension module from scratch. Links to the mentioned tools can be found in the `doc.html` file.

Systems like COM/DCOM, CORBA, XML-RPC, and ILU are sometimes useful alternatives to the code wrapping scheme described above. The Python script and the C, C++, or Fortran code communicate in this case through a layer of objects, where the data are copied back and forth between the script and the compiled language code. The codes on each side of the layer can be run as separate processes, and the communication can be over a network. The great advantage is that it becomes easy to run the light-weight script on a small computer and leave heavy computations to a more powerful machine. One can also create interfaces to C, C++, and Fortran codes that can be easily called from a wide range of languages.

The approach based on wrapper code allows transfer of huge data structures by just passing pointers around, which is very efficient when the script and the compiled language code are run on the same machine. Learning the basics of F2PY takes an hour or two, SWIG require somewhat more time, but still very much less than the the complicated and comprehensive "interface definition languages" COM/DCOM, CORBA, XML-RPC, and ILU. One can summarize these competing philosophies by saying that tools like F2PY and SWIG offer simplicity and efficiency, whereas COM/DCOM, CORBA, XML-RPC, and ILU give more flexibility and more complexity.

5.2 Scientific Hello World Examples

As usual in this book, we introduce new concepts using the simple Scientific Hello World example (see Chapters 2.1 and 6.1). In the context of mixed language programming, we make an extended version of this example where some functions in a module are involved. The first function, `hw1`, returns the sine of the sum of two numbers. The second function, `hw2`, computes the same sine value, but writes the value together with the "Hello, World!" message to the screen. A pure Python implementation of our module, called `hw`, reads

```
#!/usr/bin/env python
"""Pure Python Scientific Hello World module."""
import math

def hw1(r1, r2):
    s = math.sin(r1 + r2)
    return s

def hw2(r1, r2):
    s = math.sin(r1 + r2)
    print 'Hello, World! sin(%g+%g)=%g' % (r1, r2, s)
```

The `hw1` function returns a value, whereas `hw2` does not. Furthermore, `hw1` contains pure numerical computations, whereas `hw2` also performs I/O.

An application script utilizing the `hw` module may take the form

```
#!/usr/bin/env python
"""Scientific Hello World script using the module hw."""
```

```
import sys
from hw import hw1, hw2
try:
    r1 = float(sys.argv[1]);  r2 = float(sys.argv[2])
except IndexError:
    print 'Usage:', sys.argv[0], 'r1 r2'; sys.exit(1)
print 'hw1, result:', hw1(r1, r2)
print 'hw2, result: ',
hw2(r1, r2)
```

The goal of the next subsections is to migrate the `hw1` and `hw2` functions in the `hw` module to F77, C, and C++. The application script will remain the same, as the language used for implementing the module `hw` is transparent in the Python code. We will also involve a third function, `hw3`, which is a version of `hw1` where `s` is an *output argument*, in call by reference style, and not a return variable. A pure Python implementation of `hw3` has no meaning (cf. Chapter 3.3 and the *Call by Reference* paragraph).

The Python implementations of the module and the application script are available as the files `hw.py` and `hwa.py`, respectively. These files are found in the directory `src/py/mixed/hw`.

5.2.1 Combining Python and Fortran

A Fortran 77 implementation of `hw1` and `hw2`, as well as a main program for testing the functions, appear in the file `src/py/mixed/hw/F77/hw.f`. The two functions are written as

```
      real*8 function hw1(r1, r2)
      real*8 r1, r2
      hw1 = sin(r1 + r2)
      return
      end

      subroutine hw2(r1, r2)
      real*8 r1, r2, s
      s = sin(r1 + r2)
      write(*,1000) 'Hello, World! sin(',r1+r2,')=',s
 1000 format(A,F6.3,A,F8.6)
      return
      end
```

We shall use the F2PY tool for creating a Python interface to the F77 versions of `hw1` and `hw2`. F2PY comes with the NumPy package so when you install NumPy, automatically install F2PY and get an executable `f2py` that we sall make use of. Since creation of the F2PY interface implies generation of some files, we make a subdirectory, `f2py-hw`, and run F2PY in this subdirectory. The F2PY command is very simple:

```
f2py -m hw -c ../hw.f
```

The `-m` option specifies the name of the extension module, whereas the `-c` option indicates that F2PY should compile and link the module. The result of the F2PY command is an extension module in the file `hw.so`[2], which may be loaded into Python by an ordinary `import` statement. It is a good habit to test that the module is successfully built and can be imported:

```
python -c 'import hw'
```

The `-c` option to `python` allows us to write a short script as a text argument.

The application script `hwa.py` presented on page 194 can be used to test the functions in the module. That is, this script cannot see whether we have written the `hw` module in Fortran or Python.

The F2PY command may result in some annoying error messages when F2PY searches for a suitable Fortran compiler. To avoid these messages, we can specify the compiler to be used, for instance GNU's `g77` or `gfortran` compilers:

```
f2py -m hw -c --fcompiler=Gnu ../hw.f
```

You can run `f2py -c --help-fcompiler` to see a list of the supported Fortran compilers on your system (`--help-fcompiler` shows a list of C compilers). F2PY has lots of other options to fine-tune the interface. This is well explained in the F2PY manual.

When dealing with more complicated Fortran libraries, one may want to create Python interfaces to only some of the functions. In the present case we could explicitly demand interfaces to the `hw1` and `hw2` functions by including the specification `only: <functions> :` after the name of the Fortran file(s), e.g.,

```
f2py -m hw -c --fcompiler=Gnu ../hw.f only: hw1 hw2 :
```

The interface to the extension module is specified as Fortran 90 module interfaces, and the `-h hw.pyf` option makes F2PY write the Fortran 90 module interfaces to a file `hw.pyf` such that you can adjust them according to your needs.

Handling of Output Arguments. To see how we actually need to adjust the interface file `hw.pyf`, we have written a third function in the `hw.f` file:

```
subroutine hw3(r1, r2, s)
real*8 r1, r2, s
s = sin(r1 + r2)
return
end
```

This is an alternative version of `hw1` where the result of the computations is stored in the output argument `s`. Since Fortran 77 employs the call by reference technique for all arguments, any change to an argument is visible in the calling code. If we let F2PY generate interfaces to all the functions in `hw.f`,

[2] On Windows the extension is `.dll` and on Mac OS X the extension is `.dylib`.

```
f2py -m hw -h hw.pyf ../hw.f
```

the interface file `hw.pyf` becomes

```
python module hw ! in
    interface  ! in :hw
        function hw1(r1,r2) ! in :hw:../hw.f
            real*8 :: r1
            real*8 :: r2
            real*8 :: hw1
        end function hw1
        subroutine hw2(r1,r2) ! in :hw:../hw.f
            real*8 :: r1
            real*8 :: r2
        end subroutine hw2
        subroutine hw3(r1,r2,s) ! in :hw:../hw.f
            real*8 :: r1
            real*8 :: r2
            real*8 :: s
        end subroutine hw3
    end interface
end python module hw
```

By default, F2PY treats `r1`, `r2`, and `s` in the `hw3` function as input arguments. Trying to call `hw3`,

```
>>> from hw import hw3
>>> r1 = 1;  r2 = -1;  s = 10
>>> hw3(r1, r2, s)
>>> print s
10   # should be 0.0
```

shows that the value of the Fortran `s` variable is not returned to the Python `s` variable in the call. The remedy is to tell F2PY that `s` is an output parameter. To this end, we must in the `hw.pyf` file replace

```
            real*8 :: s
```

by the Fortran 90 specification of an output variable:

```
            real*8, intent(out) :: s
```

Without any `intent` specification the variable is assumed to be an input variable. The directives `intent(in)` and `intent(out)` specify input and output variables, respectively, while `intent(in,out)` and `intent(inout)`[3] are employed for variables used for input *and* output.

Compiling and linking the `hw` module, utilizing the modified interface specification in `hw.pyf`, are now performed by

```
f2py -c --fcompiler=Gnu hw.pyf ../hw.f
```

[3] The latter is not recommended for use with F2PY, see Chapter 9.3.3.

