Chapter 9

Fortran Programming with Numerical Python Arrays

Python loops over large array structures are known to run slowly. Tests with class `Grid2D` from Chapter 4.3.5 show that filling a two-dimensional array of size 1100×1100 with nested loops in Python may require about 150 times longer execution time than using Fortran 77 for the same purpose. With Numerical Python (NumPy) and vectorized expressions (from Chapter 4.2) one can speed up the code by a factor of about 50, which gives decent performance.

There are cases where vectorization in terms of NumPy functions is demanding or inconvenient: it would be a lot easier to just write straight loops over array structures in Fortran, C, or C++. This is quite easy, and the details of doing this in F77 are explained in the present chapter. Chapter 10 covers the same material in a C and C++ context.

The forthcoming material assumes that you are familiar with at least Chapter 5.2.1. Familiarity with Chapter 5.3 as well is an advantage.

9.1 Problem Definition

The programming in focus in the present chapter concerns evaluating a function `f(x,y)` over a rectangular grid and storing the values in an array. The algorithm is typically

```
for i = 1, ..., nx
    for j = 1, ..., ny
        a[i,j] = f(xcoor[i], ycoor[j])
```

The x and y coordinates of the grid are stored in one-dimensional arrays `xcoor` and `ycoor`, while the value of the function `f` at grid point `i,j` is stored in the `i,j` entry of a two-dimensional array `a`.

Chapter 4.3.5 documents a Python class, named `Grid2D`, for representing two-dimensional, rectangular grids and evaluating functions over the grid. Visiting the grid points in plain Python loops and evaluating a function for every point is a slow process, but vectorizing the evaluation gives a remarkable speed-up. Instead of employing vectorization we could migrate the straight double loop to compiled code. The details of this migration are explained on the following pages.

You should be familiar with Chapter 4.3.5 before proceeding with the current section. The `Grid2D` class has a method `gridloop(self,f)` implementing the loop over the grid points. For each point, the function `f` is called, and the returned value is inserted in an output array `a`:

```
def gridloop(self, f):
    nx = size(self.xcoor); ny = size(self.ycoor)
    a = zeros((nx,ny), Float)
    for i in range(nx):
        x = self.xcoor[i]
        for j in range(ny):
            y = self.ycoor[j]
            a[i,j] = f(x, y)
    return a
```

The `gridloop` method is typically used as follows:

```
# g is some Grid2D object
def myfunc(x, y): return x + 2*y
f = g.gridloop(myfunc)  # compute f array of grid point values
i=1; j=5                # grid point (i,j)
print 'myfunc at (%g,%g) = f[%d,%d] = %g' % \
    (g.xcoor[i], g.ycoor[i], i, j, f[i,j])
```

The first simple way of speeding up the `gridloop` function is to apply Psyco. In Chapter 8.10.4 we present a simple example on compiling a Python function with the aid of Psyco. In the present application we may introdce a compilation function,

```
def gridloop_psyco_init(self):
    import psyco
    self.gridloop_psyco = psyco.proxy(self.gridloop)
```

Observe that we here define a new method `gridloop_psyco`. The usage consists of first calling `g.gridloop_psyco_init()` to initialize the Psyco accelerated version of the `gridloop` method. Thereafter we use `g.gridloop_psyco` instead of `g.gridloop`. Unfortunately, Psyco seldom gives a speed-up factor of more than two for such loop applications (see Chapters 8.10.4 and 10.3.1).

Migrating the inner details of `gridloop` to Fortran, C, or C++ can take place in a subclass `Grid2Deff` of `Grid2D`. A first outline of that subclass might be

```
from Grid2D import *
import ext_gridloop   # extension module in Fortran, C, or C++

class Grid2Deff(Grid2D):

    def ext_gridloop1(self, f):
        a = zeros((size(self.xcoor),size(self.ycoor)), Float)
        ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, f)
        return a

    def ext_gridloop2(self, f):
        a = ext_gridloop.gridloop2(self.xcoor, self.ycoor, f)
        return a
```

Two different implementations of the external `gridloop` function are realized:

- `ext_gridloop1` calls a Fortran, C, or C++ function `gridloop1` with the array of function values as argument, intended for in-place modifications.
- `ext_gridloop2` calls a Fortran, C, or C++ function `gridloop2` where the array of function values is created and returned.

The `gridloop1` call follows the typical communication pattern in Fortran, C, and C++: both input and output arrays are transferred as arguments to the function. The latter function, `gridloop2`, is more Pythonic: input arrays are arguments and the output array is returned. Observe that in both calls we omit array dimensions despite the fact that these are normally explicitly required in F77 and C routines. From the Python side the array dimensions are a part of the array object, so when the compiled code needs explict dimensions, the wrapper code should retrieve and pass on this information.

Since Python does not bother about what type of language we have used in the extension module, we use the same Python script for working with the `ext_gridloop` module, regardless of whether the loops have been migrated to Fortran, C, or C++. This script is found as the file

```
src/py/mixed/Grid2D/Grid2Deff.py
```

The source code of the `ext_gridloop1` and `ext_gridloop2` functions are located in subdirectories of `src/py/mixed/Grid2D`: `F77`, `C`, and `C++`.

9.2 Filling an Array in Fortran

It turns out that writing `gridloop2` in Fortran and calling it as shown in Chapter 9.1 is easy to explain and carry out with F2PY. On the contrary, implementating the `gridloop1` and calling it with `a` as an argument to be changed in-place is a much harder task. We therefore leave the `gridloop1` details to Chapter 9.3 and deal with `gridloop2` in the forthcoming text.

9.2.1 The Fortran Subroutine

The rule of thumb when using F2PY is to *explicitly classify all (output) arguments to a Fortran function*, either by editing the `.pyf` file or by adding `Cf2py intent` comments in the Fortran source code. In our case `xcoor` and `ycoor` are input arrays, and `a` is an output array. The `nx` and `ny` array dimensions are also input parameters, but the F2PY generated wrapper code will automatically extract the array dimensions from the NumPy objects and pass them on to the Fortran routine so we do not need to do anything with the `nx` and `ny` arguments.

The `gridloop2` routine is a Fortran implementation of the `gridloop` method in class `Grid2D`, see Chapter 9.1:

```
      subroutine gridloop2(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func1
      external func1
Cf2py intent(out) a
Cf2py intent(in) xcoor
Cf2py intent(in) ycoor
Cf2py depend(nx,ny) a

      integer i,j
      real*8 x, y
      do j = 0, ny-1
         y = ycoor(j)
         do i = 0, nx-1
            x = xcoor(i)
            a(i,j) = func1(x, y)
         end do
      end do
      return
      end
```

Here we specify that `a` is an output argument (`intent(out)`), whereas `xcoor` and `ycoor` are input arguments (`intent(in)`). Moreover, the size of `a` depends on `nx` and `ny` (the lengths of `xcoor` and `ycoor`).

Python arrays always start with zero as base index. In Fortran, 1 is the default base index. Declaring `a(nx,ny)` implies that `a`'s indices run from 1 to `nx` and 1 to `ny`. When the base index differs from 1, it has to be explicitly written in the dimensions of the array, as in `a(0:nx-1,0:ny-1)`. It is in general a good idea to employ exactly the same indexing in Fortran and Python – this simplifies debugging.

