Chapter 10

C and C++ Programming with NumPy Arrays

Our purpose with this chapter is to implement the `gridloop1` and `gridloop2` functions from Chapter 9 in C and C++. The goal is the same: we want to increase the computational efficiency by moving loops from Python to compiled code, but now we use C and C++ instead of Fortran. Before proceeding the reader should be familiar with the `gridloop1` and `gridloop2` functions and the calling Python code as defined in Chapter 9.1. It is not necessary to have digested the rest of Chapter 9 about various aspects of the corresponding Fortran implementation.

The most obvious way to write the `gridloop` function in C is to use a function pointer for the callback function, a double pointer for the `a` array, and single pointers for the `xcoor` and `ycoor` arrays:

```
typedef double (*Fxy)(double x, double y);  /* function ptr Fxy */

void gridloop(double **a, double *xcoor, double *ycoor,
              int nx, int ny, Fxy func1)
{
  int i, j;
  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
       a[i][j] = func1(xcoor[i], ycoor[j]);
    }
  }
}
```

This function is not straightforward to interface from Python. First, a NumPy array has a single pointer to its data segment. A double pointer `double **a` contains additional information (pointers to all the rows of a two-dimensional array). Second, a tool like SWIG cannot automatically handle the mapping between NumPy arrays and plain C or C++ arrays, and therefore the `gridloop` function in C cannot be wrapped without some manual work. The cause of this problem is that the C syntax does not couple the integers `nx` and `ny` to the dimensions of the arrays `a`, `xcoor`, and `ycoor` (as Fortran does, which F2PY takes advantage of).

In this chapter we shall apply several approaches to wrapping C functions with NumPy array arguments. First, we simply apply F2PY from Chapter 9 to wrap a C function in Chapter 10.1.1. Instant is another tool, treated in Chapter 10.1.2, where the C function is inlined as a string in the Python code.

Weave is similar to Instant, but with Weave only the loop itself needs to be written in C++ and stored as a string in the Python code, as we demonstrate in Chapter 10.1.3.

The rest of the chapter is focused on how to write all of the code in an extension module by hand. A pure C extension module is developed in Chapter 10.2, while Chapter 10.3 applies C++ and wraps NumPy arrays in C++ class objects. How to write a wrapper for the `gridloop` function above, with a double pointer `double **a` representation of the two-dimensional array, is treated in Chapter 10.2.11. A similar function in C++, utilizing a C++ array class instead of a low-level plain C array as the `a` argument, is explained in Chapter 10.3.3. Alternative tools like SWIG, `ctypes`, or Pyrex are not covered here, but the NumPy manual has information on how to transfer arrays with these tools.

Finally, in Chapter 10.4 we compare the Fortran, C, and C++ implementations of the `gridloop1` and `gridloop2` functions with respect to computational efficiency, safety in use, and programming convenience.

10.1 Automatic Interfacing of C/C++ Code

If we write the `gridloop2` function with `a` as a single pointer, it is possible to use tools to automatically wrap the C function. The relevant version of `gridloop2` takes the form

```
typedef double (*Fxy)(double x, double y);  /* function ptr Fxy */

#define index(a, i, j) a[j*ny + i]

void gridloop2(double *a, double *xcoor, double *ycoor,
               int nx, int ny, Fxy func1)
{
  int i, j;
  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
       index(a, i, j) = func1(xcoor[i], ycoor[j]);
    }
  }
}
```

Chapter 10.1.1 applies F2PY to automatically wrap this code with minor additional manual work. An alternative tool, Instant, can do much of the same, as we exemplify in Chapter 10.1.2. Weave is a third tool that is similar to Instant and briefly treated in Chapter 8.10.4. In Chapter 10.1.3 we use Weave to migrate the loops above to C++.

10.1.1 Using F2PY

In Chapter 5.2.2 we show that F2PY can also be used to wrap C functions. The present example with the `gridloop1` function above is more involved because it contains a callback function (`func1`) as well as multi-dimensional arrays.

First, we need to create an F2PY interface file for the `gridloop2` function in C and the callback function (`func1`). One simple way to do this is to write the `gridloop2` signature function in Fortran 77 and add `Cf2py` comments. All C arguments that are passed by value must be marked with `intent(c)` (since Fortran applies pointers for all arguments). In addition, the function name itself must be marked with `intent(c)`. The array `a` is an output array with C storage and must be marked with `intent(out, c)`. Finally, we need to indicate how the callback function is used as F2PY derives the callback function's signature from how the function is called. The arguments used in the call and the return value from `func1` are straight `double` variables, transferred by value, so we need associated `intent(c)` specifications. Besides the sample call of `func1`, there is no need to fill the Fortran version of the `gridloop2` function with any sensible statements. The complete Fortran specification of the `gridloop2` function in C then becomes

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

C sample call of callback function:
      real*8 x, y, r
      real*8 func1
Cf2py intent(c) x, y, r, func1
      r = func1(x, y)
      end
```

Running F2PY on this file,

```
f2py -m ext_gridloop -h ext_gridloop.pyf \
     --overwrite-signature signatures.f
```

results in an `ext_gridloop.pyf` file that you can examine in the directory `src/py/mixed/Grid2D/C/f2py`. An alternative is to write the interface file by hand.

The next step is to compile `gridloop.c` with the `gridloop2` function in C using F2PY and the interface file:

```
f2py -c --fcompiler=Gnu --build-dir tmp1 \
     -DF2PY_REPORT_ON_ARRAY_COPY=1 ext_gridloop.pyf gridloop.c
```

We have now a module that can be tested:

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

The output becomes

```
This module 'ext_gridloop' is auto-generated with f2py (version:2_3515).
Functions:
  a = gridloop2(xcoor,ycoor,func1,
      nx=len(xcoor),ny=len(ycoor),func1_extra_args=())
```

showing that we get access to a `gridloop2` in Python with exactly the same behavior as the one we generated in Fortran.

10.1.2 Using Instant

Instant allows inlining C or C++ functions in strings in Python scripts. The functions are automatically compiled and interfaced with SWIG to form an extension module. Hence, to use Instant you need to have SWIG installed.

The use of Instant is very simple. We write a C or C++ function for processing array data and store the code in a Python string `source`. Thereafter we call `instant.inline_with_numpy` with `source` as argument, together with an argument describing the relation between array pointers and integers holding the array dimensions in the C or C++ function. At the time of this writing, C/C++ functions wrapped by Instant cannot return arrays to Python so we must make a `gridloop1` type of function.

In the `Grid2Deff` class we can add a method that creates access to a `gridloop1` function in C using Instant. Since we write the C code in the Python program it is natural to avoid callback to a Python function and instead either call a C function or insert the function expression directly in the loop. The latter approach is the most efficient and used in this example:

```
    def ext_gridloop1_instant(self, fstr):
        if not isinstance(fstr, str):
            raise TypeError, \
            'fstr must be string expression, not %s', type(fstr)

        # generate C source (fstr must be valid C code):
        source = """
void gridloop1(double *a, int nx, int ny,
               double *xcoor, double *ycoor)
{
# define index(a, i, j)  a[i*ny + j]
  int i, j;  double x, y;
  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
      x = xcoor[i];  y = ycoor[i];
      index(a, i, j) = %s
    }
  }
}
```

```
""" % fstr

        try:
            from instant import inline_with_numpy
            a = zeros((self.nx, self.ny))
            arrays = [['nx', 'ny', 'a'],
                      ['nx', 'xcoor'],
                      ['ny', 'ycoor']]
            self.gridloop1_instant = \
                 inline_with_numpy(source, arrays=arrays)
        except:
            self.gridloop1_instant = None
```

The `arrays` list has one element for each array argument in the C function. An element in `arrays` is a list of the names of the variables holding the dimensions of an array, followed by the name of the array variable. For example, `['nx', 'ny', 'a']` means that `a` in the C code argument list is an array with first dimension `nx` and second dimension `ny`.

If `g` is a `Grid2Deff` instance, we call `g.ext_gridloop1_instant(fstr)` to make a C function and interface it with Instant. Then we call

```
a = zeros((g.nx, g.ny))
g.gridloop1_instant(a, g.nx, g.ny, g.xcoor, g.ycoor)
```

to call the C function to compute the `a` array.

In the case where we want a separate callback function in C to be called inside the loop, we simply create two functions in the `source` string. The use of Instant is now a bit different as we must use the `instant.create_extension` function, which returns a module, not a function, to Python.

10.1.3 Using Weave

Weave is a tool for inlining C++ snippets in Python programs. A quick demonstration of Weave appears in Chapter 8.10.4. You should be familiar with that material before proceeding here.

Using Weave in our example is easy: we just write the loops in C++, typically

```
for (i=0; i<nx; i++) {
  for (j=0; j<ny; j++) {
    a(i,j) = cppcb(xcoor(i), ycoor(j));
  }
}
```

where `cppcb` is the callback function implemented in C++, e.g.,

```
double cppcb(double x, double y) {
  return sin(x*y) + 8*x;
}
```

Alternatively, we can avoid the `cppcb` function and insert the mathematical expression directly in the loop (as we do in the previous section). However, here we exemplify the use of a separate `cppcb` function:

```
class Grid2Deff:
    ...
    def ext_gridloop2_weave(self, fstr):
        from scipy import weave
        # the callback function is now coded in C++
        # (fstr must be valid C++ code):
        extra_code = r"""
double cppcb(double x, double y) {
  return %s;
}
""" % fstr
        # the loop in C++ (with Blitz++ array syntax):
        code = r"""
int i,j;
for (i=0; i<nx; i++) {
  for (j=0; j<ny; j++) {
    a(i,j) = cppcb(xcoor(i), ycoor(j));
  }
}
"""
        nx = self.nx;  ny = self.ny
        xcoor = self.xcoor;  ycoor = self.ycoor
        a = zeros((nx,ny))
        err = weave.inline(code,
              ['a', 'nx', 'ny', 'xcoor', 'ycoor'],
              type_converters=weave.converters.blitz,
              support_code=extra_code, compiler='gcc')
        return a
```

If `g` is a `Grid2Deff` instance, we can now compute `a` by

```
a = g.ext_gridloop2_weave(fstr)
```

10.2 C Programming with NumPy Arrays

NumPy arrays can be created and manipulated from C. Our `gridloop1` function will work with this NumPy C API directly. This means that we need to look at how NumPy arrays are represented in C and what functions we have for working with these arrays from C. This requires us to have some basic knowledge of how to program Python from C. We shall jump directly to our grid loop example here, and explain it in detail, but it might be a good idea to take a "break" and scan the chapter "Extending and Embedding the Python Interpreter" [36] in the official Python documentation, or better, read the corresponding chapter in "Python in a Nutshell" [22] or in Beazley [2], before going in depth with the next sections.

10.2.1 The Basics of the NumPy C API

A C struct `PyArrayObject` represents NumPy arrays in C. The most important attributes of this struct are listed below.

- `int nd`
  The number of indices (dimensions) in the NumPy array.
- `npy_intp *dimensions`
  Array of length `nd`, where `dimensions[0]` is the number of entries in the first index (dimension) of the NumPy array, `dimensions[1]` is the number of entries in the second index (dimension), and so on. The `npy_intp*` type is the platform-independent counterpart to `int*` which is prepared for the increased address space of 64-bit machines.
- `char *data`
  Pointer to the first data element of the NumPy array.
- `npy_intp *strides`
  Array of length `nd` describing the number of bytes between two successive data elements for a fixed index. Suppose we have a two-dimensional `PyArrayObject` array `a` with `m` entries in the first index and `n` entries in the second one. Then `nd` is 2, `dimensions[0]` is `m`, `dimensions[1]` is `n`, and entry `(i,j)` is accessed by