F2PY always equips the extension module with a doc string[4] specifying the signature of each function:

```
>>> import hw
>>> print hw.__doc__
Functions:
  hw1 = hw1(r1,r2)
  hw2(r1,r2)
  s = hw3(r1,r2)
```

Novice F2PY users will get a surprise that F2PY has changed the `hw3` interface to become more Pythonic, i.e., from Python we write

```
s = hw3(r1, r2)
```

In other words, `s` is now *returned* from the `hw3` function, as seen from Python. This is the Pythonic way of programming – results are returned form functions. For a Fortran routine

```
subroutine somef(i1, i2, o1, o2, o3, o4, io1)
```

where `i1` and `i2` are input variables, `o1`, `o2`, `o3`, and `o4` are output variables, and `io1` is an input/output variable, the generated Python interface will have `i1`, `i2`, and `io1` as arguments to `somef` and `o1`, `o2`, `o3`, `o4`, and `io1` as a returned tuple:

```
o1, o2, o3, o4, io1 = somef(i1, i2, io1)
```

Fortunately, F2PY automatically generates doc strings explaining how the signature of the function is changed.

Sometimes it may be convenient to perform the modification of the `.pyf` interface file automatically. In the present case we could use the `subst.py` script from Chapter 8.2.11 to edit `hw.pyf`:

```
subst.py 'real\*8\s*::\s*s' 'real*8, intent(out) :: s' hw.pyf
```

When the editing is done automatically, it is convenient to allow F2PY generate a new (default) interface file the next time we run F2PY, even if a possibly edited `hw.pyf` file exists. The `--overwrite-signature` option allows us to generate a new `hw.pyf` file. Our set of commands for creating the desired Python interface to `hw.f` now becomes

```
f2py -m hw -h hw.pyf ../hw.f --overwrite-signature
subst.py 'real\*8\s*::\s*s' 'real*8, intent(out) :: s' hw.pyf
f2py -c --fcompiler=Gnu hw.pyf ../hw.f
```

Various F2PY commands for creating the present extension module are collected in the `src/py/mixed/hw/f2py-hw/make_module.sh` script.

A quick one-line command for checking that the Fortran-based `hw` module passes a minium test might take the form

[4] The doc string is available as a variable `__doc__`, see Appendix B.2.

```
python -c 'import hw; print hw.hw3(1.0,-1.0)'
```

As an alternative to editing the `hw.pyf` file, we may insert an `intent` specification as a special `Cf2py` comment in the Fortran source code file:

```
      subroutine hw3(r1, r2, s)
      real*8 r1, r2, s
Cf2py intent(out) s
      s = sin(r1 + r2)
      return
      end
```

F2PY will now realize that `s` is to be specified as an output variable. If you intend to write new F77 code to be interfaced by F2PY, you should definitely insert `Cf2py` comments to specify input, output, and input/output arguments to functions as this eliminates the need to save and edit the `.pyf` file. The safest way of writing `hw3` is to specify the input/output nature of all the function arguments:

```
      subroutine hw3(r1, r2, s)
      real*8 r1, r2, s
Cf2py intent(in) r1
Cf2py intent(in) r2
Cf2py intent(out) s
      s = sin(r1 + r2)
      return
      end
```

The `intent` specification also helps to document the usage of the routine.

Case Sensitivity. Fortran is not case sensitive so we may mix lower and upper case letters with no effect in the Fortran code. However, F2PY converts all Fortran names to their lower case equivalents. A routine declared as `Hw3` in Fortran must then be called as `hw3` in Python. F2PY has an option for preserving the case when seen from Python.

Troubleshooting. If something goes wrong in the compilation, linking or module loading stage, you must first check that the F2PY commands are correct. The F2PY manual is the definite source for looking up the syntax. In some cases you need to tweak the compile and link commands. The easiest approach is to run F2PY, then cut, paste, and edit the various commands that F2PY writes to the screen. Missing libraries are occasionally a problem, but the necessary libraries can simply be added as part of the F2PY command. Another problem is that many Fortran compilers transparently add an underscore at the end of function names. F2PY has macros for adding/removing underscores in the C wrapper code. When trouble with underscores arise, you may try to switch to GNU's `gfortran` compiler as this compiler usually works smoothly with F2PY.

If you run into trouble with the interface generated by F2PY, you may want to examine in detail how F2PY builds the interface. The default behavior of F2PY is to remove the `.pyf` file and the generated wrapper code after

the extension module is built, but the `--build-dir tmp1` option makes F2PY store the generated files in a subdirectory `tmp1` such that you can inspect the files. With basic knowledge about the NumPy C API (see Chapter 10.2) you may be able to detect what the interface is actually doing. However, my main experience is that F2PY works well in automatic mode as long as you include proper `Cf2py intent` comments in the Fortran code.

Building the Extension Module Using Distutils. The standard way of building and installing Python modules, including extension modules containing compiled code in C, C++, or Fortran, is to use the Python's Distutils (Distribution Utilities) tool, which comes with the standard Python distribution. An enhanced version of Distutils with better support for Fortran code comes with Numerical Python, and its use will be illustrated here. The procedure consists of creating a script `setup.py`, which calls a function `setup` in Distutils. Building a Python module out of Fortran files is then a matter of running the `setup.py` script, e.g.,

```
python setup.py build
```

to build the extension module or

```
python setup.py install
```

to build and install the module. In the testing phase it is recommended just to build the module. The resulting shared library file, `hw.so`, is located in a directory tree `build` created by `setup.py`. To build the an extension module in the current working directory, a general command is

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

In our case where the source code for the extension module consists of the file `hw.f` in the parent directory, the `setup.py` script takes the following form:

```
from numpy.distutils.core import Extension, setup

setup(name='hw',
      ext_modules=[Extension(name='hw', sources=['../hw.f'])],
      )
```

Extension modules, consisting of compiled code, are indicated by the keyword argument `ext_modules`, which takes a list of `Extension` objects. Each `Extension` object is created with two required parameters, the name of the extension module and a list of the source files to be compiled. The `setup` function accepts additional keyword arguments like `description`, `author`, `author_email`, `license`, etc., for supplying more information with the module. There are easy-to-read introductions to Distutils in the electronic Python documentation (see link in `doc.html`): "Installing Python Modules" shows how to run a `setup.py` script, and "Distributing Python Modules" describes how to write a `setup.py` script. More information on `setup.py` scripts with Fortran code appears in the Numerical Python Manual.

5.2.2 Combining Python and C

The implementation of the `hw1`, `hw2`, and `hw3` functions in C takes the form

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

double hw1(double r1, double r2)
{
  double s;
  s = sin(r1 + r2);
  return s;
}

void hw2(double r1, double r2)
{
  double s;
  s = sin(r1 + r2);
  printf("Hello, World! sin(%g+%g)=%g\n", r1, r2, s);
}

/* special version of hw1 where the result is an argument: */
void hw3(double r1, double r2, double *s)
{
  *s = sin(r1 + r2);
}
```

The purpose of the `hw3` function is explained in Chapter 5.2.1. We use this function to demonstrate how to handle output arguments. You can find the complete code in the file `src/py/mixed/hw/C/hw.c`.

Using F2PY. F2PY is a very convenient tool also for wrapping C functions, at least for C functions taking arguments of the basic C data types that also Fortran has (`int`, `float`/`double`, `char`, and the corresponding pointers). For each C function we want to call from Python, we need to write its signature in a `.pyf` file. Personally, I prefer to quickly write the C function's signature in Fortran 77, together with appropriate `Cf2py` comments, and then use F2PY to automatically generate the corresponding `.pyf` file. Thereafter, F2PY compiles and links the C code using information in this `.pyf` file. Let us show these steps for our three C functions.

Step 1 consists in writing down the Fortran 77 signatures of the C functions, with `Cf2py` comment specifications for the arguments. By default, F2PY assumes that all C arguments are pointers (since this is the way Fortran treats arguments). An argument `arg1` that is to be passed by value must therefore be marked as `intent(c) arg1` in a `Cf2py` comment. Also the function name must be marked with `intent(c)` to indicate that it is a C function. For our three C functions, the corresponding Fortran signatures with approriate `Cf2py` comments read