9.2.2 Building and Inspecting the Extension Module

The next step is to run F2PY on the source code file

```
src/py/mixed/Grid2D/F77/gridloop.f
```

containing the shown `gridloop2` routine. A simple `f2py` command builds the extension module:

```
f2py -m ext_gridloop -c gridloop.f
```

A first test is to run

```
python -c 'import ext_gridloop'
```

Thereafter we should always print out the doc string of the generated extension module or the function of interest, here `gridloop2`:

```
python -c 'import ext_gridloop; \
           print ext_gridloop.gridloop2.__doc__'
```

The output becomes

```
gridloop2 - Function signature:
  a = gridloop2(xcoor,ycoor,func1,[nx,ny,func1_extra_args])
Required arguments:
  xcoor : input rank-1 array('d') with bounds (nx)
  ycoor : input rank-1 array('d') with bounds (ny)
  func1 : call-back function
Optional arguments:
  nx := len(xcoor) input int
  ny := len(ycoor) input int
  func1_extra_args := () input tuple
Return objects:
  a : rank-2 array('d') with bounds (nx,ny)
Call-back functions:
  def func1(x,y): return func1
  Required arguments:
    x : input float
    y : input float
  Return objects:
    func1 : float
```

Observe that `a` is removed from the argument list and appears as a return value. Also observe that the array dimensions `nx` and `ny` are moved to the end of the argument list and given default values based on the input arrays `xcoor` and `ycoor`. In a pure Python implementation we would just have `xcoor`, `ycoor`, and `func1` as arguments and then create `a` inside the function and return it, since the Pythonic programming standard is to use arguments for input data and return output data. F2PY supports this style of programming. A typical call in a `Grid2Deff` method reads

```
a = ext_gridloop.gridloop2(self.xcoor, self.ycoor, f)
```

If desired, we can supply the dimension arguments:

```
a = ext_gridloop.gridloop2(self.xcoor, self.ycoor, myfunc,
                  self.xcoor.shape[0], self.ycoor.shape[0])
```

F2PY will in this case check consistency between the supplied dimension arguments and the actual dimensions of the arrays.

Another noticeable feature of F2PY is that it successfully detects that `func1` is a callback to Python. This makes it convenient to supply the compiled extension module with mathematical expressions coded in Python. F2PY also enables us to send arguments from Python "through" Fortran and to the callback function with the aid of the additional `func1_extra_args` argument. This is demonstrated in Chapter 9.4.1. Unfortunately, the callback to Python is very expensive, as demonstrated by efficiency tests in Chapter 10.3.1, but there are methods for improving the efficiency, see Chapter 9.4.

The `depend(nx,ny) a` specification as a `Cf2py` comment is important. Without it, F2PY will let `nx` and `ny` be optional arguments that depend on `a`, but we do not supply `a` in the call. The `depend` directive ensures that `a`'s size

depends on the `nx` and `ny` parameters of the supplied `xcoor` and `ycoor` array objects.

When we specify that `a` is output data, F2PY will generate an interface where `a` is not an argument to the function. This may be annoying for programmers coming from Fortran to Python, but employing the habit of always printing the doc strings of the wrapped module helps to make the usage smooth.

Let us check that the interface works:

```
def f1(x,y):
    return x+2*y

def verify1():
    g = Grid2Deff(dx=0.5, dy=1)
    f_exact = g(f1)          # NumPy computation
    f = g.ext_gridloop2(f1)  # extension module call
    if allclose(f, f_exact, atol=1.0E-10, rtol=1.0E-12):
        print 'f is correct'
```

Executing `verify1` demonstrates that the `gridloop2` subroutine computes (approximately) the same values as the pure Python method `__call__` inherited from class `Grid2D`.

9.3 Array Storage Issues

Many newcomers to F2PY and Python may consider the call to `gridloop2` as less natural than the call to `gridloop1`. Since both Fortran routines must take the output array as a positional argument, the natural call would seemingly be to allocate `a` in Python, send it to Fortran, and let it be filled in Fortran in a call by reference fashion (cf. Chapter 3.3.4). Calling the `gridloop1` routine this way straight ahead, i.e., without noticing F2PY that `a` is an *output* array, leads to wrong results. We shall dive into this problem in Chapters 9.3.1–9.3.3. This material will in detail explain some important issues about array storage in Fortran and C/NumPy. The topics naturally lead to a discussion of F2PY interface files in Chapter 9.3.4 and a nice F2PY feature in Chapter 9.3.5 for hiding F77 work arrays from a Python interface.

9.3.1 Generating an Erroneous Interface

The version of `gridloop1` without any `intent Cf2py` comments are in the source code file `gridloop.f` called `gridloop1_v1` (the `_v1` extension indicates that we need to experiment with several versions of `gridloop1` to explore various feature of F2PY):

```
      subroutine gridloop1_v1(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
```

```
real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func1
external func1
integer i,j
real*8 x, y
do j = 0, ny-1
   y = ycoor(j)
   do i = 0, nx-1
      x = xcoor(i)
      a(i,j) = func1(x, y)
   end do
end do
return
end
```

Running a straight `f2py` build command,

```
f2py -m ext_gridloop -c gridloop.f
```

and printing out the doc string of the `gridloop1_v1` function yields

```
gridloop1_v1 - Function signature:
  gridloop1_v1(a,xcoor,ycoor,func1,[nx,ny,func1_extra_args])
Required arguments:
  a : input rank-2 array('d') with bounds (nx,ny)
  xcoor : input rank-1 array('d') with bounds (nx)
  ycoor : input rank-1 array('d') with bounds (ny)
  func1 : call-back function
Optional arguments:
  nx := shape(a,0) input int
  ny := shape(a,1) input int
  func1_extra_args := () input tuple
Call-back functions:
  def func1(xi,yj): return func1
  ...
```

Running simple tests reveal that whatever functions we provide as the `func1` argument, `a` is always zero after the call. A `write` statement in the do loops shows that correct values are indeed inserted in the array `a` in the Fortran subroutine. The problem is that the values inserted in the `a` array in Fortran are not visible in what we think is the same `a` array in the Python code.

We may investigate the case further by making a simple subroutine for changing arrays in Fortran:

```
subroutine change(a, xcoor, ycoor, nx, ny)
integer nx, ny
real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
integer j
do j = 0, ny-1
   a(1,j) = -999
end do
xcoor(1) = -999
ycoor(1) = -999
return
end
```

This function is simply added to the `gridloop.f` file defining the extension module. A small Python test,

```
from py4cs.numpytools import *
xcoor = sequence(0, 1, 0.5, Float)
ycoor = sequence(0, 1, 1, Float)
a = zeros((size(xcoor),size(ycoor)), Float)
import ext_gridloop
ext_gridloop.change(a, xcoor, ycoor)
print 'a after change:\n', a
print 'xcoor after change:\n', xcoor
print 'ycoor after change:\n', ycoor
```

leads to the output

```
a after change:
[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
xcoor after change:
[ -9.99000000e+02   5.00000000e-01   1.00000000e+00]
ycoor after change:
[-999.    1.]
```

We observe that the changes made to `xcoor` and `ycoor` in the `change` subroutine are visible in the calling code, but the changes made to `a` are not.

Why do our interfaces to the `gridloop1` and `change` Fortran routines fail to work as intended? The short answer is that `a` would be correctly modified if it was declared as an `intent(out)` argument in the `change` subroutine. A more comprehensive answer needs a discussion of how multi-dimensional arrays are stored differently in Fortran and NumPy and how this affects the usage of F2PY. With an understanding of these issues from Chapters 9.3.2 and 9.3.3 we can eventually call `gridloop1` as originally intended from Python.

Although we quickly solve our loop migration problem in Chapter 9.2, I strongly recommend to read Chapters 9.3.2 and 9.3.3 because the understanding of multi-dimensional storage issues when combining Python and Fortran is essential for avoiding unnecessary copying and obtaining efficient code.

9.3.2 Array Storage in C and Fortran

Multi-dimensional arrays are stored differently in C and Fortran. C stores a two-dimensional array row by row, while Fortran stores it column by column. Common terminology refers to *row major storage* (C) and *column major storage* (Fortran). This generalizes to higher dimensions such that, in Fortran, the first index runs faster than the second index, and so on, whereas in C the first index runs slower than the second, and so forth, with the last index as the fastest one. NumPy arrays apply the storage scheme of C, i.e., row major storage. Figure 9.1 illustrates the differences in storage.