  ```
  a->data + i*a->strides[0] + j*a->strides[1]
  ```

  in C or C++.
- `int descr->type_num`
  The type of entries in the array. The value should be compared to predefined constants: `NPY_DOUBLE` for the Python `float` type (`double` in C) and `NPY_INT` for the Python `int` type (`int` in C). We refer to the NumPy manual for for the constants corresponding to other data types.

The NumPy author recommends using convenience macros for accessing the attributes listed above. If `a` is a `PyArrayObject` pointer, we have

- `PyArray_NDIM(a)` for `a->nd`
- `PyArray_DIMS(a)` for `a->dimensions`
- `PyArray_DIM(a, i)` for `a->dimensions[i]`
- `PyArray_STRIDES(a)` for `a->strides`
- `PyArray_STRIDE(a, i)` for `a->strides[i]`
- `PyArray_TYPE(a)` for `a->descr->type_num`
- `PyArray_DATA(a)` for `(void *) (a->data)`
- `PyArray_GETPTR1(a, i)` for `(void *) a->data + i*a->strides[0]`
- `PyArray_GETPTR2(a, i, j)` for
  `(void *) a->data + i*a->strides[0] + j*a->strides[1]`

– Similar macros, `PyArray_GETPTR3` and `PyArray_GETPTR4`, exist for three- and four-dimensional arrays

Creating a new NumPy array in C code can be done by the function

```
PyObject * PyArray_SimpleNew(int nd,
                             npy_intp dimensions[nd],
                             int type_num);
```

The first argument is the number of dimensions, the next argument is a vector containing the length of each dimension, and the final argument is the entry type (`NPY_DOUBLE`, `NPY_INT`, etc.). To create a 10×21 array of `double`s we write

```
PyArrayObject *a;  npy_intp dims[2];
dims[0] = 10;  dims[1] = 21;
a = (PyArrayObject *) PyArray_SimpleNew(2, dims, NPY_DOUBLE);
```

The elements of `a` are now uninitialized. There is an alternative function `PyArray_ZEROS` which creates a new array and sets the elements to zero (like `numpy.zero`).

Sometimes one already has a memory segment in C holding an array (stored row by row) and wants to wrap the data in a `PyArrayObject` structure. The following function is available for this purpose:

```
PyObject * PyArray_SimpleNewFromData(int nd,
           npy_intp dimensions[nd],
           int type_num,
           void *data);
```

The first three arguments are as explained for the former function, while `data` is a pointer to the memory segment where the entries are stored. As an example of application, imagine that we have a 10×21 array with double-precision real numbers, stored row by row in a plain C vector `vec`. We can wrap the data in a NumPy array `a` by

```
PyArrayObject *a;  npy_intp dims[2];
dims[0] = 10;  dims[1] = 21;
a = (PyArrayObject *) PyArray_SimpleNewFromData(2, dims,
    NPY_DOUBLE, (void *) vec);
```

The programmer is responsible for not freeing the `vec` data before `a` is destroyed. If `a` is returned to Python, it is difficult to predict the lifetime of `a`, so one must be very careful with freeing `vec`.

Sometimes we have a two-dimensional C array available through a double pointer `double **v` and want to wrap this array in a NumPy structure. We then need to supply the address of the first array element, `&v[0][0]`, as the `data` pointer in the `PyArray_SimpleNew` call, *provided all array elements are stored in a contiguous memory segment.* If not, say the rows are allocated separately and scattered throughout memory, the NumPy structure must be created by calling `PyArray_SimpleNew` and copying data element by element.

The NumPy C API also contains a function for turning an arbitrary Python sequence into a NumPy array with contiguous storage:

```
PyObject * PyArray_FROM_OTF(PyObject *object,
                            int type_num,
                            int requirements)
```

The sequence is stored in `object`, the desired item type in the returned NumPy array is specified by `type_num` (e.g., `NPY_DOUBLE`), while the last arguments is typically `NPY_IN_ARRAY` if `object` is pure input or `NPY_INOUT_ARRAY` if `object` is both an input and output array. The dimensions of the resulting array are determined from the input sequence (`object`). If `object` is already a NumPy array with the right element type, the function simply returns `object`, i.e., there is no performance loss when a conversion is not required. A typical application is to use `PyArray_FROM_OTF` to ensure that an argument really is a NumPy array of a desired type:

```
/* a_ is a PyObject pointer, representing a sequence
   (NumPy array or list or tuple) */
PyArrayObject *a;
a = (PyArrayObject *) \
    PyArray_FROM_OTF(a_, NPY_DOUBLE, NPY_IN_ARRAY);
```

All the `numpy` functions and methods of arrays that we can access in Python can also be called from C. The NumPy manual has the details.

10.2.2 The Handwritten Extension Code

The complete C code for the extension module is available in the file

```
src/py/mixed/Grid2D/C/plain/gridloop.c
```

As the code is quite long we portion it out in smaller sections along with accompanying comments.

For protecting a newcomer to NumPy programming in C and C++ from potentially intricate errors, I recommend to collect all functions employing the NumPy C API in a single file.

Structure of the Extension Module. A C or C++ extension module contains different sections of code:

- the functions that make up the module (here `gridloop1` and `gridloop2`),
- a method table listing the functions to be called from Python,
- the module's initialization function.

Chapter 10.2.9 presents a C code template where the structure of extension modules is expressed in terms of reusable code snippets.

Header Files. We will need to access to the Python and NumPy C API in our extension module. The relevant header files are

```
#include <Python.h>              /* Python as seen from C */
#include <numpy/arrayobject.h>   /* NumPy  as seen from C */
```

In addition, one needs to include header files needed to perform operations in the C code, e.g., `math.h` for mathematical functions and `stdio.h` for (debug) output.

10.2.3 Sending Arguments from Python to C

The `Grid2Deff.ext_gridloop1` call to the C function `gridloop1` function looks like

```
ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, func)
```

in the Python code. This means that we expect four arguments to the C function. C functions taking input from a Python call are declared with only two arguments:

```
static PyObject *gridloop1(PyObject *self, PyObject *args)
```

Python objects are realized as subclasses of `PyObject`, and `PyObject` pointers are used to represent Python objects in C code. The `self` parameter is used when `gridloop1` is a method in some class, but here it is irrelevant. All positional arguments in the call are available as the tuple `args`. In case of keyword arguments, a third `PyObject` pointer appears as argument, holding a dictionary of keyword arguments (see the "Extending and Embedding the Python Interpreter" chapter in the official Python documentation for more information on keyword arguments).

The first step is to parse `args` and extract the individual variables, in our case three arrays and a function. Such conversion of Python arguments to C variables is performed by `PyArg_ParseTuple`:

```
PyArrayObject *a, *xcoor, *ycoor;
PyObject *func1;

/* arguments: a, xcoor, ycoor, func1 */
if (!PyArg_ParseTuple(args, "O!O!O!O:gridloop1",
                      &PyArray_Type, &a,
                      &PyArray_Type, &xcoor,
                      &PyArray_Type, &ycoor,
                      &func1)) {
  return NULL; /* PyArg_ParseTuple raised an exception */
}
```

The string argument `O!O!O!O:gridloop1` specifies what type of arguments we expect in `args`, here four pointers to Python objects. The string after the colon is the name of the function, which is conveniently inserted in exception messages if something with the conversion goes wrong. In the syntax `O!` the `O` denotes a Python object and the exclamation mark implies a check on the pointer type. Here we expect three NumPy arrays, and for each `O!`, we supply a pair of the pointer type (`PyArray_Type`) and the pointer (`a`, `xcoor`, or `ycoor`).

The fourth argument (`O`) is a Python function, and we represent this variable by a `PyObject` pointer `func1`.

The `PyArg_ParseTuple` function carefully checks that the number and type of arguments are correct, and if not, exceptions are raised. To halt the program and dump the exception, the C code must return `NULL` after the exception is raised. In the present case `PyArg_ParseTuple` returns a false value if errors and corresponding exceptions arise.

Omitting the test for NumPy array pointers allows a quicker argument parsing syntax:

```
if (!PyArg_ParseTuple(args, "OOOO", &a, &xcoor, &ycoor, &func1))
  { return NULL; }
```

10.2.4 Consistency Checks

Before proceeding with computations, it is wise to check that the dimensions of the arrays are consistent and that `func1` is really a callable object. In case we detect inconsistencies, an exception can be raised by calling the `PyErr_Format` function with the exception type as first argument, followed by a message represented by the same arguments as in a `printf` function call. The validity of the `a` array is checked by the code segment

```
if (PyArray_NDIM(a) != 2 || PyArray_TYPE(a) != NPY_DOUBLE) {
  PyErr_Format(PyExc_ValueError,
               "a array is %d-dimensional or not of type double",
               PyArray_NDIM(a));
  return NULL;
}
```

Another consistency check is to test if `xcoor` has the right type and a dimension compatible with `a`:

```
nx = PyArray_DIM(a,0);
if (PyArray_NDIM(xcoor)  != 1 ||
    PyArray_TYPE(xcoor)  != NPY_DOUBLE ||
    PyArray_DIM(xcoor,0) != nx) {
  PyErr_Format(PyExc_ValueError,
  "xcoor array has wrong dimension (%d), type or length (%d)",
               PyArray_NDIM(xcoor),PyArray_DIM(xcoor,0));
  return NULL;
}
```

A similar check is performed for the `ycoor` array. Finally, we check that the `func1` object can be called:

```
if (!PyCallable_Check(func1)) {
  PyErr_Format(PyExc_TypeError,
               "func1 is not a callable function");
  return NULL;
}
```

In Chapter 10.2.7 we show how macros can be used to make the consistency checks more compact and flexible.

10.2.5 Computing Array Values

We have now reached the point where it is appropriate to set up a loop over the entries in `a` and call `func1`. Let us first sketch the loop and how we index `a`. The value to be filled in `a` now stems from a call to a plain C function

```
double f1p(double, double)
```

instead of a callback to Python (as we actually aim at). The loop may be coded as

```
int nx, ny, i, j;
double *a_ij, *x_i, *y_j;
...
for (i = 0; i < nx; i++) {
  for (j = 0; j < ny; j++) {
    a_ij= (double *)(a->data + i*a->strides[0] + j*a->strides[1]);
    x_i = (double *)(xcoor->data + i*xcoor->strides[0]);
    y_j = (double *)(ycoor->data + j*ycoor->strides[0]);

    *a_ij = f1p(*x_i, *y_j);  /* call a C function f1p */
  }
}
```

Observe that the `a_ij` pointer points to the `i,j` entry in `a`. Using the concenience macros `PyArray_GETPTR1` and `PyArray_GETPTR2` we can write the loops as

```
for (i = 0; i < nx; i++) {
  for (j = 0; j < ny; j++) {
    a_ij = (double *) PyArray_GETPTR2(a, i, j);
    x_i  = (double *) PyArray_GETPTR1(xcoor, i);
    y_j  = (double *) PyArray_GETPTR1(ycoor, j);
    *a_ij = f1p(*x_i, *y_j);  /* call a C function f1p */
  }
}
```

For one-dimensional arrays we could also use the simpler indexing `xcoor[i]` instead of computing `x_i` and then dereferencing the value (`*x_i`). Also note that the conversion of the `void` or `char` data pointer from `PyArray_GETPTR1/2` or `a->data + ...` (resp.) to a `double` pointer requires explicit knowledge of what kind of data we are working with.

Callback Functions. In the previous loop we just called a plain C function `f1p` taking the two coordinates of a grid point as arguments. Now we want to call the Python function held by the `func1` pointer instead. This is accomplished by

```
result = PyEval_CallObject(func1, arglist);
```

where `result` is a `PyObject` pointer to the object returned from the `func1` Python function, and `arglist` is a tuple of the arguments to that function.

We need to build arglist from two double variables. Converting C data to Python objects is conveniently done by the Py_BuildValue function. It takes a string specification of the Python data structure and thereafter a list of the C variables contained in that structure. In the present case we want to make a tuple of two doubles. The corresponding string specification is "(dd)":