```
      real*8 function hw1(r1, r2)
Cf2py intent(c) hw1
```

```
      real*8 r1, r2
Cf2py intent(c) r1, r2
      end

      subroutine hw2(r1, r2)
Cf2py intent(c)  hw2
      real*8 r1, r2
Cf2py intent(c) r1, r2
      end

      subroutine hw3(r1, r2, s)
Cf2py intent(c) hw3
      real*8 r1, r2, s
Cf2py intent(c) r1, r2
Cf2py intent(out) s
      end
```

Running `f2py -m hw -h hw.pyf` on this F77 file results in a `hw.pyf` file we can use together with the C source `hw.c` for building the module with the command `f2py -c hw.pyf hw.c`. The `make.sh` script in the directory

```
src/py/mixed/hw/C/f2py-hw
```

runs the whole recipe plus a test.

Using Ctypes. Recently, Python has been extended with a module `ctypes` for interfacing C code without writing wrapper any code. In the Python program one can load a shared library and call its functions directly, provided that the arguments are of special new `ctypes` types. For example, if you want to send a `float` variable to a C function, you have to convert it to a `c_double` type and send this variable to the C function. Let us demonstrate this for the three C functions in `hw.c`.

The first step consists of making a shared library `hw.so` out of the `hw.c` file:

```
gcc -shared -o hw.so ../hw.c
```

In a Python script we can load this shared library:

```
from ctypes import *
hw_lib = CDLL('hw.so')  # load shared library
```

To call the `hw1` function, which returns a C `double`, we must specify the return value and convert arguments to `c_double`:

```
hw_lib.hw1.restype = c_double
s = hw_lib.hw1(c_double(1), c_double(2.14159))
print s, type(s)
```

The returned value in `s` is automatically converted to Python `float` object.

Instead of explicitly converting each argument to a proper C type from the `ctypes` module, we can once and for all list the argument types for a function and just call the function with ordinary Python data types:

```
hw_lib.hw2.argtypes = [c_double, c_double]
hw_lib.hw1.restype = None  # returns void
hw_lib.hw2(1, 2.14159)
```

Here, we have explicitly specified that the C function returns `void`, by setting `restype` to `None`. Finally, calling `hw3` requires `restype` to be specified, but because of the pointer argument, we must use a `byref(s)` construction for this argument, where `s` is the right C type to be returned by reference. In addition, we must explicitly convert all other arguments to the corresponding C type:

```
s = c_double()
hw_lib.hw3(c_double(1), c_double(2.14159), byref(s))
print s.value
```

Now, `c` is a `ctypes` object and its value is given by `s.value` (a Python `float` in this case). The complete example is found in

```
src/py/mixed/hw/C/ctypes-hw/hwa.py
```

For many C functions, `ctypes` provides an easy way to call the functions directly from Python, but it is also very easy for a beginner to get segmentation faults. I find F2PY to be much safer, quicker, and simpler to use. Although `ctypes` appears to be particularly attractive for interfacing small parts of a big C library that is only available in compiled form, F2PY can also be used to interface compiled C libraries as long as you have a documentation of the C API such that the proper `.pyf` files can be constructed.

Using SWIG. We shall now use the SWIG tool to automatically generate wrapper code for the three C functions in `hw.c`. As will be evident, SWIG requires considerably more manual work than F2PY and `ctypes` to produce the extension module.

Since the creation of an extension module generates several files, it is convenient to work in a separate directory. In our case we work in a subdirectory `swig-hw` of `src/py/mixed/hw/C`.

The Python interface to our C code is defined in what we call a SWIG interface file. Such files normally have the extension `.i`, and we use the name `hw.i` in the current example. A SWIG interface file to our `hw` module could be written as follows:

```
/* file: hw.i */
%module hw
%{
/* include C header files necessary to compile the interface */
#include "hw.h"
%}

double hw1(double r1, double r2);
void   hw2(double r1, double r2);
void   hw3(double r1, double r2, double *s);
```

The syntax of SWIG interface files consists of a mixture of special SWIG directives, C preprocessor directives, and C code. SWIG directives are always preceded by a `%` sign, while C preprocessor directives are recognized by a `#`. SWIG allows comments as in C and C++ in the interface file.

The `%module` directive defines the name of the extension module, here chosen to be `hw`. The `%{ ... }%` block is used for inserting C code necessary for successful compilation of the Python-C interface. Normally this is a collection of header files declaring functions in the module and including the necessary header files from system software and packages that our module depends on.

The next part of the SWIG interface file declares the functions we want to make a Python interface to. Our previously listed interface file contains the signatures of the three functions we want to call from Python. When the number of functions to be interfaced is large, we will normally have a C header file with the signatures of all functions that can be called from application codes. The interface can then be specified by just including this header file, e.g.,

```
%include "hw.h"
```

In the present case, such a header file `hw.h` takes the form

```
#ifndef HW_H
#define HW_H
extern double hw1(double r1, double r2);
extern void   hw2(double r1, double r2);
extern void   hw3(double r1, double r2, double* s);
#endif
```

One can also use `%include` to include other SWIG interface files instead of C header files[5] and thereby merge several separately defined interfaces.

The wrapper code is generated by running

```
swig -python -I.. hw.i
```

SWIG can also generate interfaces in many other languages, including Perl, Ruby, and Tcl. For example, one simply replaces `-python` with `-perl5` to create a Perl interface. The `-I` option tells `swig` where to search for C header files (here `hw.h`). Recall that the source code of our module, `hw.h` and `hw.c`, resides in the parent directory of `swig-hw`. The `swig` command results in a file `hw_wrap.c` containing the C wrapper code, plus a Python module `hw.py`. The latter constitutes our interface to the extension module.

Compiling the Shared Library. The next step is to compile the wrapper code, the C source code with the `hw1`, `hw2`, and `hw3` functions, and link the resulting objects files to form a shared library file `_hw.so`, which constitutes our extension module. Note the underscore prefix in `_hw.so`, this is required

[5] Examples of ready-made interface files that can be useful in other interface files are found in the SWIG manual.

because SWIG generates a Python module `hw.py` that loads `_hw.so`. There are different ways to compile and link the C codes, and two approaches are explained in the following.

A complete manual procedure for compiling and linking our extension module `_hw.so` goes as follows:

```
gcc -I.. -O -I/some/path/include/python2.5 -c ../hw.c hw_wrap.c
gcc -shared -o _hw.so hw.o hw_wrap.o
```

The generated wrapper code in `hw_wrap.c` needs to include the Python header file, and the `-I/some/path/include/Python2.5` option tells the compiler, here `gcc`, where to look for that header file. The path `/some/path` must be replaced by a suitable directory on your system. (If you employ the suggested set-up in Appendix A.1, `/some/path` is given by the environment variable `PREFIX`.) We have also included a `-I..` option to make `gcc` look for header files in the parent directory, where we have the source code for the C functions. In this simple introductory example we do not need header files for the source code so `-I..` has no effect, but its inclusion makes the compilation recipe more reusable.

The second `gcc` command builds a shared library file `_hw.so` out of the object files created by the first command. Occasionally, this second command also needs to link in some additional libraries.

Python knows its version number and where it is installed. We can use this information to write more portable commands for compiling and linking the extension module. The Bash script `make_module_1.sh` in the `swig-hw` directory provides the recipe:

```
swig -python -I.. hw.i

root=‘python -c ’import sys; print sys.prefix’‘
ver=‘python -c ’import sys; print sys.version[:3]’‘
gcc -O -I.. -I$root/include/python$ver -c ../hw.c hw_wrap.c
gcc -shared -o _hw.so hw.o hw_wrap.o
```

Note that we also run SWIG in this script such that all steps in creating the extension module are carried out.

Building the Extension Module Using Distutils. It is a Python standard to write a `setup.py` script to build and install modules with compiled code. A glimpse of a `setup.py` script appears on page 200 together with references to literature on how to write and run such scripts. Here we show how to make a a `setup.py` script for our `hw` module with C files.

Let us first write a version of the `setup.py` script where we use the basic Distutils functionality that comes with the standard Python distribution. The script will then first run SWIG to generate the wrapper code `hw_wrap.c` and thereafter call the Python function `setup` in the Distutils package for compiling and linking the module.