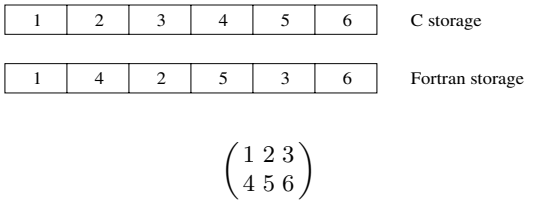

Fig. 9.1. Storage of a 2×3 matrix in C/C++/NumPy (upper) and Fortran (lower).

Since our `a` array is created in Python with the aid of NumPy, we send to `gridloop1_v1` an array with row major storage. To keep the storage scheme transparent, F2PY copies this array to a new one, with column major storage, in the wrapper code. This new array is sent to the Fortran function. Because `a` is classified as an input (and not output) argument *by default* – note that the doc string explicitly tells us that `a` is an input array – F2PY thinks it is safe to work on a copy of `a`. Correct values are computed and inserted in the copy, but the calling code never gets the copy back. That is why we experience that `a` in the Python code is unaltered after the call.

The changes to `xcoor` and `ycoor` in the `change` function are visible in the calling Python code because these arrays are one-dimensional. Fortran and C store one-dimensional arrays in the same way so F2PY does not make a copy for transposing data. Changes are then done in-place.

F2PY can be compiled with the flag

```
-DF2PY_REPORT_ON_ARRAY_COPY=1
```

to make the wrapper code report every copy of arrays. In the present case, the wrapper code will write out

```
copied an array using copy_ND_array: size=6, elsize=8
```

indicating that a copy takes place. When nothing is returned from the subroutine `gridloop1_v1`, we never get our hands on the copy.

9.3.3 Input and Output Arrays as Function Arguments

Arrays Declared as `intent(in,out)`. Quite often an array argument is both input *and* output data in a Fortran function. Say we have a Fortran function `gridloop3` that *adds* values computed by a `func1` function to the `a` array:

```
      subroutine gridloop3(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func1
Cf2py intent(in,out) a
Cf2py intent(in) xcoor
```

```
Cf2py intent(in) ycoor

      external func1
      integer i,j
      real*8 x, y
      do j = 0, ny-1
         y = ycoor(j)
         do i = 0, nx-1
            x = xcoor(i)
            a(i,j) = a(i,j) + func1(x, y)
         end do
      end do
      return
      end
```

In this case, we specify `a` as `intent(in,out)`, i.e., an input *and* output array. F2PY generates the following interface:

```
a = gridloop3(a,xcoor,ycoor,func1,[nx,ny,func1_extra_args])
```

We may write a small test program:

```
from py4cs.numpytools import *
xcoor = sequence(0, 1, 0.5, Float)
ycoor = sequence(0, 1, 1, Float)
a = zeros((size(xcoor),size(ycoor)), Float)
print xcoor, ycoor
def myfunc(x, y):  return x + 2*y
a[:,:] = -1
a = ext_gridloop.gridloop3(a, xcoor, ycoor, myfunc)
print 'a after gridloop3:\n', a
```

The output becomes

```
x = [ 0.   0.5  1. ]
y = [ 0.  1.]
a after gridloop3:
[[-1.   1. ]
 [-0.5  1.5]
 [ 0.   2. ]]
```

Figure 9.2 sketches how the grid looks like. Examining the output values in the light of Figure 9.2 shows that the values are correct. The `a` array is stored as usual in NumPy. That is, there is no effect of storage issues when computing `a` in Fortran and printing it in Python. The fact that the `a` array in Fortran is the transpose of the initial and final `a` array in Python becomes transparent when using F2PY.

Arrays Declared as `intent(inout)`. Our goal now is to get the `ext_gridloop1` to work in the form proposed in Chapter 9.1. This requires in-place (also called *in situ*) modifications of `a`, meaning that we send in an array, modify it, and experience the modification in the calling code without getting anything returned from the function. This is the typical Fortran (and C) programming

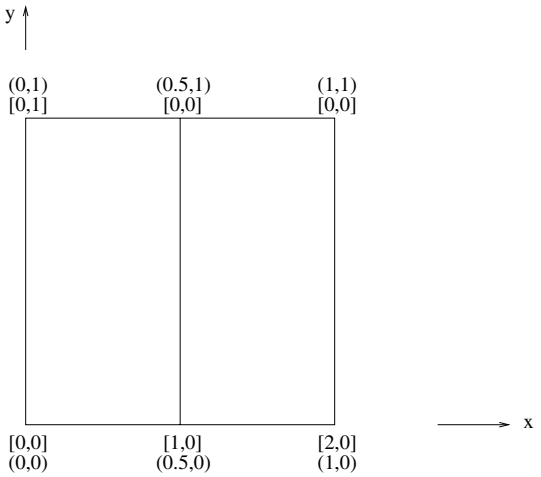

Fig. 9.2. Sketch of a 3×2 grid for testing the `ext_gridloop` module. `[,]` denotes indices in an array of scalar field values over the grid, and `(,)` denotes the corresponding (x, y) coordinates of the grid points.

style. We can do this in Python too, see Chapter 3.3.4. It is instructive to go through the details of how to achieve in-place modifications of arrays in Fortran routines because we then learn how to avoid unnecessary array copying in the F2PY-generated wrapper code. With large multi-dimensional arrays such copying can slow down the code significantly.

The `intent(inout)` specification of `a` is used for in-place modifications of an array:

```
      subroutine gridloop1_v2(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func1
Cf2py intent(inout) a
      external func1
      integer i,j
      real*8 x, y
      do j = 0, ny-1
         y = ycoor(j)
         do i = 0, nx-1
            x = xcoor(i)
            a(i,j) = func1(x, y)
         end do
      end do
      return
      end
```

F2PY now generates the interface:

```
gridloop1_v2 - Function signature:
  gridloop1_v2(a,xcoor,ycoor,func1,[nx,ny,func1_extra_args])
```

```
Required arguments:
  a : in/output rank-2 array('d') with bounds (nx,ny)
  xcoor : input rank-1 array('d') with bounds (nx)
  ycoor : input rank-1 array('d') with bounds (ny)
  func1 : call-back function
Optional arguments:
  nx := shape(a,0) input int
  ny := shape(a,1) input int
  func1_extra_args := () input tuple
```

Running

```
a = zeros((size(xcoor),size(ycoor)), Float)
ext_gridloop.gridloop1_v2(a, xcoor, ycoor, myfunc)
print 'a after gridloop1_v2:\n', a
```

results in an exception: "`intent(inout)` array must be contiguous and with a proper type and size". What happens?

For the `intent(inout)` to work properly in a Fortran function, the input array must have column major storage. Otherwise a copy is taken, and the output array is a different object than the input array, a fact that is incompatible with the `intent(inout)` requirement. Every F2PY generated extension module has a function `has_column_major_storage` returning true or false depending on whether the array has column major storage or not:

```
print 'Fortran storage of a?', \
      ext_gridloop.has_column_major_storage(a)
```

Issuing this statement before the call shows that `a` has not column major storage, which is expected since `a` is a NumPy array with row major storage computed by the `zeros` function. F2PY generated modules offer a function `as_column_major_storage` for turning an array into column major storage form. With this function we can perform the intended in-place change of `a`:

```
a = ext_gridloop.as_column_major_storage(
        zeros((size(xcoor), size(ycoor)), Float))
ext_gridloop.gridloop1_v2(a, xcoor, ycoor, myfunc)
```

We have made the final `gridloop1` function as a copy of the previously shown `gridloop1_v2` function. The call from Python can be sketched as follows:

```
class Grid2Deff(Grid2D):
    ...
    def ext_gridloop1(self, f):
        nx = size(self.xcoor);  ny = size(self.ycoor)
        a = zeros((nx,ny), Float)
        try:  # are we in Fortran?
            a = ext_gridloop.as_column_major_storage(a)
        except:
            pass
        ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, f)
        return a
```

Looking at this code, we realize that the `ext_gridloop1` function in Chapter 9.1 is too simple: for the Fortran module we need an adjustment for differences in storage schemes, i.e., `a` must have Fortran storage before we call `gridloop1`.