```
arglist = Py_BuildValue("(dd)", *x_i, *y_j);
```

A documentation of the format specification in Py_BuildValue calls is found in [2,22] or in the Python C API Reference Manual that comes with the official Python documentation (just go to Py_BuildValue in the index and follow the link).

To store the returned function value in the a array we need to convert the returned Python object in result to a double. When we know that result holds a double, parsing of the contents of results can be avoided, and the conversion reads

```
*a_ij = PyFloat_AS_DOUBLE(result);
```

The complete loop, including a debug output, can now be written as

```
for (i = 0; i < nx; i++) {
  for (j = 0; j < ny; j++) {
    a_ij = (double *) PyArray_GETPTR2(a, i, j);
    x_i  = (double *) PyArray_GETPTR1(xcoor, i);
    y_j  = (double *) PyArray_GETPTR1(xcoor, i);
    arglist = Py_BuildValue("(dd)", *x_i, *y_j);
    result = PyEval_CallObject(func1, arglist);
    *a_ij = PyFloat_AS_DOUBLE(result);
#ifdef DEBUG
    printf("a[%d,%d]=func1(%g,%g)=%g\n",i,j,*x_i,*y_j,*a_ij);
#endif
  }
}
```

Memory Management. There is a major problem with the loop above. In each pass we dynamically create two Python objects, pointed to by arglist and result. These objects are not needed in the next pass, but we never inform the Python library that the objects can be deleted. With a 1000×1000 grid we end up with 2 million Python objects when we only need storage for two of them.

Python applies reference counting to track the lifetime of objects. When a piece of code needs to ensure access to an object, the reference count is increased, and when no more access is required, the reference count is decreased. Objects with zero references can safely be deleted. In our example, we do not need the object being pointed to by arglist after the call to func1 is finished. We signify this by decreasing the reference count: Py_DECREF(arglist). Similarly, result points to an object that is not needed after its value is stored in the array. The callback segment should therefore be coded as

```
arglist = Py_BuildValue("(dd)", *x_i, *y_j);
result = PyEval_CallObject(func1, arglist);
Py_DECREF(arglist);
*a_ij = PyFloat_AS_DOUBLE(result);
Py_DECREF(result);
```

Without decreasing the reference count and allowing Python to clean up the objects, I experienced a 40% increase in the CPU time on an 1100×1100 grid.

Another aspect is that our callback function may raise an exception. In that case it returns `NULL`. To pass this exception to the code calling `gridloop1`, we should return `NULL` from `gridloop1` just before the assignment to `*a_ij`:

```
if (result == NULL) return NULL; /* exception in func1 */
```

Without this test, an exception in the callback will give a `NULL` pointer and a segmentation fault in `PyFloat_AS_DOUBLE`.

For further information regarding reference counting and calling Python from C, the reader is referred to the "Extending and Embedding the Python Interpreter" chapter in the official Python documentation.

The Return Statement. The final statement in the `gridloop1` function is the return value as a `PyObject` pointer. We may return `None`, which is done by calling `Py_BuildValue` with an empty string:

```
return Py_BuildValue("");     /* return None */
```

or by

```
Py_INCREF(Py_None);
return Py_None;
```

Alternatively, we could return an integer, say `0` for success:

```
return Py_BuildValue("i",0);  /* return integer 0 */
```

10.2.6 Returning an Output Array

The `gridloop2` function should not take `a` as argument, but create the output array inside the function and return it. The typical call from Python has the form (cf. Chapter 9.1)

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

The signature of the C function is as usual

```
static PyObject *gridloop2(PyObject *self, PyObject *args)
```

This time we expect three arguments:

```
PyArrayObject *a, *xcoor, *ycoor;
PyObject *func1;
int nx, ny;

/* arguments: xcoor, ycoor, func1 */
if (!PyArg_ParseTuple(args, "O!O!O:gridloop2",
                      &PyArray_Type, &xcoor,
                      &PyArray_Type, &ycoor,
                      &func1)) {
  return NULL; /* PyArg_ParseTuple raised an exception */
}
nx = PyArray_DIM(xcoor, 0);  ny = PyArray_DIM(ycoor, 0);
```

Based on `nx` and `ny` we may create the output array using `PyArray_SimpleNew` from the NumPy C API:

```
npy_intp a_dims[2];  a_dims[0] = nx; a_dims[1] = ny;
a = (PyArrayObject *) PyArray_SimpleNew(2, a_dims, NPY_DOUBLE);
```

We should always check if something went wrong with the allocation:

```
if (a == NULL) {
  printf("creating %dx%d array failed\n",
         (int) a_dims[0], (int) a_dims[1]);
  return NULL; /* PyArray_SimpleNew raised an exception */
}
```

Note that we first write a message with `printf` and then an allocation exception from `PyArray_SimpleNew` will appear in the output. Our message provides some additional info that can aid debugging (e.g., a common error is to extract incorrect array sizes elsewhere in the function).

The loop over the array entries is identical to the one in `gridloop1`, but we have introduced some macros to simplify the programming. These macros are presented below.

To return a NumPy array from the `gridloop2` function, we call the function `PyArray_Return`:

```
return PyArray_Return(a);
```

10.2.7 Convenient Macros

Many of the statements in the `gridloop1` function can be simplified and expressed more compactly using macros. A macro that adds quotes to an argument,

```
#define QUOTE(s) # s   /* turn s into string "s" */
```

is useful for writing the name of a variable as a part of error messages.

Checking the number of dimensions, the length of each dimension, and the type of the array entries are good candidates for macros:

```
#define NDIM_CHECK(a, expected_ndim) \
  if (PyArray_NDIM(a) != expected_ndim) { \
    PyErr_Format(PyExc_ValueError, \
    "%s array is %d-dimensional, expected to be %d-dimensional",\
                 QUOTE(a), PyArray_NDIM(a), expected_ndim); \
    return NULL; \
  }
#define DIM_CHECK(a, dim, expected_length) \
  if (PyArray_DIM(a, dim) != expected_length) { \
    PyErr_Format(PyExc_ValueError, \
    "%s array has wrong %d-dimension=%d (expected %d)", \
         QUOTE(a), dim, PyArray_DIM(a, dim), expected_length); \
    return NULL; \
  }
#define TYPE_CHECK(a, tp) \
  if (PyArray_TYPE(a) != tp) { \
    PyErr_Format(PyExc_TypeError, \
    "%s array is not of correct type (%d)", QUOTE(a), tp); \
    return NULL; \
  }
```

We can then write the check of array data like

```
NDIM_CHECK(xcoor, 1); TYPE_CHECK(xcoor, NPY_DOUBLE);
```

Supplying, for instance, a two-dimensional array as the `xcoor` argument will trigger an exception in the `NDIM_CHECK` macro:

```
exceptions.ValueError
xcoor array is 2-dimensional, but expected to be 1-dimensional
```

The `QUOTE` macro makes it easy to write out the name of the array, here `xcoor`. Another macro can be constructed to check that an object is callable.

Macros can also simplify array indexing. For example, it may be convenient to cast the `void` pointer from the `PyArray_GETPTR` macros to specific types, like `double`:

```
#define DIND1(a, i) *((double *) PyArray_GETPTR1(a, i))
#define DIND2(a, i, j) \
 *((double *) PyArray_GETPTR2(a, i, j))
```

Using these, the loop over the grid may be written as

```
for (i = 0; i < nx; i++) {
  for (j = 0; j < ny; j++) {
    arglist = Py_BuildValue("(dd)",DIND1(xcoor,i),DIND1(ycoor,j));
    result = PyEval_CallObject(func1, arglist);
    Py_DECREF(arglist);
    if (result == NULL) return NULL; /* exception in func1 */
    DIND2(a,i,j) = PyFloat_AS_DOUBLE(result);
    Py_DECREF(result);
  }
}
```

The macros shown above are used in the `gridloop2` function. These and some other macros convenient for writing extension modules in C are collected in a file `src/C/NumPy_macros.h`, which can be included in your own C extensions. We refer to the `gridloop.c` file for a complete listing of the `gridloop2` function.

10.2.8 Module Initialization

To form an extension module, we must register all functions to be called from Python in a so-called *method table*. In our case we want to register the two functions `gridloop1` and `gridloop2`. The method table takes the form

```
static PyMethodDef ext_gridloop_methods[] = {
  {"gridloop1",    /* name of func when called from Python */
   gridloop1,      /* corresponding C function */
   METH_VARARGS,   /* ordinary (not keyword) arguments */
   gridloop1_doc}, /* doc string for gridloop1 function */
  {"gridloop2",    /* name of func when called from Python */
   gridloop2,      /* corresponding C function */
   METH_VARARGS,   /* ordinary (not keyword) arguments */
   gridloop2_doc}, /* doc string for gridloop1 function */
  {NULL, NULL}     /* required ending of the method table */
};
```

The predefined C macro `METH_VARARGS` indicates that the function takes two arguments, `self` and `args` in this case, which implies that there are no keyword arguments.

The doc strings are defined as ordinary C strings, e.g.,

```
static char gridloop1_doc[] = \
  "gridloop1(a, xcoor, ycoor, pyfunc)";
static char gridloop2_doc[] = \
  "a = gridloop2(xcoor, ycoor, pyfunc)";
static char module_doc[] = \
  "module ext_gridloop:\n\
   gridloop1(a, xcoor, ycoor, pyfunc)\n\
   a = gridloop2(xcoor, ycoor, pyfunc)";
```

The module needs an initialization function, having the same name as the module, but with a prefix `init`. In this function we must register the method table above along with the name of the module and (optionally) a module doc string. When programming with NumPy arrays we also need to call a function `import_array`:

```
PyMODINIT_FUNC initext_gridloop()
{
  /* Assign the name of the module and the name of the
     method table and (optionally) a module doc string:
  */
  Py_InitModule3("ext_gridloop", ext_gridloop_methods, module_doc);
  /* or without module doc string:
  Py_InitModule ("ext_gridloop", ext_gridloop_methods); */

  import_array();   /* required NumPy initialization */
}
```

10.2.9 Extension Module Template

As summary we outline a template for extension modules involving NumPy arrays:

```
#include <Python.h>              /* Python as seen from C */
#include <numpy/arrayobject.h>   /* NumPy  as seen from C */
#include <math.h>
#include <stdio.h>               /* for debug output */
#include <NumPy_macros.h>        /* useful macros */

static PyObject *modname_function1(PyObject *self, PyObject *args)
{
  PyArrayObject *array1, *array2;
  PyObject *callback, *arglist, *result;
  npy_intp array3_dims[2];
  <more local C variables...>

  /* assume arguments array, array2, callback */
  if (!PyArg_ParseTuple(args, "O!O!O:modname_function1",
                        &PyArray_Type, &array1,
                        &PyArray_Type, &array2,
                        &callback)) {
    return NULL; /* PyArg_ParseTuple has raised an exception */
  }

  <check array dimensions etc.>

  if (!PyCallable_Check(callback)) {
    PyErr_Format(PyExc_TypeError,
    "callback is not a callable function");
    return NULL;
  }
  /* Create output arrays: */
  array3_dims[0] = nx; array3_dims[1] = ny;
  array3 = (PyArrayObject *) \
           PyArray_SimpleNew(2, array3_dims, NPY_DOUBLE);
  if (array3 == NULL) {
    printf("creating %dx%d array failed\n",
           (int) array3_dims[0], (int) array3_dims[1]);
    return NULL; /* PyArray_FromDims raises an exception */
  }

  /* Example on callback:

  arglist = Py_BuildValue(format, var1, var2, ...);
  result = PyEval_CallObject(callback, arglist);
  Py_DECREF(arglist);
  if (result == NULL) return NULL;
  <process result>
  Py_DECREF(result);
  */