```
import commands, os
from distutils.core import setup, Extension

name = 'hw'           # name of the module
version = 1.0         # the module's version number

swig_cmd = 'swig -python -I.. %s.i' % name
print 'running SWIG:', swig_cmd
failure, output = commands.getstatusoutput(swig_cmd)

sources = ['../hw.c', 'hw_wrap.c']

setup(name = name, version = version,
      ext_modules = [Extension('_' + name,  # SWIG requires _
                               sources,
                               include_dirs=[os.pardir])
                     ])
```

The `setup` function is used to build and install Python modules in general and therefore has many options. Optional arguments are used to control include directories for the compilation (demanded in the current example), libraries to link with, special compiler options, and so on. We refer to the doc string in class `Extension` for more documentation:

```
from distutils.core import Extension
print Extension.__doc__
```

The presented `setup.py` script is written in a generic fashion and should be applicable to any set of C source code files by just editing the `name` and `sources` variables.

In our `setup.py` script we run SWIG manually. We could, in fact, just list the `hw.i` SWIG interface file instead of the C wrapper code in `hw_wrap.c`. SWIG would then be run on the `hw.i` file and the resulting wrapper code would be compiled and linked.

Building the `hw` module is enabled by

```
python setup.py build
python setup.py install --install-platlib=.
```

The first command builds the module in a scratch directory, and the second command installs the extension module in the current working directory (which means copying the shared library file `_hw.so` to this directory).

Using `numpy.distutils`, the building prcoess is simpler as `numpy.distutils` has built-in SWIG support. We just have to list the interface file and the C code as the source files:

```
from numpy.distutils.core import setup, Extension
import os

name = 'hw'           # name of the module
version = 1.0         # the module's version number
sources = ['hw.i', '../hw.c']
```

```
setup(name=name, version=version,
      ext_modules = [Extension('_' + name,  # SWIG requires _
                               sources,
                               include_dirs=[os.pardir])
                     ])
```

Testing the Extension Module. The extension module is not properly built unless we can import it without errors, so the first rough test is

```
python -c 'import hw'
```

We remark that we actually import the Python module in the file `hw.py`, which then imports the extension module in the file `_hw.so`.

The application script on page 194 can be used as is with our C extension module `hw`. Adding calls to the `hw3` function reveals that there is a major problem:

```
>>> from hw import hw3
>>> r1 = 1;  r2 = -1;  s = 10
>>> hw3(r1, r2, s)
TypeError: Type error. Expected _p_double
```

That is, our `s` cannot be passed as a C pointer argument (the subdirectory `error` contains the interface file, compilation script, and test script for this unsuccessful try).

Handling Output Arguments. SWIG offers so-called typemaps for dealing with pointers that represent output arguments from a function. The file `typemaps.i`, which comes with the SWIG distribution, contains some ready-made typemaps for specifying pointers as input, output, or input/output arguments to functions. In the present case we change the declaration of `hw3` as follows:

```
%include "typemaps.i"
void hw3(double r1, double r2, double *OUTPUT);
```

The wrapper code now returns the third argument such that Python must call the function as

```
s = hw3(r1, r2)
```

In other words, SWIG makes a more Pythonic interface to `hw3` (`hw1` and `hw3` then have the same interface as seen from Python). In Chapter 5.2.1 we emphasize that F2PY performs similar adjustments of interfaces to Fortran codes.

The most convenient way of defining a SWIG interface is to just include the C header files of interest instead of repeating the signature of the C functions in the interface file. The special treatment of the output argument `double *s` in the `hw3` function required us in the current example to manually

write up all the functions in the interface file. SWIG has, however, several directives to tweak interfaces such that one can include the C header files with some predefined adjustments. The `%apply` directive can be used to tag some argument names with a, e.g., `OUTPUT` specification:

```
%apply double *OUTPUT { double *s }
```

Any `double *s` in an argument list, such as in the `hw3` function, will now be an output argument.

The above `%apply` directive helps us to specify the interface by just including the whole header file `hw.h`. The interface file thereby gets more compact:

```
/* file: hw2.i, as hw.i but we use %apply and %include "hw.h" */
%module hw
%{
/* include C header files necessary to compile the interface */
/* not required here, but typically
#include "hw.h"
*/
%}

%include "typemaps.i"
%apply double *OUTPUT { double *s }
%include "hw.h"
```

We have called this file `hw2.i`, and a corresponding script for compiling and likning the extension module is `make_module_3.sh`.

5.2.3 Combining Python and C++ Functions

We have also made a C++ version of the `hw1`, `hw2`, and `hw3` functions. The C++ code is not very different from the C code, and the integration of Python and C++ with the aid of SWIG is almost identical to the integration of Python and C as explained in Chapter 5.2.2. You should therefore be familiar with that chapter before continuing.

The C++ version of `hw1`, `hw2`, and `hw3` reads

```
#include <iostream>
#include <math.h>

double hw1(double r1, double r2)
{
  double s = sin(r1 + r2);
  return s;
}

void hw2(double r1, double r2)
{
  double s = sin(r1 + r2);
  std::cout << "Hello, World! sin(" << r1 << "+" << r2
            << ")=" << s << std::endl;
```

```
}

void hw3(double r1, double r2, double* s)
{
  *s = sin(r1 + r2);
}
```

The `hw3` function will normally use a reference instead of a pointer for the `s` argument. This version of `hw3` is called `hw4` in the C++ code:

```
void hw4(double r1, double r2, double& s)
{
  s = sin(r1 + r2);
}
```

The complete code is found in `src/py/mixed/hw/C++/func/hw.cpp`.

We create the extension module in the directory

```
src/py/mixed/hw/C++/func/swig-hw
```

For the `hw1`, `hw2`, and `hw3` functions we can use the same SWIG interface as we developed for the C version of these three functions. To handle the reference argument in `hw4` we can use the `%apply` directive as explained in Chapter 5.2.2. Using `%apply` to handle the output arguments in both `hw3` and `hw4` enables us to define the interface by just including the header file `hw.h`, where all the C++ functions in `hw.cpp` are listed. The interface file then takes the form

```
/* file: hw.i */
%module hw
%{
/* include C++ header files necessary to compile the interface */
#include "hw.h"
%}

%include "typemaps.i"
%apply double *OUTPUT { double* s }
%apply double *OUTPUT { double& s }
%include "hw.h"
```

This file is named `hw.i`. The `hw.h` file is as in the C version, except that the C++ version has an additional line declaring `hw4`:

```
extern void hw4(double r1, double r2, double& s);
```

Running SWIG with C++ code should include the `-c++` option:

```
swig -python -c++ -I.. hw.i
```

The result is then a C++ wrapper code `hw_wrap.cxx` and a Python module file `hw.py`.

The next step is to compile the wrapper code and the C++ functions, and then link the pieces together as a shared library `_hw.so`. A C++ compiler is used for this purpose. The relevant commands, written in Bash and using Python to parameterize where Python is installed and which version we use, may be written as

```
swig -python -c++ -I.. hw.i

root=`python -c 'import sys; print sys.prefix'`
ver=`python -c 'import sys; print sys.version[:3]'`
g++ -O -I.. -I$root/include/python$ver -c ../hw.cpp hw_wrap.cxx
g++ -shared -o _hw.so hw.o hw_wrap.o
```

We are now ready to test the module:

```
>>> import hw
>>> hw.hw2(-1,1)
Hello, World! sin(-1+1)=0
```

Compiling and linking the module can alternatively be done by Distutils and a `setup.py` script as we explained in Chapter 5.2.2. Complete scripts `setup.py` (Python's basic Distutils) and `setup2.py` (`numpy.distutils`) can be found in the directory

```
src/py/mixed/hw/C++/func/swig-hw
```

The four functions in the module are tested in the `hwa.py` script, located in the same directory.

Interfacing C++ code containing classes is a bit more involved, as explained in the next section.

5.2.4 Combining Python and C++ Classes

Chapter 5.2.3 explained how to interface C++ functions, but when we combine Python and C++ we usually work with classes in C++. The present section gives a brief introduction to interfacing classes in C++. To this end, we have made a class version of the `hw` module. A class `HelloWorld` stores the two numbers `r1` and `r2` as well as `s`, where `s=sin(r1+r2)`, as private data members. The public interface offers functions for setting `r1` and `r2`, computing `s`, and writing "Hello, World!" type messages. We want to use SWIG to generate a Python version of class `HelloWorld`.

The Complete C++ Code. Here is the complete declaration of the class and an associated `operator<<` output function, found in the file `HelloWorld.h` in `src/py/mixed/hw/C++/class`:

```
#ifndef HELLOWORLD_H
#define HELLOWORLD_H
#include <iostream>

class HelloWorld
{
 protected:
  double r1, r2, s;
  void compute();    // compute s=sin(r1+r2)
 public:
```

```
  HelloWorld();
   HelloWorld();

  void set(double r1, double r2);
  double get() const { return s; }
  void message(std::ostream& out) const;
};

std::ostream&
operator << (std::ostream& out, const HelloWorld& hw);
#endif
```

The definition of the various functions is collected in `HelloWorld.cpp`. Its content is

```
#include "HelloWorld.h"
#include <math.h>

HelloWorld:: HelloWorld()
{ r1 = r2 = 0;  compute(); }

HelloWorld::  HelloWorld() {}

void HelloWorld:: compute()
{ s = sin(r1 + r2); }

void HelloWorld:: set(double r1_, double r2_)
{
  r1 = r1_;  r2 = r2_;
  compute();  // compute s
}

void HelloWorld:: message(std::ostream& out) const
{
  out << "Hello, World! sin(" << r1 << " + "
      << r2 << ")=" << get() << std::endl;
}

std::ostream&
operator << (std::ostream& out, const HelloWorld& hw)
{ hw.message(out);  return out; }
```

To exemplify subclassing we have made a trivial subclass, implemented in the files `HelloWorld2.h` and `HelloWorld2.cpp`. The header file `HelloWorld2.h` declares the subclass

```
#ifndef HELLOWORLD2_H
#define HELLOWORLD2_H
#include "HelloWorld.h"

class HelloWorld2 : public HelloWorld
{
 public:
  void gets(double& s_) const;
};
#endif
```

The `HelloWorld2.cpp` file contains the body of the `gets` function:

```
#include "HelloWorld2.h"
void HelloWorld2:: gets(double& s_) const { s_ = s; }
```

The `gets` function has a reference argument, intended as an output argument, to exemplify how this is treated in a class context (`gets` is thus a counterpart to the `hw4` function in Chapter 5.2.3).

The SWIG Interface File. In the present case we want to reflect the complete `HelloWorld` class in Python. We can therefore use `HelloWorld.h` to define the interface in the SWIG interface file `hw.i`. To compile the interface, we also need to include the header files in the section after the `%module` directive:

```
/* file: hw.i */
%module hw
%{
/* include C++ header files necessary to compile the interface */
#include "HelloWorld.h"
#include "HelloWorld2.h"
%}

%include "HelloWorld.h"
%include "HelloWorld2.h"
```

With the `double& s` output argument in the `HelloWorld2::gets` function we get the same problem as with the `s` argument in the `hw3` and `hw4` functions. Using the SWIG directive `%apply`, we can specify that `s` is an output argument and thereafter just include the header file to define the interface to the `HelloWorld2` subclass

```
%include "HelloWorld.h"
%include "typemaps.i"
%apply double *OUTPUT { double& s_ }
%include "HelloWorld2.h"
```

The Python call syntax of `gets` reads `s = hw2.gets()` if `hw2` is a `HelloWorld2` instance. As with the `hw3` and `hw4` functions in Chapter 5.2.3, the output argument in C++ becomes a return value in the Python interface.

The `HelloWorld.h` file defines support for printing `HelloWorld` objects. A calling Python script cannot directly make use of this output facility since the "output medium" is an argument of type `std::ostream`, which is unknown to Python. (Sending, e.g., `sys.stdout` to such functions will fail if we have not "swig-ed" `std::ostream`, a task that might be highly non-trivial.) It would be simpler to have an additional function in class `HelloWorld` for printing the object to standard output. Fortunately, SWIG enables us to define additional class functions as part of the interface file. The `%extend` directive is used for this purpose:

```
%extend HelloWorld {
    void print_() { self->message(std::cout); }
}
```

Note that the C++ object is accessed as `self` in functions inside the `%extend` directive. Also note that the name of the function is `print_`: we cannot use `print` since this will interfere with the reserved keyword `print` in the calling Python script. It is a convention to add a single trailing underscore to names coinciding with Python keywords (see page 704).

Making the Extension Module. When the interface file `hw.i` is ready, we can run SWIG to generate the wrapper code:

```
swig -python -c++ -I.. hw.i
```

SWIG issues a warning that the `operator<<` function cannot be wrapped. The files generated by SWIG are `hw_wrap.cxx` and `hw.py`. The former contains the wrapper code, and the latter is a module with a Python mapping of the classes `HelloWorld` and `HelloWorld2`).

Compiling and linking must be done with the C++ compiler:

```
root=‘python -c ’import sys; print sys.prefix’‘
ver=‘python -c ’import sys; print sys.version[:3]’‘
g++ -O -I.. -I$root/include/python$ver \
    -c ../HelloWorld.cpp ../HelloWorld2.cpp hw_wrap.cxx
g++ -shared -o _hw.so HelloWorld.o HelloWorld2.o hw_wrap.o
```

Recall that `_hw.so` is the name of the shared library file when `hw` is the name of the module.

An alternative to the manual procedure above is to write a `setup.py` script, either using Python's standard Distutils or the improved `numpy.distutils`. Examples on both such scripts are found in the directory

```
src/py/mixed/hw/C++/class/swig-hw
```

A simple test script for the generated extension module might take the form

```
import sys
from hw import HelloWorld, HelloWorld2

hw = HelloWorld()
r1 = float(sys.argv[1]);  r2 = float(sys.argv[2])
hw.set(r1, r2)
s = hw.get()
print "Hello, World! sin(%g + %g)=%g" % (r1, r2, s)
hw.print_()

hw2 = HelloWorld2()
hw2.set(r1, r2)
s = hw.gets()
print "Hello, World2! sin(%g + %g)=%g" % (r1, r2, s)
```

Readers who intend to couple Python and C++ via SWIG are strongly encouraged to read the SWIG manual, especially the Python chapter, and study the Python examples that come with the SWIG source code.

Remark on Efficiency. When SWIG wraps a C++ class, the wrapper functions are stand-alone functions, not member functions of a class. For example, the wrapper for the `HelloWorld::set` member function becomes the global function `HelloWorld_set` in the `_hw.so` module. However, SWIG generates a file `hw.py` containing so-called proxy classes, in Python, with the same interface as the underlying C++ classes. A method in a proxy class just calls the appropriate wrapper function in the `_hw.so` module. In this way, the C++ class is reflected in Python. A downside is that there is some overhead associated with the proxy class. For C++ functions called a large number of times from Python, one should consider bypassing the proxy class and calling the underlying function in `_hw.so` directly, or one can write more optimal extension modules by hand, see Chapter 10.3, or one can use SIP which produces more efficient interfaces to C++ code.

5.2.5 Exercises

Exercise 5.1. Implement a numerical integration rule in F77.

Implement the Trapezoidal rule (4.1) from Exercise 4.5 on page 150 in F77 along with a function to integrate and a main program. Verify that the program works (check, e.g., that a linear function is integrated exactly, i.e., the error is zero to machine precision). Thereafter, interface this code from Python and write a new main program in Python calling the integration rule in F77 (the function to be integrated is still implemented in F77). Compare the timings with the plain and vectorized Python versions in the test problem suggested in Exercise 4.5. ◇

Exercise 5.2. Implement a numerical integration rule in C.

As Exercise 5.1, but implement the numerical integration rule and the function to be integrated in C. ◇

Exercise 5.3. Implement a numerical integration rule in C++.

This is an extension of Exercise 5.2. Make an integration rule class hierarchy in C++, where different classes implement different rules. Here is an example on typical usage (in C++):

```
#include <Trapezoidal.h>
#include <math.h>
int main()
{
  MyFunc1 f;           // function object to be integrated
  f.w = 0.11; f.a = 2; // parameters in f
  double a = 1; double b = 2*M_PI/f.w; // integration limits
  int n = 100;     // no of integration points
  Trapezoidal t;   // integration rule
  double I = t.integrate(a, b, f, n);
}
```

The function to be integrated is an object with an overloaded `operator()` function such that the object can be called like an ordinary function (just like the special method `__call__` in Python):