We emphasize that our final `gridloop1` function does not demonstrate the recommended usage of F2PY to interface a Fortran function. One should avoid the `intent(inout)` specification and instead use `intent(in,out)`, as we did in `gridloop3`, or one can use `intent(in,out,overwrite)`. There is more information on these important constructs in the next paragraph.

Allowing Overwrite. Recall the `gridloop3` function from page 451, which defines `a` as `intent(in,out)`. If we supply a NumPy array, the Fortran wrapper functions will by default return an array different from the input array in order to hide issues related to different storage in Fortran and C. On the other hand, if we send an array with column major storage to `gridloop3`, the function can work directly on this array. The following interactive session illustrates the point:

```
>>> a = zeros((size(xcoor),size(ycoor)), Float)
>>> ext_gridloop.has_column_major_storage(a)
0
>>> b = ext_gridloop.gridloop3(a, xcoor, ycoor, myfunc)
copied an array using copy_ND_array: size=6, elsize=8
>>> a is b
False   # b is a different array

>>> a = zeros((size(xcoor),size(ycoor)), Float)  # C storage
>>> a = ext_gridloop.as_column_major_storage(a)
copied an array using copy_ND_array: size=6, elsize=8
>>> ext_gridloop.has_column_major_storage(a)
1
>>> b = ext_gridloop.gridloop4(a, xcoor, ycoor, myfunc)
>>> a is b
True   # b is the same array as a; a is overwritten
```

With the `-DF2PY_REPORT_ON_ARRAY_COPY=1` flag, we can see exactly where the wrapper code makes a copy. This enables precise monitoring of the efficiency of the Fortran-Python coupling. The `intent` specification allows a keyword `overwrite`, as in `intent(in,out,overwrite) a`, to explicitly ask F2PY to overwrite the array if it has the right storage and element type. With the `overwrite` keyword an extra argument `overwrite_a` is included in the function interface. Its default value is 1, and the calling code can supply 0 or 1 to monitor whether `a` is to be overwritten or not.

More information about these issues are found in the F2PY manual.

Mixing C and Fortran Storage. One can ask the wrapper to work with an array with row major storage by specifying `c` in the `intent` statement: `intent(inout,c) a`. Doing this in a routine like `gridloop1` (it is done in `gridloop1_v3` in `gridloop.f`) gives wrong `a` values in Python. The erroneous result is not surprising as the Fortran function fills values in `a` as if it had col-

umn major storage, whereas we treat its storage as row major in the Python code. The remedy in this case would be to transpose `a` in the Fortran function after it is computed. This requires an extra scratch array and a trick utilizing the fact that we may declare the transpose with different dimensions in different subroutines. The interested reader might take a look at the `gridloop1_v4` function in `gridloop.f`. The corresponding Python call is found in the `gridloop1_session.py` script[1] in `src/py/mixed/Grid2D/F77`. Unfortunately, Fortran does not have dynamic memory so the scratch array is supplied from the Python code.

The bottom line of these discussions is that F2PY hides all problems with different array storage in Fortran and Python, but you need to specify input, output, and input/output variables – and check the signature of the generated interface.

Input Arrays and Repeated Calls to a Fortran Function. In this paragraph we outline a typical problem with hidden array copying. The topic is of particular importance when sending large arrays repeatedly to Fortran subroutines, see Chapter 12.3.6 for a real-world example involving numerical solution of partial differential equations. Here we illustrate the principal difficulties in a much simpler problem setting. Suppose we have a Fortran function `somefunc` with the signature

```
      subroutine somefunc(a, b, c, m, n)
      integer m, n
      real*8 a(m,n), b(m,n), c(m,n)
Cf2py intent(out) a
Cf2py intent(in) b
Cf2py intent(in) c
```

The Python code calling `somefunc` looks like

```
<create b and c>

for i in xrange(very_large_number):
    a = extmodule.somefunc(b, c)
    <do something with a>
```

The first problem with this solution is that the `a` array is created in the wrapper code in every pass of the loop. Changing the `a` array in the Fortran code to an `intent(in,out)` array opens up the possibility for reusing the same storage from call to call:

```
Cf2py intent(in,out) a
```

The calling Python code becomes

[1] This script actually contains a series of tests of the various `gridloop1_v*` subroutines.

```
<create a, b, and c>

for i in xrange(very_large_number):
    a = extmodule.somefunc(a, b, c)
    print 'address of a:', id(a)
    <do something with a>
```

The `id` function gives a unique identity of a variable. Tracking `id(a)` will show if `a` is the same array throughout the computations. The `print` statement prints the same address in each pass, except for the first time. Initially, `a` has row major storage and is copied by the wrapper code to an array with column major storage in the first pass. Thereafter the physical storage can be reused from call to call.

The storage issues related to the `a` array are also relevant to `b` and `c`. If we turn on the `F2PY_REPORT_ON_ARRAY_COPY` macro when running F2PY, we will see that two copies take place in every call to `somefunc`. The reason is that `b` and `c` have row major storage when calling `somefunc`, and the wrapper code converts these arrays to column major storage. Since neither `b` nor `c` is returned, we never get the column major storage version back in the Python code.

Because `somefunc` is called a large number of times, the extra copying of `b` and `c` may represent a significant decrease in computational efficiency. The remedy is to convert `b` and `c` to column major storage before the loop. My own rule of thumb is in fact to transform all arrays to column major storage in the Python code such that the F2PY generated wrapper codes does not need to make copies and transpose arrays.

```
<create a, b, and c>
a = extmodule.as_column_major_storage(a)
b = extmodule.as_column_major_storage(b)
c = extmodule.as_column_major_storage(c)

for i in range(very_large_number):
    a = extmodule.somefunc(a, b, c)
    <do something with a>
```

To summarize, (i) ensure that all multi-dimensional input arrays being sent many times to Fortran subroutines have column major storage, and (ii) let output arrays be declared with `intent(in,out)` such that storage is reused.

To be sure that storage really is reused in the Fortran routine, one can declare all arrays with `intent(in,out)` and store the returned references also of input arrays. Recording the `id` of each array before and after the Fortran call will then check if there is no unnecessary copying. Afterwards the `intent(in,out)` declaration of input arrays can be brought back to `intent(in)` to make the Python call statements easier to read. An alternative or additional strategy is to monitor the memory usage with the function `memusage` in the `py4cs.misc` module (a pure copy of the `memusage` function in SciPy's test suite).

Based on the previous discussion, the `gridloop1` and `gridloop2` subroutines should, at least if they are called a large number of times, be merged to one version where the `a` array is input and output argument:

```
a = ext_gridloop.gridloop_noalloc(a, self.xcoor, self.ycoor, func)
```

In the efficiency tests reported in Chapter 10.3.1, the Fortran subroutines are called many times, and we have therefore included this particular subroutine to measure the overhead of allocating `a` repeatedly in the wrapper code (`gridloop_noalloc` is the same subroutine as `gridloop2_str` in Chapter 9.4.2 except that `a` is declared as `intent(in,out)`).