  /* Example on array processing:
  for (i = 0; i <= imax; i++) {
    for (j = 0; j <= jmax; j++) {
```

```
      <work with DIND1(array2,i) if array2 is 1-dimensional>
      <or DIND2(array3,i,j) if array2 is 2-dimensional etc.>
      <or IIND1/2/3 for integer arrays>
    }
  }
  */
  return PyArray_Return(array3);
  /* or None: return Py_BuildValue(""); */
  /* or integer: return Py_BuildValue("i", some_int); */
}

static PyObject *modname_function2(PyObject *self, PyObject *args)
{ ... }

static PyObject *modname_function3(PyObject *self, PyObject *args)
{ ... }

/* Doc strings: */
static char modname_function1_doc[] = "...";
static char modname_function2_doc[] = "...";
static char modname_function3_doc[] = "...";
static char module_doc[] = "...";

/* Method table: */
static PyMethodDef modname_methods[] = {
  {"function1",        /* name of func when called from Python */
   modname_function1,  /* corresponding C function */
   METH_VARARGS,       /* positional (no keyword) arguments */
   modname_function1_doc}, /* doc string for function */
  {"function2",        /* name of func when called from Python */
   modname_function2,  /* corresponding C function */
   METH_VARARGS,       /* positional (no keyword) arguments */
   modname_function2_doc}, /* doc string for function */
  {"function3",        /* name of func when called from Python */
   modname_function3,  /* corresponding C function */
   METH_VARARGS,       /* positional (no keyword) arguments */
   modname_function3_doc}, /* doc string for function */
  {NULL, NULL}         /* required ending of the method table */
};

PyMODINIT_FUNC initmodname()
{
  Py_InitModule3("modname", modname_methods, module_doc);
  import_array();   /* required NumPy initialization */
}
```

This file is found as

```
src/misc/ext_module_template.c
```

To get started with a handwritten extension module, copy this file and replace `modname` by the name of the module. Then edit the text according to your needs. With such a template one can make a script for automatically generating much of the code in such a module. More details about this are given in Exercise 10.10.

10.2.10 Compiling, Linking, and Debugging the Module

Compiling and Linking. The next step is to compile the `gridloop.c` file containing the source code of the extension module, and then make a shared library file named `ext_gridloop.so`. This is most easily done using a `setup.py` script:

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

name = 'ext_gridloop'
setup(name=name,
      include_dirs=[os.path.join(os.environ['scripting'],
                                 'src', 'C'),
                    numpy.get_include()],
      ext_modules=[Extension(name, ['gridloop.c'])])
```

To build the module in the current directory we run

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

Thereafter we can test the module in a Python shell:

```
>>> import ext_gridloop as m; print dir(m)
['__doc__', '__file__', '__name__', 'gridloop1', 'gridloop2']
```

If the build procedure based on `setup.py` should fail by some reason, it might be advantageous to manually run the compile and link steps. Here is a Bourne shell script doing this with the Python version and install directories parameterized:

```
root=`python -c 'import sys; print sys.prefix'`
numpy=`python -c 'import numpy; print numpy.get_include()'`
ver=`python -c 'import sys; print sys.version[:3]'`
gcc -O3 -g -I$numpy \
    -I$root/include/python$ver \
    -I$scripting/src/C \
    -c gridloop.c -o gridloop.o
gcc -shared -o ext_gridloop.so gridloop.o
```

Debugging. Writing so much C code as we have to do in the present extension module may easily lead to errors. Inserting lots of tests and raising exceptions (do not forget the `return NULL`) is an efficient technique to make the module development safer and faster. However, low level C code often aborts with "segmentation fault", "bus error", or similar annoying messages. Invoking a debugger is then a quick way to find out where the error arose. On Unix systems one can start the Python interpreter under the `gdb` debugger:

```
unix> which python
/usr/bin/python
unix> gdb /usr/bin/python
...
(gdb) run test.py
```

Here test.py is a script testing the module. When the script crashes, issue the gdb command where to see a traceback. If you compiled the extension module with debugging enabled (usually the -g option), the line number in the C code where the crash occurred will be detectable from the traceback. Doing more than this with gdb is not convenient when running Python under management of gdb.

There is a tool PyDebug (see doc.html), which allows you to print code, examine variables, set breakpoints, etc. under a standard debugger like gdb.

10.2.11 Writing a Wrapper for a C Function

Suppose the gridloop1 function is already available as a C function taking plain C arrays as arguments:

```
void gridloop_C(double **a, double *xcoor, double *ycoor,
                int nx, int ny, Fxy func1)
{
  int i, j;
  for (i=0; i<nx; i++) {
    for (j=0; j<ny; j++) {
       a[i][j] = func1(xcoor[i], ycoor[j]);
    }
  }
}
```

Here, func1 is a pointer to a standard C function taking two double arguments and returning a double. The pointer is defined as

```
typedef double (*Fxy)(double x, double y);
```

Such code is frequently a starting point. How can we write a wrapper for such a function? The answer is of particular interest if we want to interface C functions in existing libraries. Basically, we can write a wrapper function like gridloop1 and gridloop2, but migrate the loop over the a array to the gridloop_C function above. However, we face two major problems:

– The gridloop_C function takes a C matrix a, represented as a double pointer (double**). The provided NumPy array represents the data in a by a single pointer.
– The function to be called for each grid point is in gridloop_C a function pointer, not a PyObject callable Python object as provided by the calling Python code.

To solve the first problem, we may allocate the necessary extra data, i.e., a pointer array, in the wrapper code before calling gridloop1_C. The second problem might be solved by storing the PyObject function object in a global pointer variable and creating a function with the specified Fxy interface that performs the callback to Python using the global pointer.

The C function `gridloop1_C` is implemented in a file `gridloop1_C.c`. A prototype of the function and a definition of the `Fxy` function pointer is collected in a corresponding header file `gridloop_C.h`. The wrapper code, offering `gridloop1` and `gridloop2` functions to be called from Python, as defined in Chapter 9.1, is implemented in the file `gridloop_wrap.c`. All these files are found in the directory

```
src/mixed/py/Grid2D/C/clibcall
```

Conversion of a Two-Dimensional NumPy Array to a Double Pointer. The double pointer argument `double **a` in the `gridloop_C` function is an array of `double*` pointers, where pointer no. i points to the first element in the i-th row of the two-dimensional array of data. The array of pointers is not available as part of a NumPy array. The NumPy array struct only has a `char*` or `void*` pointer to the beginning of the data block containing all the array entries. We may cast this pointer to a `double*` pointer, allocate a new array of `double*` entries, and then set these entries to point at the various rows of the two-dimensional NumPy array:

```
/* a is a PyArrayObject* pointer */
double **app;  double *ap;

ap = (double *) PyArray_DATA(a);
/* allocate the pointer array: */
app = (double **) malloc(nx*sizeof(double*));
/* set each entry of app to point to rows in ap: */
for (i = 0; i < nx; i++) {
  app[i] = &(ap[i*ny]);
}

.... call gridloop_C ...

free(app);  /* deallocate the app array */
```

The NumPy C API has convenience functions for making this code segment shorter:

```
double **app;
npy_intp *app_dims;
PyArray_AsCArray(&a, (void*) &app, app_dims, 2, NPY_DOUBLE, 0);
.... call gridloop_C ...
PyArray_Free(a, (void*) &app);
```

The Callback to Python. The `gridloop1_C` function requires the grid point values to be computed by a function of the form

```
double somefunc(double x, double y)
```

while the calling Python code provides a Python function accessible through a `PyObject` pointer in the wrapper code. To resolve the problem with incompatible function representations, we may store the `PyObject` pointer to the provided Python function as a global `PyObject` pointer `_pyfunc_ptr`. We can then create a generic function, with the signature dictated by the definition of the `Fxy` function pointer, which applies `_pyfunc_ptr` to perform the callback to Python:

```
double _pycall(double x, double y)
{
  PyObject *arglist, *result; double C_result;
  arglist = Py_BuildValue("(dd)", x, y);
  result = PyEval_CallObject(_pyfunc_ptr, arglist);
  Py_DECREF(arglist);
  if (result == NULL) { /* cannot return NULL... */
    printf("Error in callback..."); exit(1);
  }
  C_result = PyFloat_AS_DOUBLE(result);
  Py_DECREF(result);
  return C_result;
}
```

This `_pycall` function is a general wrapper code for all callbacks Python functions taking two floats as input and returning a float.

The Wrapper Functions. The `gridloop1` wrapper now extracts the arguments sent from Python, stores the Python function in `_pyfunc_ptr`, builds the double pointer structure, and calls `gridloop_C`:

```
PyObject* _pyfunc_ptr = NULL;  /* init of global variable */


static PyObject *gridloop1(PyObject *self, PyObject *args)
{
  PyArrayObject *a, *xcoor, *ycoor;
  PyObject *func1;
  int nx, ny, i;
  double **app;
  double *ap, *xp, *yp;

  /* arguments: a, xcoor, ycoor, func1 */
  /* parsing without checking the pointer types: */
  if (!PyArg_ParseTuple(args, "OOOO", &a, &xcoor, &ycoor, &func1))
    { return NULL; }
  nx = PyArray_DIM(a,0);  ny = PyArray_DIM(a,1);
  NDIM_CHECK(a,     2)
  TYPE_CHECK(a,     NPY_DOUBLE);
  NDIM_CHECK(xcoor, 1); DIM_CHECK(xcoor, 0, nx);
  TYPE_CHECK(xcoor, NPY_DOUBLE);
  NDIM_CHECK(ycoor, 1); DIM_CHECK(ycoor, 0, ny);
  TYPE_CHECK(ycoor, NPY_DOUBLE);
  CALLABLE_CHECK(func1);
  _pyfunc_ptr = func1;  /* store func1 for use in _pycall */

  /* allocate help array for creating a double pointer: */
  app = (double **) malloc(nx*sizeof(double*));
  ap = (double *) PyArray_DATA(a);
  for (i = 0; i < nx; i++) { app[i] = &(ap[i*ny]); }
  xp = (double *) PyArray_DATA(xcoor);
```

```
    yp = (double *) PyArray_DATA(ycoor);
    gridloop_C(app, xp, yp, nx, ny, _pycall);
    free(app);
    return Py_BuildValue("");  /* return None */
  }
```

Note that we have used the macros from Chapter 10.2.7 to perform consistency tests on the arrays sent from Python.

The `gridloop2` function is almost identical, the only difference being that the NumPy array `a` is allocated in the function and not provided by the calling Python code. The statements for doing this are the same as for the previous version of the C extension module. In addition we must code the doc strings, method table, and the initializing function. We refer to the previous sections or to the `gridloop_wrap.c` file for all details.

The Python script `Grid2Deff.py`, which calls the `ext_gridloop` module, is outlined in Chapter 9.1.

10.3 C++ Programming with NumPy Arrays

Now we turn the attention to implementing the `gridloop1` and `gridloop2` functions with aid of C++. The reader should, before continuing, be familiar with the problem setting, as explained in Chapter 9.1, and programming with the NumPy C API, as covered in Chapter 10.2. The code we present in the following is, in a nutshell, just a more user-friendly wrapping of the C code from Chapter 10.2.

C++ programmers may claim that abstract data types can be used to hide many of the low-level details of the implementation in Chapter 10.2 and thereby simplify the development of extension modules significantly. We will show how classes can be used in various ways to achieve this. Chapter 10.3.1 deals with wrapping NumPy arrays in a more user-friendly, yet very simple, C++ array class. Chapter 10.3.2 applies the C++ library SCXX to simplify writing wrapper code, using the power of C++ to increase the abstraction level. In Chapter 10.3.3 we explain how NumPy arrays can be converted to and from the programmer's favorite C++ array class.