```
class MyFunc1
{
 public:
  double a, w;
  MyFunc1(double a_=1, double w_=1, ) { a=a_; w=w_; }
  virtual double operator() (double x) const
    { return a*exp(-x*x)*log(x + x*sin(w*x)); }
};
```

Implement this code and the `Trapezoidal` class. Use SWIG to make a Python interface to the C++ code, and write the main program above in Python. ⋄

5.3 A Simple Computational Steering Example

A direct Python interface to functions in a simulation code can be used to start the simulation, view results, change parameters, continue simulation, and so on. This is referred to as *computational steering*. The current section is devoted to an initial example on computational steering, where we add a Python interface to a Fortran 77 code. Our simulator is the `oscillator` code from Chapter 2.3. The Fortran 77 implementation of this code is found in

```
src/app/oscillator/F77/oscillator.f
```

The original program reads input data from standard input, computes a time series (by solving a differential equation), and stores the results in a file. You should review the material from Chapter 2.3 before continuing reading.

When steering this application from a Python script we would like to do two core operations in Fortran 77:

- set the parameters in the problem,
- run a number of time steps.

The F77 code stores the parameters in the problem in a common block. This common block can be accessed in the Python code, but assignment strings in this block directly is not recommended. It is safer to send strings from the Python script to the F77 code through a function call and let F77 store the supplied strings in the internal common block variables. Here we employ the same technique for all variables that we need to transfer from Python to Fortran. Fortunately, `oscillator.f` already has a function `scan2` for this purpose:

```
subroutine scan2(m_, b_, c_, A_, w_, y0_, tstop_, dt_, func_)
real*8 m_, b_, c_, A_, w_, y0_, tstop_, dt_
character func_*(*)
```

When it comes to running the simulation a number of steps, the original `timeloop` function in `oscillator.f` needs to be modified for computational steering. Similar adjustments are needed in lots of other codes as well, to enable computational steering.

5.3.1 Modified Time Loop for Repeated Simulations

In computational steering we need to run the simulation for a specified number of time steps or in a specified time interval. We also need access to the computed solution such that it can be visualized from the scripting interface. In the present case it means that we need to write a tailored time loop function working with NumPy arrays and other data structures from the Python code.

The `timeloop` function stores the solution at the current and the previous time levels only. Visualization and arbitrary rewinding of simulations demand the solution to be stored for all time steps. We introduce the two-dimensional array `y` with dimensions `n` and `maxsteps-1` for this purpose. The `n` and `maxsteps` parameters are explained later. Internally, the new time loop routine needs to convert back and forth between the `y` array and the one-dimensional array used for the solution in the `oscillator.f` code. These modifications just exemplify that computational steering usually demands some new functions having different interfaces and working with different data structures compared with the existing functions in traditional codes without support for steering.

Our alternative time loop function, called `timeloop2`, is found in a file `timeloop2.f` in the directory

```
src/py/mixed/simviz
```

The function has the following Fortran signature:

```
subroutine timeloop2(y, n, maxsteps, step, time, nsteps)

integer n, step, nsteps, maxsteps
real*8 time, y(n,0:maxsteps-1)
```

The parameter `n` is the number of components in the system of first-order differential equations, i.e., 2 in the present example. Recall that a second-order differential equation, like (2.1) on page 47, is rewritten as a system of two first-order differential equations before applying standard numerical methods to compute the solution. The unknown functions in the first-order system are y and dy/dt. The `y` array stores the solution of component `i` (y for `i=0` and dy/dt for `i=1`) at time step `j` in the entry `y(i,j)`. That is, discrete values of y are stored in the first row of `y`, and discrete values of dy/dt are stored in the second row.

The `step` parameter is the time step number of the initial time step when `timeloop2` is called. At return, `step` equals the current time step number.

The parameter `time` is the corresponding time value, i.e., initial time when `timeloop2` is called and present time at return. The simulation is performed for `nsteps` time steps, with a time step size `dt`, which is already provided through a `scan2` call and stored in a common block in the F77 code. The `maxsteps` parameter is the total number of time steps that can be stored in `y`.

For the purpose of making a Python interface to `timeloop2`, it is sufficient to know the argument list, that `step` and `time` are input *and* output parameters, that the function advances the solution `nsteps` time steps, and that the computed values are stored in `y`.

5.3.2 Creating a Python Interface

We use F2PY to create a Python interface to the `scan2` and `timeloop2` functions in the F77 files `oscillator.f` and `timeloop2.f`. We create the extension module in a subdirectory `f2py-oscillator` of the directory where `timeloop2.f` is located.

Working with F2PY consists basically of three steps as described on page 478: (i) classifying all arguments to all functions by inserting appropriate `Cf2py` directives, (ii) calling F2PY with standard command-line options to build the module, and (iii) importing the module in Python and printing the doc strings of the module and each of its functions.

The first step is easy: looking at the declaration of `timeloop2`, we realize that `y`, `time`, and `step` are input *and* output parameters, whereas `nsteps` is an input parameter. We therefore insert

```
Cf2py intent(in,out) step
Cf2py intent(in,out) time
Cf2py intent(in,out) y
Cf2py intent(in)     nsteps
```

in `timeloop2`, after the declaration of the subroutine arguments.

The `n` and `maxsteps` parameters are array dimensions and are made optional by F2PY in the Python interface. That is, the F2PY generated wrapper code extracts these parameters from the NumPy objects and feeds them to the Fortran subroutine. We can therefore (very often) forget about array dimension arguments in subroutines.

The second step consists of running the appropriate command for building the module:

```
f2py -m oscillator -c --build-dir tmp1 --fcompiler=Gnu \
   ../timeloop2.f $scripting/src/app/oscillator/F77/oscillator.f \
   only: scan2 timeloop2 :
```

The name of the module (`-m`) is `oscillator`, we demand a compilation and linking (`-c`), files generated by F2PY are saved in the `tmp1` subdirectory (`--build-dir`), we specify the compiler, we list the two Fortran files that

constitute the module, and we restrict the interface to two functions only: `scan2` and `timeloop2`.

The third step tests if the module can be successfully imported and what the interface from Python looks like:

```
>>> import oscillator
>>> print oscillator.__doc__
This module 'oscillator' is auto-generated with f2py (version:...)
Functions:
  y,step,time = timeloop2(y,step,time,nsteps,
                          n=shape(y,0),maxsteps=shape(y,1))
  scan2(m_,b_,c_,a_,w_,y0_,tstop_,dt_,func_)
COMMON blocks:
  /data/ m,b,c,a,w,y0,tstop,dt,func(20)
```

If desired, one can also examine the generated interface file `oscillator.pyf` in the `tmp1` subdirectory.

Notice from the documentation of the `timeloop2` interface that F2PY moves array dimensions, here `n` and `maxsteps`, to the end of the argument list. Array dimensions become keyword arguments with default values extracted from the associated array objects. We can therefore omit array dimensions when calling Fortran from Python. The importance of printing out the extension module's doc string can hardly be exaggerated since the Python interface may have an argument list different from what is declared in the Fortran code.

Looking at the doc string of the `oscillator` module, we see that we get access to the common block in the Fortran code. This allows us to adjust, e.g., the time step parameter `dt` directly from the Python code:

```
oscillator.data.dt = 2.5
```

Support for setting character strings in common blocks is "poor" in the current version of F2PY. However, other data types like `float`, `int`, etc., can safely be set directly in common blocks.

For convenience, the Bourne shell script `make_module.sh`, located in the directory `f2py-oscillator`, builds the module and writes out doc strings.

5.3.3 The Steering Python Script

When operating the `oscillator` code from Python, we want to repeat the following procedure:

- adjust a parameter in Python,
- update the corresponding data structure in the F77 code,
- run a number of time steps, and
- plot the solution.

To this end, we create a function `setprm()` for transferring parameters in the Python script to the F77 code, and a function `run(nsteps)` for running the simulation `nsteps` steps and plotting the solution.

The physical and numerical parameters are variables in the Python script. Their values can be set in a GUI or from command-line options, as we demonstrate in the scripts `simvizGUI2.py` and `simviz1.py` from Chapters 6.2 and 2.3, respectively. However, scripts used to steer simulations are subject to frequent changes so a useful approach is often to just hardcode a set of approprite default values, for instance,

```
m = 1.0; b = 0.7; c = 5.0; func = 'y'; A = 5.0; w = 2*math.pi
y0 = 0.2; tstop = 30.0; dt = 0.05
```

and then assign new values when needed, directly in the script file, or in an interactive Python session, as we shall demonstrate.

The `setprm()` function for transferring the physical and numerical parameters from the Python script to the F77 code is just a short notation for a complete call to the `scan2` F77 function:

```
def setprm():
    oscillator.scan2(m, b, c, A, w, y0, tstop, dt, func)
```

The `run(nsteps)` function calls the `timeloop2` function in the `oscillator` module and plots the solution. We have here chosen to exemplify how the `Gnuplot` module can be used directly to plot array data:

```
from scitools.numpyutils import seq, zeros
maxsteps = 10000
n = 2
y = zeros((n,maxsteps))
step = 0; time = 0.0

import Gnuplot
g1 = Gnuplot.Gnuplot(persist=1)  # (y(t),dy/dt) plot
g2 = Gnuplot.Gnuplot(persist=1)  # y(t) plot

def run(nsteps):
    global step, time, y
    if step+nsteps > maxsteps:
        print 'no more memory available in y'; return

    y, step, time = oscillator.timeloop2(y, step, time, nsteps)

    t = seq(0.0, time, dt)
    y1 = y[0,0:step+1]
    y2 = y[1,0:step+1]
    g1.plot(Gnuplot.Data(y1,y2, with='lines'))
    g2.plot(Gnuplot.Data(t, y1, with='lines'))
```

In the present case we use 0 as base index for `y` in the Python script (required) and 1 in the F77 code. Such "inconsistency" is unfortunately a candidate for bugs in numerical codes, but 1 as base index is a common habit in Fortran routines so it might be an idea to illustrate how to deal with this.

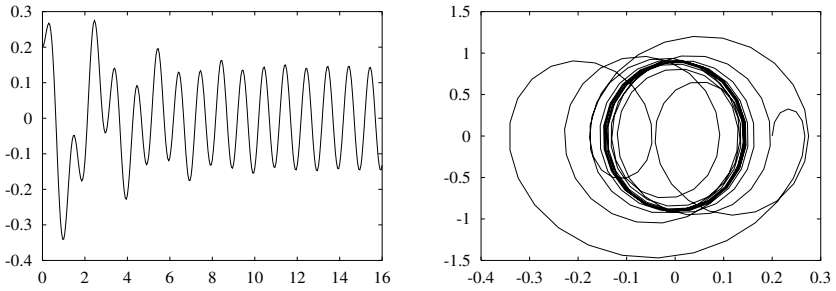

Fig. 5.1. Plots produced by an interactive session involving the `oscillator` module, as explained in Chapter 5.3.3. To the left is the displacement $y(t)$, and to the right is the trajectory $(y(t), y'(t))$.

The first plot is a phase space curve $(y, dy/dt)$, easily created by extracting the steps `0` up to, but not including, `step+1`. We can write the extraction compactly as `y[:,0:step+1]`. To plot the $y(t)$ curve, we extract the first component of the solution for the same number of steps: `y[0,0:step+1]`. The corresponding t values are stored in an array `t` (note that we use `seq` from `scitools.numpyutils` to ensure that the upper limit, `time`, is included as last element, cf. Chapter 4.3.7).

A complete steering Python module is found in

```
src/py/mixed/simviz/f2py/simviz_steering.py
```

This module imports the `oscillator` extension module, defines physical parameters such as `m`, `b`, `c`, etc., and the previously shown `setprm` and `run` functions, plus more to be described later.

Let us demonstrate how we can perform a simulation in several steps. First, we launch a Python shell (IPython or the IDLE shell) and import the steering interface to the `oscillator` program:

```
from simviz_steering import *
```

We can now issue commands like

```
setprm()     # send default values to the oscillator code
run(60)      # simulate the first 60 time steps

w = math.pi  # change the frequency of the applied load
setprm()     # notify simulator about any parameter change
run(120)     # simulate for another 120 steps

A = 10       # change the amplitude of the applied load
setprm()
run(100)
```

The `run` function updates the solution in a plot on the screen so we can immediately see the effect of changing parameters and running the simulator.

To rewind the simulator `nsteps`, and perhaps change parameters and rerun some steps, the `simviz_steering` module contains the function

```
def rewind(nsteps=0):
    global step, time
    if nsteps == 0:  # start all over again?
        step = 0
        time  = 0.0
    else:            # rewind nsteps
        step -= nsteps
        time -= nsteps*dt
```

Here is an example in the interactive shell:

```
>>> from simviz_steering import *
>>> run(50)
>>> rewind(50)
>>> A=20
>>> setprm()
>>> run(50)   # try again the 50 steps, now with A=20
```

A session where we check the effect of changing the amplitude and frequency of the load during the simulation can look like this:

```
>>> rewind()
>>> A=1; setprm(); run(100)
>>> run(300)
>>> rewind(200)
>>> A=10; setprm(); run(200)
>>>> rewind(200)
>>> w=1; setprm(); run(400)
```

With the following function from `simviz_steering.py` we can generate hardcopies of the plots when desired:

```
def psplot():
    g1.hardcopy(filename='tmp_phaseplot_%d.ps' % step,
                enhanced=1, mode='eps', color=0,
                fontname='Times-Roman', fontsize=28)
    g2.hardcopy(filename='tmp_y1_%d.ps' % step,
                enhanced=1, mode='eps', color=0,
                fontname='Times-Roman', fontsize=28)
```

Hopefully, the reader has realized how easy it is to create a dynamic working environment where functionality can be added on the fly with the aid of Python scripts.

Remark. You should not change `dt` during a simulation without a complete rewind to time zero. The reason is that the `t` array used for plotting $y_1(t)$ is based on a constant time step during the whole simulation. However, recomputing the solution with a smaller time step is often necessary if the first try leads to numerical instabilities.

5.3.4 Equipping the Steering Script with a GUI

We can now easily combine the `simviz_steering.py` script from the last section with the GUI `simvizGUI2.py` from Chapter 6.2. The physical and numerical parameters are fetched from the GUI, sent to the `oscillator` module by calling its `scan2` function, and when we press Compute in the GUI, we call up the `run` function to run the Fortran code and use Gnuplot to display results. That is, we have a GUI performing function calls to the simulator code and the visualization program. This is an alternative to the file-based communication in Chapter 6.2.

The GUI code could be placed at the end of the `simviz_steering` module. A better solution is to import `simviz_steering` in the GUI script. We want the GUI script to run the initializing statements in `simviz_steering`, and this will be done by a straight

```
import simviz_steering as S
```

statement.

It would be nice to have a slider reflecting the number of steps in the solution. Dragging this slider backwards and clicking on compute again will then correspond to rewinding the solution and repeating the simulation, with potentially new physical or numerical data. All we have to do in the constructor in class `SimVizGUI` is

```
self.p['step'] = IntVar(); self.p['step'].set(0)
self.slider(slider_frame, self.p['step'], 0, 1000, 'step')
```

The `self.compute` function in the `simvizGUI2.py` script must be completely rewritten (we do not launch `simviz1.py` as a stand-alone script anymore):

```
def compute(self):
    """run oscillator code"""
    rewind_nsteps = S.step - self.p['step'].get()
    if rewind_nsteps > 0:
        print 'rewinding', rewind_nsteps, 'steps, ',
        S.rewind(rewind_nsteps)  # adjust time and step
        print 'time =', S.time
    nsteps = int((self.p['tstop'].get()-S.time)\
                 /self.p['dt'].get())
    print 'compute', nsteps, 'new steps'
    self.setprm() # notify S and oscillator about new parameters
    S.run(nsteps)
    # S.step is altered in S.run so update it:
    self.p['step'].set(S.step)
```

The new `self.setprm` function looks like

```
def setprm(self):
    """transfer GUI parameters to oscillator code"""
    # safest to transfer via simviz_steering as that
    # module employs the parameters internally:
```

```
S.m = self.p['m'].get(); S.b = self.p['b'].get()
S.c = self.p['c'].get(); S.A = self.p['A'].get()
S.w = self.p['w'].get(); S.y0 = self.p['y0'].get()
S.tstop = self.p['tstop'].get()
S.dt = self.p['dt'].get(); S.func = self.p['func'].get()
S.setprm()
```

These small modifications to `simvizGUI.py` have been saved in a new file