9.3.4 F2PY Interface Files

In the previous examples we inserted, in the Fortran code, simple `Cf2py` comments containing F2PY directives like `intent` and `depend`. As an alternative, we could edit the F2PY-generated interface file with extension `.pyf`. This is preferable if we interface large software packages where direct editing of the source code may be lost in future software updates. Consider the `gridloop2` function without any `Cf2py` comments:

```
      subroutine gridloop2(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func1
      external func1
```

Suppose we store this version of `gridloop2` in a file `tmp.f`. We may run F2PY and make an interface file with the `-h` option:

```
f2py -m tmp -h tmp.pyf tmp.f
```

The main content of the interface file `tmp.pyf` is shown below:

```
python module gridloop2__user__routines
  interface gridloop2_user_interface
    function func1(x,y) ! in :tmp:tmp.f
      real*8 :: x
      real*8 :: y
      real*8 :: func1
    end function func1
  end interface gridloop2_user_interface
end python module gridloop2__user__routines

python module tmp ! in
  interface  ! in :tmp
    subroutine gridloop2(a,xcoor,ycoor,nx,ny,func1) ! in :tmp:tmp.f
      use gridloop2__user__routines
      real*8 dimension(nx,ny) :: a
      real*8 dimension(nx),depend(nx) :: xcoor
      real*8 dimension(ny),depend(ny) :: ycoor
      integer optional,check(shape(a,0)==nx),depend(a) :: nx=shape(a,0)
```

```
      integer optional,check(shape(a,1)==ny),depend(a) :: ny=shape(a,1)
      external func1
    end subroutine gridloop2
  end interface
end python module
```

Let us explain this file in detail. The interface file uses a combination of Fortran 90 syntax and F2PY-specific keywords for specifying the interface. F2PY assumes that `external` functions are callbacks to Python and guesses their signatures based on sample calls in the Fortran source code. Each function `f` having one or more `external` arguments gets a special interface `f__user__routines` defining the signature of the callback function. In the present example we see that F2PY has guessed that the `func1` argument is a callback function taking two `real*8` numbers as arguments and returning a `real*8` number.

The pair `python module tmp` and `end python module` encloses the list of functions to be wrapped. Each function is presented with its signature. When F2PY has no information about an argument, it assumes that the argument is input data. In the present case, all arguments are therefore treated as input data. The `dimension` statement declares an array of the indicated size. The line

```
      real*8 dimension(nx),depend(nx) :: xcoor
```

says that `xcoor` is an array of dimension `nx` and that `xcoor`'s size depends on `nx`. The line

```
      integer optional,check(shape(a,1)==ny),depend(a) :: ny=shape(a,1)
```

declares `ny` as an integer, which is optional and whose value depends on `a`. Furthermore, it should be checked that `ny` equals the length of the second dimension of `a`, `shape(a,1)`. We also notice the `use gridloop2__user__routines` statement, indicating that the signature of the callback function `func1` is defined in the `gridloop2__user__routines` module in the beginning of the interface file.

We need to edit the interface file to tell F2PY that `a` is an output argument of `gridloop2`. The `intent(out)` specification must be added to the declaration of `a`, `nx` and `ny` must depend on `xcoor` and `ycoor` (not `a`, which will not be supplied in the call), and the size of `a` must depend on `nx` and `ny`:

```
    subroutine gridloop2(a,xcoor,ycoor,nx,ny,func1)
      use gridloop2__user__routines
      real*8 dimension(nx,ny),intent(out),depend(nx,ny) :: a
      real*8 dimension(nx),intent(in) :: xcoor
      real*8 dimension(ny),intent(in) :: ycoor
      integer optional,check(len(xcoor)==nx),depend(xcoor) \
                                          :: nx=len(xcoor)
      integer optional,check(len(ycoor)==ny),depend(ycoor) \
                                          :: ny=len(ycoor)
      external func1
    end subroutine gridloop2
```

To get the right specification in the interface file, one can insert `Cf2py` comments in the code, run `f2py -h ...`, keep the interface file in a safe place, and thereafter remove the `Cf2py` comments again. I find this procedure simpler than editing the default interface file directly.

Remarks on Nested Callbacks. The version of F2PY available at the time of this writing cannot correctly determine the callback signature if the Fortran function receiving a callback argument passes this argument to another Fortran function. The following example illustrates the point:

```
      subroutine r1(x, y, n, f1)
      integer n
      real*8 x(n), y(n)
      external f1
      call f1(x, y, n)
      return
      end

      subroutine r2(x, y, n, f2)
      integer n
      real*8 x(n), y(n)
      external f2
      call r1(x, y, n, f2)
      return
      end
```

The `r2` routine has no call to `f2` and therefore F2PY cannot guess the signature of `f2`. In this case, we have to edit the interface file. Running

```
f2py -m tmp -h tmp.pyf somefile.f
```

yields an interface file `tmp.pyf` of the form

```
python module r1__user__routines
    interface r1_user_interface
        subroutine f1(x,y,n)
            real*8 dimension(n) :: x
            real*8 dimension(n),depend(n) :: y
            integer optional,check(len(x)>=n),depend(x) :: n=len(x)
        end subroutine f1
    end interface r1_user_interface
end python module r1__user__routines

python module r2__user__routines
    interface r2_user_interface
        external f2
    end interface r2_user_interface
end python module r2__user__routines

python module tmp ! in
    interface  ! in :tmp
        subroutine r1(x,y,n,f1) ! in :tmp:somefile.f
            use r1__user__routines
            real*8 dimension(n) :: x
            real*8 dimension(n),depend(n) :: y
```

```
            integer optional,check(len(x)>=n),depend(x) :: n=len(x)
            external f1
        end subroutine r1
        subroutine r2(x,y,n,f2) ! in :tmp:somefile.f
            use r2__user__routines
            real*8 dimension(n) :: x
            real*8 dimension(n),depend(n) :: y
            integer optional,check(len(x)>=n),depend(x) :: n=len(x)
            external f2
        end subroutine r2
    end interface
end python module tmp
```

The callback functions are specified in the `*__user__routines` modules. As we can see, the `r2__user__routines` module has no information about the signature of `f2`. We can either insert the right `f2` signature in this module, or we can edit the specification of the callback in the declaration of the `r2` routine. Following the latter idea, we replace

```
            use r2__user__routines
```

by

```
            use r1__user__routines, f2=>f1
```

This means that the callback subroutine (`f2`) in `r2` now applies the specification given in the `r1__user__routines` module, with the name `f1` replaced by `f2`.

Editing interface files is acceptable if the underlying Fortran library is static with respect to its function signatures. However, if you develop a Fortran library and frequently need new Python interfaces, the required interface file editing should be automated. In the present case, the following statements for building the extension module can be placed in a Bourne shell script:

```
f2py -m tmp -h tmp.pyf --overwrite-signature somefile.f
subst.py 'use r2__user__routines' \
         'use r1__user__routines, f2=>f1' tmp.pyf
f2py -c tmp.pyf somefile.f
```

The directory `src/misc/f2py-callback` contains such a script and the Fortran source code file for the present example.

A demonstration of the `tmp` module with the example on nested callbacks might read

```
>>> import tmp
>>> from py4cs.numpytools import *
>>> def myfunc(x, y):
        y += x

>>> p = zeros(2,Float) + 2.0;  q = p + 4
>>> p, q
(array([ 2.,  2.]), array([ 6.,  6.]))
>>> tmp.r1(p, q, myfunc)
>>> p, q
(array([ 2.,  2.]), array([ 8.,  8.]))
```

The important thing to note here is that the Python callback function `myfunc` must perform *in-place* modifications of its arguments if the modifications are to be experienced in the Fortran code. Fortran makes a straight `call` statement to `myfunc` and cannot make use of any return values from `myfunc`.