10.3.1 Wrapping a NumPy Array in a C++ Object

The most obvious improvement of the C versions of the functions `gridloop1` and `gridloop2` is to encapsulate NumPy arrays in a class to make creation and indexing more convenient. Such a class should support arrays of varying dimension. Our very simple implementation works for one-, two-, and three-dimensional arrays. To save space, we outline only the parts of the class relevant for two-dimensional arrays:

```
class NumPyArray_Float
{
 private:
  PyArrayObject* a;

 public:
  NumPyArray_Float () { a=NULL; }
  NumPyArray_Float (int n1, int n2)  { create(n1, n2); }
  NumPyArray_Float (double* data, int n1, int n2)
    { wrap(data, n1, n2); }
  NumPyArray_Float (PyArrayObject* array) { a = array; }

  int create (int n1, int n2) {
    npy_intp dim2[2]; dim2[0] = n1; dim2[1] = n2;
    a = (PyArrayObject*) PyArray_SimpleNew(2, dim2, NPY_DOUBLE);
    if (a == NULL) { return 0; } else { return 1; } }

  void wrap (double* data, int n1, int n2) {
    npy_intp dim2[2]; dim2[0] = n1; dim2[1] = n2;
    a = (PyArrayObject*) PyArray_SimpleNewFromData(
        2, dim2, NPY_DOUBLE, (void *) data);
  }

  int checktype () const;
  int checkdim  (int expected_ndim) const;
  int checksize (int expected_size1, int expected_size2=0,
                 int expected_size3=0) const;

  double  operator() (int i, int j) const
  { return *((double*) PyArray_GETPTR2(a,i,j)); }
  double& operator() (int i, int j)
  { return *((double*) PyArray_GETPTR2(a,i,j)); }

  int dim()   const { return PyArray_NDIM(a);  }
  int size1() const { return PyArray_DIM(a,0); }
  int size2() const { return PyArray_DIM(a,1); }
  PyArrayObject* getPtr () { return a; }
};
```

The `create` function allocates a new array, whereas the `wrap` function just wraps an existing plain memory segment as a NumPy array. One of the constructors also wrap a `PyArrayObject` struct as a `NumPyArray_Float` object. Some boolean functions `checktype`, `checkdim`, and `checksize` check if the array has the anticipated properties. The probably most convenient feature of the class is the `operator()` function for indexing arrays. The complete implementation of the class is found in the files `NumPyArray.h` and `NumPyArray.cpp` in the directory `src/py/mixed/Grid2D/C++/plain`. Observe that there is no destructor in the class for freeing memory created by the `create` functions. Since such a class will frequently lend out `a` to other parts of the C and Python code (cf. `gridloop2`), the memory management must use proper reference counting (which is quite straightforward, but the details clutter the exposition of the basics of this class, and for our purposes here the empty default destructor is sufficient).

The `gridloop1` and `gridloop2` functions follow the patterns explained in Chapter 10.2, except that they wrap `PyArrayObject` data structures in the new C++ `NumPyArray_Float` objects to enable use of more readable indexing as well as more compact checking of array properties. Here is the `gridloop2` code utilizing class `NumPyArray_Float`:

```
static PyObject* gridloop2(PyObject* self, PyObject* args)
{
  PyArrayObject *xcoor_, *ycoor_;
  PyObject *func1, *arglist, *result;

  /* arguments: xcoor, ycoor, func1 */
  if (!PyArg_ParseTuple(args, "O!O!O:gridloop2",
                        &PyArray_Type, &xcoor_,
                        &PyArray_Type, &ycoor_,
                        &func1)) {
    return NULL; /* PyArg_ParseTuple raised an exception */
  }
  NumPyArray_Float xcoor (xcoor_); int nx = xcoor.size1();
  if (!xcoor.checktype()) { return NULL; }
  if (!xcoor.checkdim(1)) { return NULL; }
  NumPyArray_Float ycoor (ycoor_); int ny = ycoor.size1();
  // check ycoor dimensions, check that func1 is callable...
  NumPyArray_Float a(nx, ny);  // return array

  int i,j;
  for (i = 0; i < nx; i++) {
    for (j = 0; j < ny; j++) {
      arglist = Py_BuildValue("(dd)", xcoor(i), ycoor(j));
      result = PyEval_CallObject(func1, arglist);
      Py_DECREF(arglist);
      if (result == NULL) return NULL; /* exception in func1 */
      a(i,j) = PyFloat_AS_DOUBLE(result);
      Py_DECREF(result);
    }
  }
  return PyArray_Return(a.getPtr());
}
```

The `gridloop1` function is constructed in a similar way. Both functions are placed in a file `gridloop.cpp`. This file also contains the method table and initializing function. These are as explained in Chapter 10.2.8.

As mentioned on page 491, a special hack is needed if we access the NumPy C API in multiple files within the same extension module. Therefore, we include both the header file of class `NumPyArray_Float` and the corresponding C++ file (with the body of some member functions) in `gridloop.cpp` and compile this file only.

10.3.2 Using SCXX

Memory management is hidden in Python scripts. Objects can be brought into play when needed, and Python destroys them when they are no longer in use. This memory management is based on tracking the number of references of each object, as briefly mentioned in Chapter 10.2.5. In extension modules, the reference counting must be explicitly dealt with by the programmer, and this can be a quite complicated task. This is the reason why we only briefly touch reference counting technicalities in this book. Fortunately, there are some C++ layers on top of the Python C API where the reference counting is hidden in C++ objects. Examples on such layers are CXX, SCXX, and Boost.Python (see `doc.html` for references to documentation of these tools). In the following we shall exemplify SCXX, which is by far the simplest of these tools, both with respect to design, functionality, and usage.

SCXX was developed by Gordon McMillan and consists of a thin layer of C++ classes on top of the Python C API. For each basic Python type, such as numbers, tuples, lists, dictionaries, and functions, there is a corresponding C++ class encapsulating the underlying C struct and its associated functions. The result is simpler and more convenient programming with Python objects in C++. The documentation is very sparse, but if you have some knowledge of the Python C API and know C++ quite well, it should be straightforward to use the code in the header files as documentation of SCXX.

Here is an example concerning creation of numbers, adding two numbers, filling a list, converting the list to a tuple, and writing out the elements in the tuple:

```
#include <PWONumber.h>     // class for numbers
#include <PWOSequence.h>   // class for tuples
#include <PWOMSequence.h>  // class for lists (immutable sequences)

void test_scxx()
{
  double a_ = 3.4;
  PWONumber a = a_; PWONumber b = 7;
  PWONumber c; c = a + b;
  PWOList list; list.append(a).append(c).append(b);
  PWOTuple tp(list);
  for (int i=0; i<tp.len(); i++) {
    std::cout << "tp["<<i<<"]="<<double(PWONumber(tp[i]))<<" ";
  }
  std::cout << std::endl;
  PyObject* py_a = (PyObject*) a;  // convert to Python C struct
}
```

For comparison, the similar C++ code, employing the plain Python C API, may look like this (without any reference counting):

```
void test_PythonAPI()
{
  double a_ = 3.4;
  PyObject* a = PyFloat_FromDouble(a_);
  PyObject* b = PyFloat_FromDouble(7);
  PyObject* c = PyNumber_Add(a, b);
  PyObject* list = PyList_New(0);
  PyList_Append(list, a);
```

```
    PyList_Append(list, c);
    PyList_Append(list, b);
    PyObject* tp = PyList_AsTuple(list);
    int tp_len = PySequence_Length(tp);
    for (int i=0; i<tp_len; i++) {
      PyObject* qp = PySequence_GetItem(tp, i);
      double q = PyFloat_AS_DOUBLE(qp);
      std::cout << "tp[" << i << "]=" << q << " ";
    }
    std::cout << std::endl;
  }
```

If we point to a tuple item by `qp` and send this pointer to another code segment, we need to update the reference counter such that neither the item nor the tuple is deleted before our code has finished the use of these data. This is automatically taken care of when programming with SCXX.

Let us take advantage of SCXX in the `gridloop.cpp` code. The modified file, called `gridloop_scxx.cpp`, resides in `src/py/mixed/Grid2D/C++/scxx`. Parsing of arguments is quite different with SCXX:

```
static PyObject* gridloop1(PyObject* self, PyObject* args_)
{
  /* arguments: a, xcoor, ycoor, func1 */
  try {
    PWOSequence args (args_);
    NumPyArray_Float a ((PyArrayObject*) ((PyObject*) args[0]));
    NumPyArray_Float xcoor ((PyArrayObject*) ((PyObject*) args[1]));
    NumPyArray_Float ycoor ((PyArrayObject*) ((PyObject*) args[2]));
    PWOCallable func1 (args[3]);

    // work with a, xcoor, ycoor, and func1
    ...

    return PWONone();
  }
  catch (PWException e) { return e; }  // wrong args_
}
```

The error checking of `NumPyArray_Float` objects is explained in the `gridloop2` code from Chapter 10.3.1. Checking that `func1` is a callable object can be carried out by the built-in function `isCallable` in a `PWOCallable` object:

```
    if (!func1.isCallable()) {
      PyErr_Format(PyExc_TypeError,
                   "func1 is not a callable function");
      return NULL;
    }
```

The loop over the array entries take advantage of (i) a `PWOTuple` object to represent the arguments of the callback function, (ii) a member function `call` in `func1` for calling the Python function, and (iii) SCXX conversion operators for turning C numbers into corresponding SCXX objects. Here is the code:

```
int i,j;
for (i = 0; i < nx; i++) {
  for (j = 0; j < ny; j++) {
    PWOTuple arglist(Py_BuildValue("(dd)", xcoor(i), ycoor(j)));
    PWONumber result(func1.call(arglist));
    a(i,j) = double(result);
  }
}
```

The `gridloop2` function is similar, the only difference being an argument less and the creation of an internal array object. The latter task is shown in the `gridloop2` function in Chapter 10.3.1. The method table and initialization function are coded as shown in Chapter 10.2.8.

The base class `PWOBase` of all SCXX classes performs the reference counting of objects. By subclassing `PWOBase`, our simple `NumPyArray_Float` class can easily be equipped with reference counting. Every time the `PyArrayObject*` pointer `a` is bound to a new NumPy C struct, we call

```
PWOBase::GrabRef((PyObject*) a);
```

This is done in all the `create` and `wrap` functions in class `NumPyArray_Float` in a new version of the class found in the directory `src/py/mixed/Grid2D/C++/scxx`. The calling Python code (`Grid2Deff.py`) is described in Chapter 9.1 and independent of how we actually implement the extension module.

10.3.3 NumPy–C++ Class Conversion

In the two previous C++ implementations of the `ext_gridloop` extension module we showed how to access NumPy arrays through C++ classes, with the purpose of simplifying programming with NumPy arrays. The developed C++ classes could not be accessed from Python since we did not create corresponding wrapper code. Using SWIG, wrapping C++ classes might be as straightforward as shown in Chapter 5.2.4. However, there are many cases where we want to grab data from one library and send it to another, via Python, without having to create interfaces to all classes and functions in all libraries. The present section will show how we can just grab a pointer from one library and convert it to a data object suitable for the other library. To this end, we make a *conversion* class.

To make the setting relevant for many numerical Python-C++ couplings, we assume that we have a favorite class library, here called `MyArray`, which we want to use extensively in numerical algorithms being coded either in C++ or Python. We do not bother with interfacing the whole `MyArray` class. Instead we make a special class with static functions for converting a `MyArray` object to a NumPy array and vice versa. The conversion functions in this class can be called from manually written wrapper functions, or we can use SWIG to automatically generate the wrapper code. SWIG is straightforward to use because the conversion functions have only pointers or references as input and output data. The calling Python code must explicitly convert its NumPy reference to a `MyArray` reference before invoking the `gridloop1` and `gridloop2` functions. SWIG can communicate these references as C pointers between Python and C, without any need for information about the type of data the pointers are pointing to. The source code related to the present example will explain the attractive simplicity of pointer communication and SWIG in more detail.