```
src/py/mixed/simviz/f2py/simvizGUI_steering.py
```

Run that file, set tstop to 5, click Compute, watch that the step slider has moved to 100, change the m slider to 5, w to 0.1, tstop to 40, move step back to step 50, and click Compute again.

The resulting application is perhaps not of much direct use in science and engineering, but it is sufficiently simple and general to demonstrate how to glue simulation, visualization, and GUIs by sending arrays and other variables between different codes. The reader should be able to extend this introductory example to more complicated applications.

5.4 Scripting Interfaces to Large Libraries

The information on creating Python interfaces to Fortran, C, and C++ codes so far in this chapter have been centered around simple educational examples to keep the focus on technical details. Migration of slow Python code to complied languages will have a lot in common with these examples. However, one important application of the technology is to generate Python interfaces to existing codes. How does this work out in practice for large legacy codes? The present section shares some experience from interfacing the C++ library Diffpack [15].

About Diffpack. Diffpack is a programming environment aimed at scientists and engineering who develop codes for solving partial differential equations (PDEs). Diffpack contains a huge C++ library of numerical functionality needed when solving PDEs. For example, the library contains class hierarchies for arrays, linear systems, linear system solvers and preconditioners, grids and corresponding fields for finite difference, element, and volume methods, as well as utilities for data storage, adaptivity, multi-level methods, parallel computing, etc. To solve a specific PDE, one must write a C++ program, which utilizes various classes in the Diffpack library to perform the basic steps in the solution method (e.g., generate mesh, compute linear system, solve linear system, store solution).

Diffpack comes with lots of example programs for solving basic equations like wave equations, heat equations, Poisson equations, nonlinear convection-diffusion equations, the Navier-Stokes equations, the equations of linear elasticity and elasto-viscoplasticity, as well as systems of such equations. Many

of these example programs are equipped with scripts for automating simulation and visualization [15]. These scripts are typically straightforward extensions of the `simviz1.py` (Chapter 2.3) and `simvizGUI2.py` (Chapter 6.2) scripts heavily used throughout the present text. Running Diffpack simulators and visualization systems as stand-alone programs from a tailored Python script may well result in an efficient working environment. The need to use C++ functions and classes directly in the Python code is not critical for a ready-made Diffpack simulator applied in a traditional style.

During program development, however, the request for calling Diffpack directly from Python scripts becomes evident. Code is changing quickly, and convenient tools for rapid testing, dumping of data, immediate visualization, etc., are useful. In a way, the interactive Python shell may in this case provide a kind of problem-specific scientific debugger. Doing such dynamic testing and developing is more effective in Python than in C++. Also when it comes to gluing Diffpack with other packages, without relying on stand-alone applications with slow communication through files, a Python-Diffpack interface is of great interest.

Using SWIG. At the time of this writing, we are trying to interface the whole Diffpack library with the aid of SWIG. This is a huge task because a robust interface requires many changes in the library code. For example, `operator=` and the copy constructor of user-defined classes are heavily used in the wrapper code generated by SWIG. Since not all Diffpack classes provided an `operator=` or copy constructor, the default versions as automatically generated by C++ were used "silently" in the interface. This led in some cases to strange behavior whose reason was difficult to find. The problem was absent in Diffpack, simply because the problematic objects were (normally) not used in a context where `operator=` and the copy constructor were invoked. Most of the SWIG-induced adjustments of Diffpack are technically sound, also in a pure C++ context. The main message here is simple: C++ code developers must be prepared for some adjustments of the source before generating scripting interfaces via SWIG.

Earlier versions of SWIG did not support macros, templates, operator overloading, and some more advanced C++ features. This has improved a lot with the SWIG version 1.3 initiative. Now quite complicated C++ can be handled. Nevertheless, Diffpack applies macros in many contexts, and not all of the macros were satisfactorily handled by SWIG. Our simplest solution to the problem was to run the C++ preprocessor and automatically (via a script) generate (parts of) the SWIG interface based on the preprocessor output with macros expanded.

Wrapping Simulators. Rather than wrapping the complete Diffpack library, one can wrap the C++ simulator, i.e. the "main program", for solving a specific PDE, as this is a much simpler and limited task. Running SWIG successfully on the simulator header files requires some guidelines and automation scripts. Moreover, for such a Python interface to be useful, some

of the most important classes in the Diffpack library must also be wrapped and used from Python scripts. The techniques and tools for wrapping simulators are explained in quite some detail in [17]. Here we shall only mention some highlights regarding the technical issues and share some experience with interfacing Python and a huge C++ library.

Preprocessing header files to expand macros and gluing the result automatically in the SWIG interface file is performed by a script. The interface file can be extended with extra access functions, but the automatically generated file suffices in many cases.

Compiling and Linking. The next step in creating the interface is to compile and link Diffpack and the wrapper code. Since Diffpack relies heavily on makefiles, compiling the wrapper code is easiest done with SWIG's template makefiles. These need access to variables in the Diffpack makefiles so we extended the latter with a functionality of dumping key information, in form of make variables, to a file, which then is included in the SWIG makefile. In other words, tweaking makefiles from two large packages (SWIG and Diffpack) was a necessary task. With the aid of scripts and some adjustments in the Diffpack makefiles, the compilation and linking process is now fully automatic: the extension module is built by simply writing `make`. The underlying makefile is automatically generated by a script.

Converting Data Between Diffpack and Python. Making Python interfaces to the most important Diffpack classes required a way of transferring data between Python and Diffpack. Data in this context is usually potentially very large arrays. By default, SWIG just applies pointers, and this is efficient, but unsafe. Our experience so far is that copying data is the recommended default behavior. This is safe for newcomers to the system, and the copying can easily be replaced by efficient pointer communication for the more advanced Python-SWIG-Diffpack developer. Copying data structures back and forth between Diffpack and Python can be based on C++ code (conversion classes, as explained in Chapter 10.3.3) or on SWIG's typemap facility. We ended up with typemaps for the simplest and smallest data structures, such as strings, while we used filters for arrays and large data structures. Newcomers can more easily inspect C++ conversion functions than typemaps to get complete documentation of how the data transfer is handled.

Basically, the data conversion takes place in static functions. For example, a NumPy array created in Python may be passed on as the array of grid point values in a Diffpack field object, and this object may be transformed to a corresponding Vtk object for visualization.

Visualization with Vtk. The visualization system Vtk comes with a Python interface. This interface lacks good documentation, but the source code is well written and represented satisfactory documentation for realizing the integration of Vtk, Python, and Diffpack. Any Vtk object can be converted into a `PyObject` Python representation. That is, Vtk is completely wrapped

in Python. For convenience we prefer to call Vtk through MayaVi, a high-level interface to Vtk written in Python.

Example on a Script. Below is a simple script for steering a simulation involving a two-dimensional, time-dependent heat equation. The script feeds input data to the simulator using Diffpack's menu system. After solving the problem the solution field (temperature) is grabbed and converted to a Vtk field. Then we open MayaVi and specify the type of visualization we want.

```
from DP import *          # import some Diffpack library utilities
from Heat1 import *       # import heat equation simulator
menu = MenuSystem()       # enable programming Diffpack menus
...                       # some init of the menu system
heat = Heat1()            # make simulator object
heat.define(menu)         # generate input menus
grid_str = 'P=PreproBox | d=2 [0,1]x[0,1] | d=2 e=ElmB4n2D '\
           'div=[16,16] grading=[1,1]'
menu.set('gridfile', grid_str) # send menu commands to Diffpack
heat.scan()               # load menu and initialize data structs
heat.solveProblem()       # solve PDE problem

dp2py = dp2pyfilters()    # copy filters for Diffpack-Vtk-Python
import vtk, mayavi        # interfaces to Vtk-based visualization
vtk_field = dp2py.dp2vtk(heat.u())  # solution u -> Vtk field

v = mayavi.mayavi()       # use MayaVi for visualization
v_field = v.open_vtk_data(vtk_field)

m = v.load_module('SurfaceMap', 0)
a = v.load_module('Axes', 0)
a.axes.SetCornerOffset(0.0) # configure the axes module
o = v.load_module('Outline', 0)
f = v.load_filter('WarpScalar', config=0)
config_file = open('visualize.config')
f.load_config(config_file)
v.Render()  # plot the temperature
```

Reference [17] contains more examples. For instance, in [17] we set up a loop over discretization parameters in the steering Python script and compute convergence rates of the solution using the nonlinear least squares module in ScientificPython.