9.3.5 Hiding Work Arrays

Since Fortran prior to version 90 did not have support for dynamic memory allocation, there is a large amount of Fortran 77 code requiring the calling program to supply work arrays. Suppose we have a routine

```
subroutine myroutine(a, b, m, n, w1, w2)
integer m, n
real*8 a(m), b(n), w1(3*n), w2(m)
```

Here, `w1` and `w2` are work arrays. If `myroutine` were implemented in Python, its signature would be `myroutine(a, b)` since `m` and `n` can be extracted from the size of the `a` and `b` objects and since `w1` and `w2` can be allocated internally in the function when needed. The same signature for the F77 version of `myroutine` can be realized by making `m` and `n` optional, which is the default F2PY behavior, and telling F2PY that `w1` and `w2` are to be dynamically allocated in the wrapper code. The latter action is specified by

```
Cf2py intent(in,hide) w1
Cf2py intent(in,hide) w2
```

in the Fortran source or by

```
real*8 dimension(3*n),intent(in,hide),depend(n) :: w1
real*8 dimension(m),intent(in,hide),depend(m) :: w2
```

in the interface file. The `hide` instruction implies that F2PY hides the variable from the argument list. We could specify `m` and `n` with `hide` too, this would remove them from the argument list instead of making them optional.

If `myroutine` is called a large number of times, the overhead in dynamically allocating `w1` and `w2` in the wrapper function for every call may be significant. A better solution would be to allocate the arrays once (if `m` and `n` do not change throughout the call history) and reuse the storage for every call. F2PY has a keyword `cache` for specifying this behavior:

```
Cf2py intent(in,hide,cache) w1
```

In the interface file we can write

```
real*8 dimension(3*n),intent(in,hide,cache),depend(n) :: w1
```

There is a command-line argument `-DF2PY_REPORT_ATEXIT` which makes F2PY report the time spent in the the wrapper functions. This provides a good indicator whether it is necessary to supply a `cache` directive or not.

9.4 Increasing Callback Efficiency

As will be evident from the efficiency tests in Chapter 10.3.1, callbacks to Python are expensive. We shall present three techniques to deal with the low performance associated with point-wise callbacks in our `gridloop1` and `gridloop2` routines. Chapter 9.4.1 demonstrates the use of vectorized callbacks. An approach with optimal efficiency is of course to avoid calling Python functions at all and instead implement the function to be evaluated at each grid point in Fortran. How this can be done with some flexibility on the Python side is explained in Chapter 9.4.2. Another possibility is to turn string formulas in Python into compiled Fortran functions on the fly. Such "inline" compilation of Fortran code has many other applications, and Chapter 9.4.3 goes through the technicalities.

9.4.1 Callbacks to Vectorized Python Functions

One remedy for increasing the efficiency of callbacks to Python is to make vectorized callbacks. For our example with the `gridloop1` and `gridloop2` routines it means that the Python function supplied as the `func1` argument should work with NumPy arrays and evaluate the mathematical expression at all the grid points in a vectorized fashion. (This actually means that the `gridloop1` or `gridloop2` calls are totally unnecessary: if the mathematical expression is to be provided in Python, we should stay in Python for a vectorized computation and not call Fortran at all in this particular example. However, there are lots of occasions where it could be convenient to supply mathematical expressions to a large Fortran program. That program should then call Python to do vectorized calculations.)

There are some technical issues with callbacks to vectorized Python functions so we provide an example on this. In the Python script we call Fortran, but jump immediately back to Python for a vectorized implementation of the `func1` function. From the Python side the code looks as follows:

```
class Grid2Deff(Grid2D):
    ...
    def ext_gridloop_vec1(self, f):
        """As ext_gridloop2, but vectorized callback."""
        a = zeros((size(self.xcoor),size(self.ycoor)), Float)
        a = ext_gridloop.gridloop_vec1(a, self.xcoor, self.ycoor, f)
        return a

def myfunc(x, y):
    return sin(x*y) + 8*x

def myfuncf1(a, xcoor, ycoor, nx, ny):
    """Vectorized function to be called from extension module."""
    x = xcoor[:,NewAxis];  y = ycoor[NewAxis,:]
    a[:,:] = myfunc(x, y)  # in-place modification of a
```

```
g = Grid2Deff(dx=0.2, dy=0.1)
a = g.ext_gridloop_vec1(myfuncf1)
```

We feed in a kind of wrapper function `myfuncf1`, to be called from Fortran, to the grid object's `ext_gridloop_vec1` method. This method sends arrays and `myfuncf1` to the F77 routine `gridloop_vec1`. This routine calls up `myfuncf1` with the necessary information for doing a vectorized computation, i.e., all arrays as well as `nx` and `ny`. The latter variables are optional in `myfuncf1`, but the wrapper code calling `myfuncf1` needs to wrap a Fortran/C array into a NumPy object and therefore needs the dimensions explicitly in the call from Fortran. One could also think that the `a` array is an output argument of `gridloop_vec1` such that it could be omitted in the call, but this does not work – we have to treat it as an input and output argument.

The `gridloop_vec1` routine looks like this:

```
      subroutine gridloop_vec1(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
Cf2py intent(in,out) a
      external func1

      call func1(a, xcoor, ycoor, nx, ny)
      return
      end
```

There is an important feature of `myfuncf1` regarding the handling of the array `a`. This is a NumPy array that must be filled with values in-place. A simple call

```
a = myfunc(x, y)
```

just binds a new NumPy array, the returned result from `myfunc`, to `a`. No changes are then made to the original argument `a`. We therefore need to perform the in-place assignment

```
a[:,:] = myfunc(x, y)
```

F2PY allows us to supply extra arguments to the callback function. Providing a reference to the grid object as extra argument enables use of the ready-made `xcoorv` and `ycoorv` arrays in the grid object. This saves two array copying operations in `myfuncf1`. Let us make the details clear by showing the exact code. We call Fortran by just passing the `a` array and the `func1` callback function, as there is now no need for the `xcoor` and `ycoor` arrays in the Fortran routine when we have this information in the grid object:

```
      subroutine gridloop_vec2(a, nx, ny, func1)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1)
Cf2py intent(in,out) a
      external func1
```

```
        call func1(a, nx, ny)
        return
        end
```

F2PY generates the signature

```
a = gridloop_vec2(a, func1, nx=shape(a,0), ny=shape(a,1),
                  func1_extra_args=())
```

for this function. The argument `func1_extra_args` can be used to supply a tuple of Python data structures that are augmented to the argument list in the callback function. In our case we may equip class `Grid2Deff` with the method

```
def ext_gridloop_vec2(self, f):
    """As ext_gridloop_vec1, but callback to func. w/grid arg."""
    a = zeros((size(self.xcoor),size(self.ycoor)), Float)
    a = ext_gridloop.gridloop_vec2(a, f, func1_extra_args=(self,))
    return a
```

The grid object itself is supplied as extra argument, which means that the wrapper code makes a callback to a Python function taking `a` and a grid object as arguments:

```
def myfuncf2(a, g):
    """Vectorized function to be called from extension module."""
    a[:,:] = myfunc(g.xcoorv, g.ycoorv)

g = Grid2Deff(dx=0.2, dy=0.1)
a = g.ext_gridloop_vec2(myfuncf2)
```

We can also make a callback to a method in class `Grid2Deff`:

```
class Grid2Deff(Grid2D):
    ...
    def myfuncf3(self, a):
        a[:,:] = myfunc(self.xcoorv, self.ycoorv)

    def ext_gridloop_vec3(self, f):
        """As ext_gridloop_vec2, but callback to class method."""
        a = zeros((size(self.xcoor),size(self.ycoor)), Float)
        a = ext_gridloop.gridloop_vec2(a, f)
        return a

g = Grid2Deff(dx=0.2, dy=0.1)
a = g.ext_gridloop_vec3(g.myfuncf3)
```

This solution avoids sending the grid object as an extra argument since the function object `g.myfuncf3` is invoked with `self` as the grid `g`.

As we have seen, calling Python functions with arrays as arguments requires a few more details to be resolved compared to callback functions with scalar arguments. Nevertheless, callback functions with array arguments are often a necessity from a performance point of view.