The C++ Array Class. As a prototype of a programmer's array class in some favorite array library, we have created a minimal array class:

```
template< typename T > class MyArray
{
 public:
  T* A;                   // the data
  int ndim;               // no of dimensions (axis)
  int size[MAXDIM];       // size/length of each dimension
  int length;             // total no of array entries
  T* allocate(int n1);
  T* allocate(int n1, int n2);
  T* allocate(int n1, int n2, int n3);
  void deallocate();
  bool indexOk(int i) const;
  bool indexOk(int i, int j) const;
  bool indexOk(int i, int j, int k) const;

 public:
  MyArray() { A = NULL; length = 0; ndim = 0; }
  MyArray(int n1) { A = allocate(n1); }
  MyArray(int n1, int n2) { A = allocate(n1, n2); }
  MyArray(int n1, int n2, int n3) { A = allocate(n1, n2, n3); }
  MyArray(T* a, int ndim_, int size_[]);
  MyArray(const MyArray<T>& array);
   MyArray() { deallocate(); }

  bool redim(int n1);
  bool redim(int n1, int n2);
  bool redim(int n1, int n2, int n3);

  // return the size of the arrays dimensions:
  int shape(int dim) const { return size[dim-1]; }

  // indexing:
  const T& operator()(int i) const;
  T& operator()(int i);
  const T& operator()(int i, int j) const;
  T& operator()(int i, int j);
  const T& operator()(int i, int j, int k) const;
  T& operator()(int i, int j, int k);

  MyArray<T>& operator= (const MyArray<T>& v);


  // return pointers to the data:
  const T* getPtr() const { return A;}
  T* getPtr() { return A; }
```

```
  void print_(std::ostream& os);
  void dump(std::ostream& os);  // dump all
};
```

The `allocate` functions perform the memory allocation for one-, two-, and three-dimensional arrays. The `indexOk` functions check that an index is within the array dimensions. The `redim` functions enable redimensioning of an existing array object and return true if new memory is allocated. Hopefully, the rest of the functions are self-explanatory, at least for readers familiar with how C++ array classes are constructed (the books [1] and [15] are sources of information).

The complete code is found in `MyArray.h` and `MyArray.cpp`. Both files are located in the directory

```
src/py/mixed/Grid2D/C++/convertptr
```

The Grid Loop Using MyArray. Having the `MyArray` class as our primary array object, we can use the following function to compute an array of grid point values:

```
void gridloop1(MyArray<double>& a,
               const MyArray<double>& xcoor,
               const MyArray<double>& ycoor,
               Fxy func1)
{
  int nx = a.shape(1), ny = a.shape(2);
  int i, j;
  for (i = 0; i < nx; i++) {
    for (j = 0; j < ny; j++) {
      a(i,j) = func1(xcoor(i), ycoor(j));
    }
  }
}
```

Here, `Fxy` is a function pointer as defined in Chapter 10.2.11, i.e., `func1` must be a C/C++ function taking two `double` arguments and returning a `double`. Alternatively, `func1` could be a C++ functor, i.e., a C++ object with an overloaded `operator()` function such that we can call the object as a plain function.

We have also made a `gridloop2` function without the `a` array as an argument. Instead, `a` is created inside the function, by a `new` statement, and passed out of the function by a `return a` statement.

Conversion Functions: NumPy to/from `MyArray`. We need some functions for converting NumPy arrays to `MyArray` objects and back again. These conversion functions can be collected in a C++ class:

```
class Convert_MyArray
{
 public:
```

```
    Convert_MyArray();
    Convert_MyArray();

    // borrow data:
    PyObject*        my2py (MyArray<double>& a);
    MyArray<double>* py2my (PyObject* a);

    // copy data:
    PyObject*        my2py_copy (MyArray<double>& a);
    MyArray<double>* py2my_copy (PyObject* a);

    // npy_intp to/from int array for array size:
    npy_intp         npy_size[MAXDIM];
    int              int_size[MAXDIM];
    void             set_npy_size(int*      dims, int nd);
    void             set_int_size(npy_intp* dims, int nd);

    // print array:
    void             dump(MyArray<double>& a);

    // convert Py function to C/C++ function calling Py:
    Fxy              set_pyfunc (PyObject* f);
   protected:
    static PyObject* _pyfunc_ptr;  // used in _pycall
    static double    _pycall (double x, double y);
  };
```

The `_pycall` function is, as in Chapter 10.2.11, a wrapper for the provided Python function to be called at each grid point. A `PyObject` pointer to this function is stored in the class variable `_pyfunc_ptr`. This variable, as well as the `_pycall` function, are static members of the conversion class. That is, instead of being global data as in the C code in Chapter 10.2.11, they are collected in a class namespace `Convert_MyArray`. The `_pycall` function is static such that we can use it as a stand-alone C/C++ function for the `func1` argument in the `gridloop1` and `gridloop2` functions. When `_pycall` is static, it also requires the class data it accesses, in this case `_pyfunc_ptr`, to be static.

Let us briefly show the bodies of the conversion functions. The constructor must call `import_array`:

```
Convert_MyArray:: Convert_MyArray() { import_array(); }
```

This is a crucial point: forgetting the call leads to a segmentation fault the first time a function in the NumPy C API is called. Tracking down this error may be frustrating. In previous examples, we have placed the `import_array` in the module's initialization function, but this time we plan to automatically generate wrapper code by SWIG. It is then easy to forget the `import_array` call.

Converting a `MyArray` object to a NumPy array is done in the following function:

```
PyObject* Convert_MyArray:: my2py(MyArray<double>& a)
{
```

```
  set_npy_size(a.size, a.ndim);
  PyArrayObject* array =  (PyArrayObject*) \
          PyArray_SimpleNewFromData(a.ndim, npy_size, NPY_DOUBLE,
                                    (void *) a.A);
  if (array == NULL) {
    return NULL; /* exception was raised */
  }
  return PyArray_Return(array);
}
```

Observe that we need to copy the dimension information from NumPy's representation, based on an `npy_intp*` pointer, to `MyArray`'s representation, based on an `int*` pointer. This is done by the functions `set_npy_size` and `set_int_size`, which simply fills statically allocated arrays in the class.

The `my2py` function is memory friendly: the data segment holding the array entries in the `MyArray` object is reused directly in the NumPy array. This requires that the memory layout used in `MyArray` matches the layout in NumPy objects. Fortunately, `MyArray` stores the entries in the same way as NumPy arrays, i.e., row by row with a pointer to the first array entry. The data type of the array elements must also be identical (here C `double` or Python/NumPy `float`).

Other C++ array classes may apply a different storage scheme. In such cases data must be *copied* back and forth between the NumPy struct and the C++ array object. We might request copying in the present context as well, so the `my2py` function has a counterpart for copying data:

```
PyObject* Convert_MyArray:: my2py_copy(MyArray<double>& a)
{
  set_npy_size(a.size, a.ndim);
  PyArrayObject* array =  (PyArrayObject*) \
          PyArray_SimpleNew(a.ndim, npy_size, NPY_DOUBLE);
  if (array == NULL) {
    return NULL; /* PyArray_SimpleNew raised an exception */
  }
  double* ad = (double*) PyArray_DATA(array);
  for (int i = 0; i < a.length; i++) {
    ad[i] = a.A[i];
  }
  return PyArray_Return(array);
}
```

The conversion from NumPy arrays to `MyArray` objects is particularly simple since `MyArray` is equipped with a constructor that takes the raw data available in the NumPy C struct and creates a corresponding C++ `MyArray` object:

```
MyArray<double>* Convert_MyArray:: py2my(PyObject* a_)
{
  PyArrayObject* a = (PyArrayObject*) a_;
  // borrow the data, but wrap it in MyArray:
  set_int_size(PyArray_DIMS(a), PyArray_NDIM(a));
  MyArray<double>* ma = new MyArray<double> \
```

```
      ((double*) PyArray_DATA(a), PyArray_NDIM(a), int_size);
  return ma;
}
```

If not a NumPy-compatible constructor is available, which is normally the case in a C++ array class, one needs more statements to extract data from the NumPy C struct and feed them into the appropriate creation function in the C++ class.

The `py2my` function above can be made slightly more general by allowing `a_` to be an arbitrary Python sequence (list, tuple, NumPy array). Using the function `PyArray_FROM_OTF` in the NumPy C API, we can transform any Python sequence to a NumPy array:

```
MyArray<double>* Convert_MyArray:: py2my(PyObject* a_)
{
  PyArrayObject* a = (PyArrayObject*)
    PyArray_FROM_OTF(a_, PyArray_DOUBLE, NPY_IN_ARRAY);
  if (a == NULL) { return NULL; }
  // borrow the data, but wrap it in MyArray:
  set_int_size(PyArray_DIMS(a), PyArray_NDIM(a));
  MyArray<double>* ma = new MyArray<double> \
      ((double*) PyArray_DATA(a), PyArray_NDIM(a), int_size);
  return ma;
}
```

The `PyArray_FROM_OTF` function copies the original data to a new data structure if the type does not match or if the original sequence is not stored in a contiguous memory segment.

The `MyArray` object computed by the `py2my` function borrows the array data from the NumPy array. If we want the `MyArray` object to store a copy of the data, a slightly different function is needed:

```
MyArray<double>* Convert_MyArray:: py2my_copy(PyObject* a_)
{
  PyArrayObject* a = (PyArrayObject*)
    PyArray_FROM_OTF(a_, PyArray_DOUBLE, NPY_IN_ARRAY);
  if (a == NULL) { return NULL; }

   MyArray<double>* ma = new MyArray<double>();
  if (PyArray_NDIM(a) == 1) {
    ma->redim(PyArray_DIM(a,0));
  } else if (PyArray_NDIM(a) == 2) {
    ma->redim(PyArray_DIM(a,0), PyArray_DIM(a,1));
  }
  // copy data:
  double* ad = (double*) PyArray_DATA(a);
  double* mad = ma->A;
  for (int i = 0; i < ma->length; i++) {
    mad[i] = ad[i];
  }
  return ma;
}
```

A part of the `Convert_MyArray` class is devoted to handling callbacks to Python. A general callback function for all Python functions taking two floats and returning a float is `_pycall` from page 505, now written in the current C++ context:

```
double Convert_MyArray:: _pycall (double x, double y)
{
  PyObject* arglist = Py_BuildValue("(dd)", x, y);
  PyObject* result =  PyEval_CallObject(
                      Convert_MyArray::_pyfunc_ptr, arglist);
  Py_DECREF(arglist);
  if (result == NULL) { /* cannot return NULL... */
    printf("Error in callback..."); exit(1);
  }
  double C_result = PyFloat_AS_DOUBLE(result);
  Py_DECREF(result);
  return C_result;
}
```

This function assumes that the Python function to call is pointed to by the `Convert_MyArray::_pyfunc_ptr` pointer. This pointer is defined with an initial value,

```
PyObject* Convert_MyArray::_pyfunc_ptr = NULL;
```

and set explicitly in the calling Python code by invoking

```
Fxy Convert_MyArray:: set_pyfunc (PyObject* f)
{
  _pyfunc_ptr = f;
  Py_INCREF(_pyfunc_ptr);
  return _pycall;
}
```

Later we show exactly how this and other functions are used from Python. Notice that we increase the reference count of `_pyfunc_ptr`. Without the `Py_INCREF` call there is a danger that Python deletes the function object before we have finished our use of it. It will therefore also be necessary to decrease the reference count in the destructor of `Convert_MyArray`:

```
Convert_MyArray::  Convert_MyArray()
{
  if (_pyfunc_ptr != NULL)
    Py_DECREF(_pyfunc_ptr);
}
```

The SWIG Interface File. Our plan is to wrap the conversion functions, i.e., class `Convert_MyArray`, plus functions computing with `MyArray` objects, here `gridloop1` and `gridloop2` (see page 513). A central point is that we do not wrap the `MyArray` class. This means that we cannot create `MyArray` instances directly in Python. Instead, we create a NumPy array and call a conversion function returning a `MyArray` pointer, which can be fed into lots of computational routines. This demonstrates that Python can work with C++ data types that we have not run SWIG on. For a large C++ library the principle is important (cf. Chapter 5.4) because we can generate quite functional Python interfaces without SWIG-ing all the key classes (which might be non-trivial or even tricky).

The SWIG interface file has the same name as the module, `ext_gridloop`, with the `.i` extension. The file can be made very short as we just need to create an interface to the `Convert_MyArray` class and the grid loop functions, i.e., the functions and data defined in `convert.h` and `gridloop.h`:

```
/* file: ext_gridloop.i */
%module ext_gridloop
%{
#include "convert.h"
#include "gridloop.h"
%}

%include "convert.h"
%include "gridloop.h"
```

Running SWIG,

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

generates the wrapper code in `ext_gridloop_wrap.cxx`. This file, together with `convert.cpp` and `gridloop.cpp` must be compiled and linked to a shared library file with name `_ext_gridloop.so`. You can inspect the Bourne shell script `make_module_1.sh` to see the steps of a manual build. As an alternative, `make_module_2.sh` runs a `setup.py` script to build the extension module.

The Calling Python Code. The `Grid2Deff.py` script needs to be slightly adjusted to utilize the new extension module, since we need to explicitly perform the conversion to and from NumPy and `MyArray` data structures in Python. Instead of just calling

```
ext_gridloop.gridloop1(a, self.xcoor, self.ycoor, func)
return a
```

in the `ext_gridloop1` function, we must introduce the conversion from NumPy arrays to `MyArray` objects:

```
a_p = self.c.py2my(a)
x_p = self.c.py2my(self.xcoor)
y_p = self.c.py2my(self.ycoor)
f_p = self.c.set_pyfunc(func)
ext_gridloop.gridloop1(a_p, x_p, y_p, f_p)
return a
```

Note that we can just return `a` since the filling of `a_p` in `gridloop1` actually fills the borrowed data structures from `a`. If we had converted `a` to `a_p` by the copy function,

```
a_p = self.c.py2my_copy(a)
```

the `gridloop1` function would have filled a local data segment in the `MyArray` object `a_p`, and we would need to copy the data back to a NumPy array object before returning:

```
a = self.c.my2py_copy(a_p)
return a
```

Calling `gridloop2` follows the same set-up, but now we get a `MyArray` object from `gridloop2`, and this object needs to be converted to a NumPy array to be returned from `ext_gridloop2`:

```
x_p = self.c.py2my(self.xcoor)
y_p = self.c.py2my(self.ycoor)
f_p = self.c.set_pyfunc(func)
a_p = ext_gridloop.gridloop2(x_p, y_p, f_p)
a = self.c.my2py(a_p)
return a
```

We repeat that SWIG does not know about the members of `MyArray` or the NumPy C struct. SWIG just sees the two pointer types `MyArray*` and `PyArrayObject*`. This fact makes it easy to quickly interface large libraries without the need to interface all pieces of the libraries.

10.4 Comparison of the Implementations

In Chapters 9–10.3 we have described numerous implementations of an extension module for filling a uniform, rectangular, two-dimensional grid with values at the grid points. Each point value is computed by a function or formula in the steering Python script. The various implementations cover

- Fortran 77 subroutines, automatically wrapped by F2PY, with different techniques for handling callbacks,
- handwritten extension module written in C,
- handwritten extension modules written in C++, using C++ array classes and the SCXX interface to Python.

This section looks at the computational efficiency of these implementations, we compare error handling, and we summarize our experience with writing the F77, C, and C++ extension modules.

10.4.1 Efficiency

After having spent much efforts on various implementations of the `gridloop1` and `gridloop2` functions it is natural to compare the speed of these implementations with pure Fortran, C, and C++ code. When invoking `gridloop1` and gridloop2 from Python in our efficiency tests, we make a callback to the Python function

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

at every grid point.

A word of caution about the implementation of the callback function is necessary. The myfunc function is now aimed at scalar arguments x and y. We should therefore make sure that sin is the sine function for scalar arguments from the math module and not the sin function from numpy. We have tested both versions to quantify the performance loss of using vectorized sine functions in a scalar context.

The timing2 function in the Grid2Deff module performs the tests with a particular extension module. The Bourne shell script

```
src/py/mixed/Grid2D/efficiency-tests.sh
```

visits all relevant directories and executes all tests, including stand-alone F77 and C++ programs where the myfunc function above is implemented in compiled code. Simulations with Numeric and numarray arrays were done on my IBM X30 laptop with Linux, GNU compilers v3.3, Python v2.3.3, Numeric v23, and numarray v0.9. Later, simulations with numpy were performed with Python v2.5, GNU compilers v4.0, and numpy v1.0.4. The combined results are displayed in Table 10.1.

The fastest implementation of the current problem is to code the loops and the function evaluation solely in Fortran 77. All CPU times in Table 10.1 have been scaled by the CPU time of this fastest implementation.

The second row in Table 10.1 refers to the C++ code in the convertptr directory, where the NumPy array is wrapped in a MyArray class, and the computations are expressed in terms of MyArray functionality. The overhead in using MyArray, compared to plain Fortran 77, was 7%.

The two versions of the handwritten C code (in the plain and clibcall directories) led to the same results. Also the plain C++ code, using the NumPyArray_Float class, and the version with a conversion class, utilizing MyArray, ran at the same speed. The SCXX-based version, however, was slower – in fact as much as 40%.

Using the various NumPy sin functions for scalar arguments inside the myfunc callback function slowed down the code by a factor of four compared to math.sin. The rule is to always use math.sin, or an alias for that function, if we know that the argument is a scalar.

Python callbacks from Fortran, C, or C++ are very expensive. The callback to Python inside the loops is so expensive that the rest of the compiled language code in a sense runs for free. The loop runs faster in compiled languages than in pure Python, but a factor of almost 40 is lost compared to the pure F77 code.

Table 10.1. Efficiency comparison of various implementations of the `gridloop1` and `gridloop2` functions in Python, Fortran 77, C, and C++.

language	function	`func1` argument	array tp.	time
F77	`gridloop1`	everything in F77 code		1.0
C++	`gridloop1`	everything in C++ code		1.07
Python	`__call__`	vectorized `myfunc`	`numpy`	1.5
Python	`__call__`	vectorized `myfunc`	`numarray`	2.7
Python	`__call__`	vectorized `myfunc`	`Numeric`	3.0
Python	`gridloop_itemset`	Py. `myfunc` (`math.sin`), Psyco	`numpy`	15
Python	`gridloop_itemset`	Py. `myfunc` (`math.sin`)	`numpy`	70
Python	`gridloop`	Py. `myfunc` (`math.sin`)	`numpy`	120
Python	`gridloop`	Py. `myfunc` (`Numeric.sin`)	`Numeric`	220
Python	`gridloop`	Py. `myfunc` (`numpy.sin`)	`numpy`	220
Python	`gridloop`	Py. `myfunc` (`numarray.sin`)	`numarray`	350
Python	`gridloop`	Py. `myfunc` (`math.sin`), Psyco	`numpy`	57
Python	`gridloop`	Py. `myfunc` (`math.sin`), Psyco	`Numeric`	80
F77	`gridloop1`	Py. `myfunc` (`math.sin`)	`numpy`	40
F77	`gridloop1`	Py. `myfunc` (`Numeric.sin`)	`Numeric`	160
F77	`gridloop1`	Py. `myfunc` (`numpy.sin`)	`numpy`	180
F77	`gridloop2`	Py. `myfunc` (`math.sin`)	`numpy`	40
F77	`gridloop_vec2`	vectorized Python `myfuncf2`	`numpy`	2.7
F77	`gridloop_vec2`	vectorized Python `myfuncf2`	`Numeric`	5.4
F77	`gridloop2_str`	F77 code	`numpy`	1.1
F77	`gridloop2_fcb`	F77 code	`numpy`	1.1
F77	`gridloop2_fcb_ptr`	F77 code	`numpy`	1.1
F77	`gridloop_noalloc`	F77 code, no `a` allocation	`numpy`	1.0
C	`gridloop2`	inline C code w/Instant	`numpy`	1.0
C	`gridloop1`	Py. `myfunc` (`math.sin`)	`numpy`	38
C	`gridloop2`	Py. `myfunc` (`math.sin`)	`numpy`	38
C	`gridloop1`	Py. `myfunc` (`Numeric.sin`)	`Numeric`	160
C	`gridloop1`	Py. `myfunc` (`numpy.sin`)	`numpy`	170
C++	`gridloop1`	Py. `myfunc` (`Numeric.sin`)	`Numeric`	160
C++	`gridloop1`	Py. `myfunc` (`math.sin`)	`numpy`	38
C++	`gridloop2`	Py. `myfunc` (`math.sin`)	`numpy`	38
C++	`ext_gridloop2_weave`	C++ code	`numpy`	1.4

The callback to a vectorized function, as explained in Chapter 9.4.1, has decent performance. Although a factor of almost four is lost, this might well be acceptable if the callback provides a convenient initialization of arrays prior to much more computationally intensive algorithms in Fortran subroutines. If a large number of callbacks is needed by a Fortran routine, high performance demands the callback function to be implemented in Fortran. Chapters 9.4.2 and 9.4.3 outline different strategies for letting a Fortran subroutine (`gridloop2`) invoke a callback function implemented in Fortran, whose content or name is flexibly set in the steering Python script. The different strategies lead to approximately the same performance. I find the most flexible strategy to be the one where the F77 callback function is compiled to an extension module by F2PY and we send the `_cpointer` attribute of the function in the module as callback argument to `gridloop2`. This technique of extracting the pointer to a Fortran function in Python also applies to C code if we use F2PY to wrap the C code. The other strategies explained for Fortran code can be used in a C and C++ context as well, see Exercises 10.4 and 10.5. In particular, when using Instant or Weave (Chapters 10.1.2 and 10.1.3) it is very easy to insert the expression of the callback function in the generated C/C++ code.

An important remark must be made. The programs written solely in Fortran or C++ allocate the `a` array only once, while our mixed Python-Fortran/C/C++ scripts calls the various compiled functions many times and the wrapper code allocates a new `a` array in each call. This extra allocation implies some overhead and explains why it is hard for the mixed language implementations to run at the same speed as the pure Fortran and C++ codes. To quantify the overhead, I made the `gridloop_noalloc` subroutine, which is identical to `gridloop2_str` but with `a` as `intent(in,out)` to avoid repeated allocation in the wrapper code (see also Chapters 9.3.3 and 12.3.6). This trick brought down the scaled CPU time from 1.1 to 1.0.

The example of filling an array with values from a Python function is simple to understand, and the implementation techniques cover many of the most important aspects of integrating Python and compiled code. The knowledge gained from this very simple case study is highly relevant for more complicated mathematical computations involving grids. For example, solving a two-dimensional partial differential equation on a uniform rectangular grid often leads to algorithms of the type (see Chapter 12.3.5)

```
for i in xrange(1,len(self.xcoor)-1):
    for j in xrange(1,len(self.ycoor)-1):
        x = self.xcoor[i];  y = self.ycoor[j]
        up[i,j] = u[i,j] + C*(u[i-1,j] + u[i+1,j] + u[i,j-1]
                             + u[i,j+1] - 4*u[i,j]) + D*f(x,y,t)
```

Here, `u` and `up` are NumPy arrays, and `f(x,y,t)` is a function. This loop, and even a vectorized version of it, may benefit significantly from migration to a compiled language. If `f` is defined in Python, we should use the aforementioned techniques to avoid calling the Python function inside the loop. However, in this case much more work is done inside the loop so the relative overhead of callbacks is smaller than in the examples with the `gridloop1` and `gridloop2` functions. The software associated with Chapter 12.3.6 illustrates and evaluates various techniques for implementing the loop above.