9.4.2 Avoiding Callbacks to Python

Instead of providing a function as argument to the `gridloop2` routine we could send a string specifying a *Fortran* function to call for each grid point. In the Fortran routine we test on the string and make the appropriate calls. The simplest way of implementing this idea is to create a small wrapper for the `gridloop2` subroutine:

```
      subroutine gridloop2_str(a, xcoor, ycoor, nx, ny, func_str)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
      character*(*) func_str
Cf2py intent(out) a
Cf2py depend(nx,ny) a
      real*8 myfunc, f2
      external myfunc, f2

      if (func_str .eq. 'myfunc') then
         call gridloop2(a, xcoor, ycoor, nx, ny, myfunc)
      else if (func_str .eq. 'f2') then
         call gridloop2(a, xcoor, ycoor, nx, ny, f2)
      end if
      return
      end
```

Here, `myfunc` and `f2` are F77 functions. F2PY handles the mapping between Python strings and Fortran character arrays transparently so we can call the `gridloop2_str` function with plain Python strings:

```
class Grid2Deff(Grid2D):
    ...
    def ext_gridloop_str(self, f77_name):
        a = ext_gridloop.gridloop2_str(self.xcoor, self.ycoor,
                                       f77_name)
        return a

g = Grid2Deff(dx=0.2, dy=0.1)
a = g.ext_gridloop_str('f2')
a = g.ext_gridloop_str('myfunc')
```

This approach is typically 30-40 times faster than using point-wise Python callbacks in the test problem treated in Chapter 10.3.1.

9.4.3 Compiled Inline Callback Functions

A way of avoiding expensive callbacks to Python is to let the steering script compile the mathematical expression into F77 code and then direct the callback to the compiled F77 function. F2PY offers a module `f2py2e` with functions for building extension modules out of Python strings containing Fortran code. This allows us to migrate time-critical code to Fortran on the fly!

To create a Fortran callback function, we need to have a Python string expression for the mathematical formula. This can be a plain string or we may represent a function as a `StringFunction` instance from Chapter 12.2.1. Notice that the syntax of the string expression now needs to be compatible with Fortran. Mathematical expressions like `sin(x)*exp(-y)` have the same syntax in Python and Fortran, but Python-specific constructs like `math.sin` and `math.exp` will of course not compile. Letting the Python variable `fstr` hold the string expression, we embed the expression in an F77 function `fcb` (= Fortran callback):

```
source = """
      real*8 function fcb(x, y)
      real*8 x, y
      fcb = %s
      return
      end
""" % fstr
```

If we instead of a plain Python string `fstr` apply a `StringFunction` instance `f`, we may extract the string formula by `str(f)`. However, `StringFunction` instances also offer a method `F77_code`, which dumps out a Fortran function. We could then just write

```
source = f.F77_code('fcb')
```

We want the F77 `gridloop2` routine to call `fcb` so we make a wrapper routine to be called from Python:

```
      subroutine gridloop2_fcb(a, xcoor, ycoor, nx, ny)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
Cf2py intent(out) a
Cf2py depend(nx,ny) a
      real*8 fcb
      external fcb

      call gridloop2(a, xcoor, ycoor, nx, ny, fcb)
      return
      end
```

The Python script must compile these Fortran routines and build an extension module, here named `callback`. The `callback` shared library calls `gridloop2` so it must be linked with the `ext_gridloop.so` library. From Python we call `callback.gridloop2_fcb`.

Building an extension module out of some Fortran source code is done by

```
import f2py2e
f2py_args = "--fcompiler='Gnu' --build-dir tmp2 etc..."
r = f2py2e.compile(source, modulename='callback',
                   extra_args=f2py_args, verbose=True,
                   source_fn='sourcecodefile.f')
if r:
    print 'unsuccessful compilation'; sys.exit(1)
import callback
```

The `compile` function builds a standard `f2py` command and runs it under `os.system`.

It might be attractive to make two separate functions, one for building the `callback` extension module and one for calling the `gridloop2_fcb` function. The inline definition of the appropriate Fortran code and the compile/build process may be implemented as done below.

```
        def ext_gridloop2_fcb_compile(self, fstr):
            if not isinstance(fstr, str):
                raise TypeError, \
                'fstr must be string expression, not', type(fstr)

            # generate Fortran source
            import f2py2e
            source = """
          real*8 function fcb(x, y)
          real*8 x, y
          fcb = %s
          return
          end

          subroutine gridloop2_fcb(a, xcoor, ycoor, nx, ny)
          integer nx, ny
          real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
    Cf2py intent(out) a
    Cf2py depend(nx,ny) a
          real*8 fcb
          external fcb

          call gridloop2(a, xcoor, ycoor, nx, ny, fcb)
          return
          end
    """ % fstr
            # compile code and link with ext_gridloop.so:
            f2py_args = "--fcompiler='Gnu' --build-dir tmp2"\
                        " -DF2PY_REPORT_ON_ARRAY_COPY=1 "\
                        " ./ext_gridloop.so"
            r = f2py2e.compile(source, modulename='callback',
                               extra_args=f2py_args, verbose=True,
                               source_fn='_cb.f')
            if r:
                print 'unsuccessful compilation'; sys.exit(1)
            import callback  # can we import successfully?
```

The `f2py2e.compile` function stores in this case the `source` code in a file with name `_cb.f` and runs an `f2py` command. If something goes wrong, we have the `_cb.f` file together with the generated wrapper code and F2PY interface file in the `tmp2` subdirectory for human inspection and manual building, if necessary.

The array computation method in class `Grid2Deff`, utilizing the new extension module `callback`, may take the form

```
        def ext_gridloop2_fcb(self):
            """As ext_gridloop2, but compiled Fortran callback."""
```

```
        import callback
        a = callback.gridloop2_fcb(self.xcoor, self.ycoor)
        return a
```

Our original string with a mathematical expression is now called as a Fortran function inside the loop in `gridloop2`.

With the `f2py2e` module we can more or less "inline" Fortran code when desired. For example, a method in class `Grid2Deff` could create the Fortran `gridloop2` subroutine at run time, with the mathematical formula to be evaluated hardcoded into the loop:

```
    def ext_gridloop2_compile(self, fstr):
        if not isinstance(fstr, str):
            raise TypeError, \
            'fstr must be string expression, not', type(fstr)

        # generate Fortran source for gridloop2:
        import f2py2e
        source = """
      subroutine gridloop2(a, xcoor, ycoor, nx, ny)
      integer nx, ny
      real*8 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1)
Cf2py intent(out) a
Cf2py depend(nx,ny) a

      integer i,j
      real*8 x, y
      do j = 0, ny-1
         y = ycoor(j)
         do i = 0, nx-1
            x = xcoor(i)
            a(i,j) = %s
         end do
      end do
      return
      end
""" % fstr
        f2py_args = "--fcompiler='Gnu' --build-dir tmp1"\
                    " -DF2PY_REPORT_ON_ARRAY_COPY=1"
        r = f2py2e.compile(source, modulename='ext_gridloop2',
                           extra_args=f2py_args, verbose=True,
                           source_fn='_cb.f')
```

Now there is no call to any external function, and the Python call to `gridloop2` does not supply any callback information:

```
    def ext_gridloop2_v2(self):
        import ext_gridloop2
        return ext_gridloop2.gridloop2(self.xcoor, self.ycoor)
```

Tailoring Fortran routines as shown here is easy to do at run time in a Python script. The great advantage is that we have more user-provided information available than when pre-compiling an extension module. The disadvantage is that a script with a build process is more easily broken.

9.5 Summary

Let us summarize how to work with F2PY:

1. Classify all arguments with the `intent` keyword, either with the aid of `Cf2py` comments in the Fortran source code or by editing the interface file. Common `intent` specifications are provided in Table 9.1.

Table 9.1. List of some important `intent` specifications in F2PY interface files or in `Cf2py` comments in the Fortran source.

`intent(in)`	input variable
`intent(out)`	output variable
`intent(in,out)`	input *and* output variable
`intent(in,hide)`	hide (e.g. work arrays) from argument list
`intent(in,hide,cache)`	keep hidden allocated arrays in memory
`intent(in,out,overwrite)`	enable an array to be overwritten (if feasible)
`depend(m,n) q`	make `q`'s dimensions depend on `m` and `n`

2. Run F2PY. A typical command (not involving interface files explicitly) is

```
f2py -m modulename -c --fcompiler 'Gnu' --build-dir tmp1 \
   file1.f file2.f  only: routine1 routine2 routine3 :
```

   An interface to three subroutines in two files are built with this command.

3. Import the module in Python and print the doc strings of the module and each of its functions, e.g.,

```
import modulename
# print summary of all functions and data:
print modulename.__doc__
# print detailed info about each item in the module:
for i in dir(modulename):
    print '===', eval('modulename.' + i + '.__doc__')
```

9.6 Exercises

Exercise 9.1. Extend Exercise 5.1 with a callback to Python.

Modify the solution of Exercise 5.1 such that the function to be integrated is implemented in Python (i.e., perform a callback to Python) and transferred to the Fortran code as a subroutine or function argument. Test different types of callable Python objects: a plain function, a lambda function, and a class instance with a `__call__` method. ◇

Exercise 9.2. Compile callback functions in Exercise 9.1.

The script from Exercise 9.1 calls a Python function for every point evaluation of the integrand. Such callbacks to Python are known to be expensive. As an alternative we can use the technique from Chapter 9.4.3: the integrand is specified as a mathematical formula stored in a string, the string is turned into a Fortran function, and this function is called from the Fortran function performing the numerical integration. From the Python side, we make a function like

```
def Trapezoidal(expression, a, b, n):
    """Integrate expression (string) from a to b in n steps."""
    <turn expression into an F77 callback function>
    <make wrapper for the F77 integration function>
    <build extension module via f2py2e.compile>
    <compute integral and return result>

# usage:
expr = '1 + 2*x';  a = 0;  b = 1
for k in range(1, 10):
    n = 2**k
    print 'integrate %s from %g to %g with n=%d: %g' % \
          (expr, a, b, n, Trapezoidal(expr, a, b, n))
```

Implement this `Trapezoidal` function. Perform timings to compare the efficiency of the solutions in Exercises 5.1, 9.1, and 9.2. ◇

Exercise 9.3. Smoothing of time series.

Assume that we have a noisy time series $y_0, y_1, y_2, \ldots, y_n$, where y_i is a signal $y(t)$ evaluated at time $t = i\Delta t$. The $y_0, y_1, y_2, \ldots, y_n$ data are stored in a NumPy array `y`. The time series can be smoothed by a simple scheme like

$$\bar{y}_i = \frac{1}{2}(y_{i-1} + y_{i-1}), \quad i = 1, \ldots, n-1\,.$$

Implement this scheme in three ways:

1. a Python function with a plain loop over the array `y`,
2. a Python function with a vectorized expression of the scheme (see Chapter 4.2.2),
3. a Fortran function with a plain loop over the array `y`.

Write a main program in Python calling up these three alternative implementations to perform m smoothing operations. Compare timings (use, e.g., the `timer` function in `py4cs.misc`, described in Chapter 8.10.1) for a model problem generated by

```
from py4cs.numpytools import *
def noisy_data(n):
    T = 40              # time interval (0,T)
    dt = T/float(n)     # time step
    t = sequence(0,T,dt)
    y = sin(t) + RandomArray.normal(0,0.1,size(t))
    return y, t
```

How large must n and m be before you regard the plain Python loop as too slow? (This exercise probably shows that plain Python loops over quite large one-dimensional arrays run sufficiently fast so there is little need for vectorization or migration to compiled languages unless the loops are repeated a large number of times. Nevertheless, the algorithms execute much faster with NumPy or in a compiled language. My NumPy and F77 implementations ran 8 and 30 times faster than pure Python with plain loops on my laptop.) ◇

Exercise 9.4. Smoothing of 3D data.

This is an extension of Exercise 9.3. Assume that we have noisy 3D data $w_{i,j,k}$ of a function $w(x,y,z)$ at uniform points with indices (i,j,k) in a 3D unit cube, $i,j,k = 0,\ldots,n$. An extension of the smoothing scheme in Exercise 9.3 to three dimensions reads

$$\bar{w}_{i,j,k} = \frac{1}{6}(w_{i-1,j,k} + w_{i+1,j,k} + w_{i,j-1,k} + w_{i,j+1,k} + w_{i,j,k-1} + w_{i,j,k+1})\,.$$

Implement this scheme in three ways: plain Python loop, vectorized expression, and code migrated to Fortran. Compare timings for a model problem generated by

```
from py4cs.numpytools import *
def noisy_data(n):
    q = n+1  # n+1 data points in each direction
    w = RandomArray.normal(0,0.1,q**3)
    w.shape = (q,q,q)
    return w
```

This exercise demonstrates that processing of 3D data is very slow in plain Python and migrating code to a compiled language is usually demanded. ◇

Exercise 9.5. Type incompatibility between Python and Fortran.

Suppose you implement the `gridloop1` F77 function from Chapter 9.3.3 as

```
      subroutine gridloop1(a, xcoor, ycoor, nx, ny, func1)
      integer nx, ny
      real*4 a(0:nx-1,0:ny-1), xcoor(0:nx-1), ycoor(0:ny-1), func1
Cf2py intent(inout) a
Cf2py intent(in) xcoor
Cf2py intent(in) ycoor
Cf2py depend(nx, ny) a
      ...
```

That is, the array elements are now `real*4` (single precision) floating-point numbers. Demonstrate that if `a` is created with `Float` elements in the Python code, changes in `a` are not visible in Python (because F2PY takes a copy of `a` when the element type in Python and Fortran differs).

Use the `-DF2PY_REPORT_ON_ARRAY_COPY=1` flag when creating the module and monitor the extra copying. What is the remedy to avoid copying and get the function to work? ◇

Exercise 9.6. Problematic callbacks to Python from Fortran.

In this exercise we shall work with a Scientific Hello World example of the type encountered in Chapter 5.2.1, but now a Fortran routine makes a callback to Python:

```
      subroutine hello(hw, r1, r2)
      external hw
      real*8 r1, r2, s
C     compute s=r1+r2 in hw:
      call hw(r1, r2, s)
      write(*,*) 'Hello, World! sin(',r1,'+',r2,')=',s
      return
      end
```

Make an extension module `tmp` containing the `hello` routine and try it out with the following script:

```
def hw3(r1, r2, s):
    import math
    s = math.sin(r1 + r2)
    return s

import tmp
tmp.hello(hw3, -1, 0)
```

Explain why the value of `s` in the `hello` routine is wrong after the `hw` call.

Change `s` to an array in the Fortran routine as a remedy for achieving a correct output value of `r1+r2`. ◇

Exercise 9.7. Array look-up efficiency: Python vs. Fortran.

Consider filling a NumPy array `a` with values,

```
for i in xrange(n):
    for j in xrange(n):
        a[i, j] = i*j-2
```

Is there anything to be gained by merging the array assignment line to Fortran? That is, the Python code looks like

```
for i in xrange(n):
    for j in xrange(n):
        a = set(a, i, j, i*j-2)
```

where `set` is a Fortran subroutine. Perform this efficiency investigation, both with a simple arithmetic expression such as `i*j-2` as value to assign, and with an expression that is more costly, e.g., `sin(x)*sin(y)*exp(-x*y)` with `x=i*0.1` and `y=j*0.1`. Also test if there are differences in performance of indexing `Numeric` versus `numarray` objects. You will experience that NumPy indexing is faster in Fortran than in Python. ◇