10.4.2 Error Handling

We have made a method `ext_gridloop_exception` in class `Grid2Deff` for testing how the extension module handles errors. The first call

```
ext_gridloop.gridloop1((1,2), self.xcoor, self.ycoor[1:], f)
```

sends a tuple as first argument and a third argument with wrong dimension. The Fortran wrappers automatically provide exception handling and issue the following exception in this case:

```
array_from_pyobj:intent(inout) argument must be an array.
```

That is, `gridloop1` expects an array, not a tuple as first argument.

The C code has partly manually inserted exception handling and partly built-in exceptions. An example of the latter is the `PyArg_ParseTuple` function, which raises exceptions if the supplied arguments are not correct. In our `gridloop1` call the function raises the exception

```
exceptions.TypeError gridloop1() argument 1 must be array, not tuple
```

The next erroneous call reads

```
ext_gridloop.gridloop1(self.xcoor, self.xcoor, self.ycoor[1:], f)
```

The first and third arguments have wrong dimensions. Fortran says

```
ext_gridloop.error failed in converting 1st argument
    'a' of ext_gridloop.gridloop1 to C/Fortran array
```

and C communicates our handwritten message

```
exceptions.ValueError a array is 1-dimensional or not of type float
```

The final test

```
ext_gridloop.gridloop2(self.xcoor, self.ycoor, 'abc')
```

has wrong type for the third argument. Fortran raises the exception

```
exceptions.TypeError ext_gridloop.gridloop2()
    argument 3 must be function, not str
```

and C gives the message

```
exceptions.TypeError func1 is not a callable function
```

These small tests involving wrong calls show that F2PY automatically builds quite robust interfaces.

10.4.3 Summary

It is time to summarize the experience with writing extension modules in Fortran, C, and C++.

- *Using F2PY, Instant, or Weave is easy.* These tools automates the process with creating extension modules such that the programmer can concentrate on just writing a function containing the loops to be migrated to compiled code. F2PY and Fortran is a very user-friendly combination, but has to be careful with input/output specification of arguments, and be prepared for changes (by F2PY) in the argument list on the Python side. F2PY is also very well suited for C code, but you either need to write the `.pyf` file yourself or let F2PY generate it from a Fortran 77 specification of the C functions' signatures. Instant is even easier to use than F2PY for inline C and C++ function in the Python code, but Instant is at this time of writing not so flexible in the types of input/output argument. Weave is also very easy to use and is a good choice if you want to program C++.
- *F2PY modules are robus wrt. erroneous arguments.* F2PY automatically generates consistency tests and associated exceptions. These were as comprehensive as our manually written tests in the C and C++ code.
- *Fortran and C/C++/NumPy/Python store multi-dimensional arrays differently.* An array made in C, C++, NumPy, or Python appears as transposed in Fortran. F2PY makes the problem with transposing multi-dimensional arrays transparent, at a cost of automatically generating copies of input arrays. This is usually not a problem if one follows the F2PY guidelines and carefully specifies input and output arguments. To write efficient and safe code, you need to understand how F2PY treats multi-dimensional arrays. In C and C++ modules, whether generated automatically by Instant or Weave, or written by hand, there is no storage incompatibility with Python.
- *C++ is more flexible and convenient than C.* One of the great advantages of C++ over C is the possibility to hide low level details of the Python and NumPy C API in new, more user-friendly data types. This makes C++ my language of choice for handwritten extension modules.
- *Callback to Python must be used with care.* F2PY automatically directs calls declared with `external` back to Python. Such callbacks degrade performance significantly if they are performed inside long loops. With F2PY one can implement the callback function in compiled code and grab a pointer to this function in Python and feed the pointer to another function in an extension module. We have also exemplified several alternative technqiues where the callback function is implemented in compiled code and where the user of the Python script can flexibly define the callback function.

10.5 Exercises

Exercise 10.1. Extend Exercise 5.2 or 5.3 with a callback to Python.

Modify the solution of Exercise 5.2 or 5.3 such that the function to be integrated is implemented in Python (i.e., perform a callback to Python) and transferred to the C or C++ code as a function argument. The simplest approach is to write the C or C++ wrapper code by hand. ◇

Exercise 10.2. Investigate the efficiency of vector operations.

A DAXPY[1] operation performs the calculation $u = ax + y$, where u, x, and y are vectors and a is a scalar. Implement the DAXPY operation in various ways:

– a plain Python loop over the vector indices,

– a NumPy vector expression `u = a*x + y`,

– a Fortran 77 subroutine with a `do` loop (called from Python).

Optionally, depending on your access to suitable software, you can test

– a Fortran 90 subroutine utilizing a vector expression `u = a*x + y`,

– a Matlab function utilizing a vector expression `u = a*x + y`,

– a Matlab function using a plain `for` loop over the vector indices,

– a C++ library that allows the vector syntax `u = a*x + y`.

Run m DAXPY operations with vector length n, such that $n = 2^{2k}$, $k = 1, \ldots, 11$, and $mn = \text{const}$ (i.e., the total CPU time is ideally the same for each test). Plot for each implementation the logarithm[2] of the (normalized) CPU time versus the logarithm of n. ◇

Exercise 10.3. Debug a C extension module.

The purpose of this exercise is to gain experience with debugging C extension modules by introducing errors in a working module and investigating the effect of each error. First make a copy of the `src/py/mixed/Grid2D/C/plain` directory. Then, for each of the errors below, edit the `gridloop.c` file, build the extension module, run the `Grid2Deff.py` script with command-line argument `verify1`, and observe the behavior of the execution. In the cases where the application fails with a "segmentation fault" or similar message, invoke a debugger (see Chapter 10.2.10) and find out exactly where the failure occurs. Here are some frequent errors to get experience with:

---

[1] The name DAXPY originates from the name of the subroutine in the standard BLAS library offering this computation.

[2] The smallest arrays will probably lead to a blow-up of the CPU time of the Python implementations, and that is why it might be convenient to use the logarithm of the CPU time.

1. remove the whole initialization function `initext_gridloop`,
2. remove the `import_array` call in `initext_gridloop`,
3. remove the `Py_InitModule3` call in `initext_gridloop`,
4. change the upper loop limits in `gridloop2` to `nx+1` and `ny+1`,
5. add a call to some function `mydebug` in `gridloop1`, but do not implement any `mydebug` function.

◇

Exercise 10.4. Make callbacks to vectorized Python functions.
Chapter 9.4.1 explains how to send arrays from F77 to a callback function in Python. Implement this idea in the `gridloop1` and `gridloop2` functions in the C or C++ extension modules. ◇

Exercise 10.5. Avoid Python callbacks in extension modules.
Chapter 9.4.2 explains how to avoid callbacks to Python in a Fortran setting. The purpose of this exercise is to implement the same idea in a C/C++ setting. Consider the extension module made in

```
src/py/mixed/Grid2D/C/clibcall
```

From Python we will call `gridloop1` and `gridloop2` with a string specification of the function to be evaluated at each grid point:

```
ext_gridloop.gridloop2(self.xcoor, self.ycoor, 'yourfunc')
```

Let the wrapper code test on the string value and supply the corresponding function pointer argument in the call to the `gridloop_C` function. What is the efficiency gain compared with the original code in the `clibcall` directory? ◇

Exercise 10.6. Extend Exercise 9.4 with C and C++ code.
Add a C implementation of the loop over the 3D array in Exercise 9.4 on page 480, using the `gridloop1` function as a template. Also add a C++ implementation using a class wrapper for NumPy arrays. ◇

Exercise 10.7. Apply SWIG to an array class in C++.
The purpose of this exercise is to wrap the `MyArray` class from Chapter 10.3.3 such that `MyArray` objects can be used in Python in almost the same way as they are used in C++. Use SWIG to generate wrapper code. ◇

Exercise 10.8. Build a dictionary in C.
Consider the following Python function[3]:

[3] This function builds a sparse matrix as a dictionary, based on connectivity information in a finite element grid [15]. For large grids the loops are long and a C implementation may improve the speed significantly.

```
def buildsparse(connectivity):
    smat = {}
    # connectivity is a NumPy array
    nel = connectivity.shape[0]
    nne = connectivity.shape[1]
    for e in range(nel):
        for r in range(nne):
            for s in range(r+1):
                i = connectivity[e, r]
                j = connectivity[e, s]
                smat[(i,j)] = 0.0
                smat[(j,i)] = 0.0
    return smat
```

Implement this function in C. You can use the script `src/misc/buildsparse.py` for testing both the function above and the C extension module (the script computes a sample `connectivity` array). Time the Python and C implementation when the loops are long. ◇

Exercise 10.9. Make a C module for computing random numbers.

The file `src/misc/draw.h` declares three functions in a small C library for drawing random numbers. The corresponding implementation of the functions is found in `src/misc/draw.c`. Make an extension module out of this C library and compare its efficiency with Python's `random` module. (Note: the modules apply different algorithms for computing random numbers so an efficiency comparison may not be completely fair.) ◇

Exercise 10.10. Almost automatic generation of C extension modules.

To simplify writing of C/C++ extension modules processing NumPy arrays, we could let a script generate much of the source code. The template from Chapter 10.2.9 is a good starting point for dumping code. Let the code generation script read a specification of the functions in the module. A suggested syntax for specifying a function may look like

```
fname; i:NumPy(dim1,dim2) v1; io:NumPy(dim1) v2; o:float v3; code
```

Such a line consists of fields separated by semi-colon. The first field, `fname`, is the name of the function. The next fields are the input and output arguments, where `i:` means input, `o:` output, and `io:` input and output. The variable type appears after the input/output prefix: `NumPy` for NumPy arrays, `int` for integer, `float` for floating-point numbers (`double` in C), `str` for strings (`char*` arrays in C), and `func` for callbacks. After the type specification we list the name of the variable. NumPy arrays have their dimensions enclosed in parenthesis, e.g., `v1` has associated C integers called `dim1` and `dim2` for holding its dimensions. The last field is the name of a file containing some core code of the function to be inserted before the return statement. If `code` is simply the word `none`, no such user-provided code exists.

Arguments specified as `o:` are returned, the others are taken as positional arguments in the same order as specified. Return `None` if there are no output arguments.

For each callback function the script should generate a skeleton with key statements in the callback, but the user is supposed to manually specify the argument list and process the result.

Consistency checks of actual array dimensions and those specified in the parenthesis proceeding `NumPy` must be generated.

For each function the script should generate a doc string with the call syntax as seen from Python. This string should also be a part of the module doc string.

As an example, the `gridloop2` function can be specified as

```
gridloop2; i:NumPy(nx) xcoor; i:NumPy(ny) ycoor; i:func func1;
          o:NumPy(nx,ny) a; gridloop2.c
```

(Everything is supposed to be on a single line. The line was broken here because of page width limitations.) The file `gridloop2.c` is supposed to contain the loop over the grid points, perhaps without the callback details. Since these details to some extent will be generated by the script, the user can move that code inside the loop and fill in the missing details.

The syntax of the function specifications is constructed such that a simple split with respect to semi-colon extracts the fields, a split with respect to white space distinguishes the type information from the variable name, and a split with respect to colon of the type information extracts the input/output specification. Extraction of array dimensions can be done by splitting the appropriate substring (`[6:-1]`) with respect to comma.

◇

Exercise 10.11. Introduce C++ array objects in Exercise 10.10.

Add an option to the script developed in Exercise 10.10 such that NumPy arrays can be wrapped in `NumPyArray_Float` objects from Chapter 10.3.1 to simplify programming. ◇

Exercise 10.12. Introduce SCXX in Exercise 10.11.

Modify the script from Exercise 10.10 to take advantage of the SCXX library for simplified programming with the Python C API. ◇