Lab 1

Computing with Cython

Lab Objective: *Use Cython to avoid the overhead of some parts of the Python language.*

Python is reasonably fast for computing the solutions to small problems. However, for large computations Python quickly becomes unable to solve anything in a reasonable amount of time. Why then is Python used if it's so slow? One important answer to this question is programmer productivity. The amount of time required to write code is dramatically less than the time required by low-level programming languages.

Low-level languages like C are renowned for their speed. These languages are compiled to machine code that is better adapted on a computer's processor. When a C program adds two numbers together, we have to be very specific about what two numbers are added together. The computer can understand basic data types such as integers, floats, and arrays. Python represents each of these data types using its own objects. These are objects are generally fast and we don't notice much the incurred overhead. Over large calculations, the computational cost of these objects is compounded and can slow things down tremendously. In these cases, we need to be able to access the machine-level data types for integers, floats, and strings for maximum efficiency. A common way of doing this is to write slower portions of the code in a lower-level, compiled language. This is usually accomplished by writing a C-extension for Python. A C-extension is a module that can be used like a Python module, but is written in C. Python is written in C, and it has well-defined ways to perform operations on Python objects from C. This enables performance critical code to be executed as fast as possible in a Python environment. Writing these C-extensions can often be difficult and tedious work.

Much of the time, the operations that are performed in these C extensions are direct analogs of simple operations from either Python or C, but the interface for working with both languages simultaneously often complicates what would otherwise be a series of simple operations. Cython was created to automate the creation of these C extension modules. As a language, **it is essentially just Python with some extra type declarations.** Using Cython, C-extensions can be written very quickly and give all the benefits of C speed. Cython also makes it easier to avoid bugs (like memory leaks) that often arise when writing C extensions.

Cython is like Python with added syntax that enables you to call C functions and declare C types on Python variables. The additional type declarations of Cython allow it to generate highly optimized C code that will compile with all major C/C++ compilers. Beginning users often use it to speed up loops and array access times. It is often used to create Python interfaces for C and C++ libraries and to write well-optimized Python libraries. Since Cython outputs C code, you can easily call any C function from any point in your Python program.

Cython is used extensively in many packages. SciPy uses Cython extensively in many of its packages for both improved speed and creating interfaces to packages that are written in C or C++. It also uses a tool called f2py to interface with Fortran (another fast programming language). Most core NumPy operations are written exclusively in C, but some of the components in its `random` module are written in Cython. Several other packages like pyfftw, pandas, scikit-learn, scikit-image, and astropy use Cython extensively. The package Kivy, that is used to write Android and iOS apps in Python, relies heavily on Cython. The computer algebra system Sage also relies very heavily on Cython to write fast code and interface between the many different packages it uses.

Warning

Don't expect Cython to speed up Python code automatically. If you run unmodified Python code through the Cython compiler, the result may be somewhat faster, but it may not. The benefits from Cython come when you change some portions of your code (like loops and operations that are performed inside loops) so that they are performed in C and no longer need to use generic Python objects.

Warning

Poorly written code will not perform well, regardless of the language in which it is written. It is often easier and more effective to focus on writing good code in Python than it is to try to interface with a different "faster" language. If you need to speed up your code, first profile it to see which parts are slow (IPython's magic function `%prun` can be used for this), then focus on removing unneeded operations from the slow portion of your code. Effective use of array operations in NumPy can often make code simpler and faster than it would have been if you had tried to write all the operations yourself in C, Fortran, or any other language. Cython should be used when other attempts at optimization and vectorization are insufficient or when you need to use functionality from a C or C++ library that is not available in Python.

Compilation

Cython has a more complex compilation process than Python and is not usually run interactively. Cython code is usually written in a `.pyx` file. Such a file is then

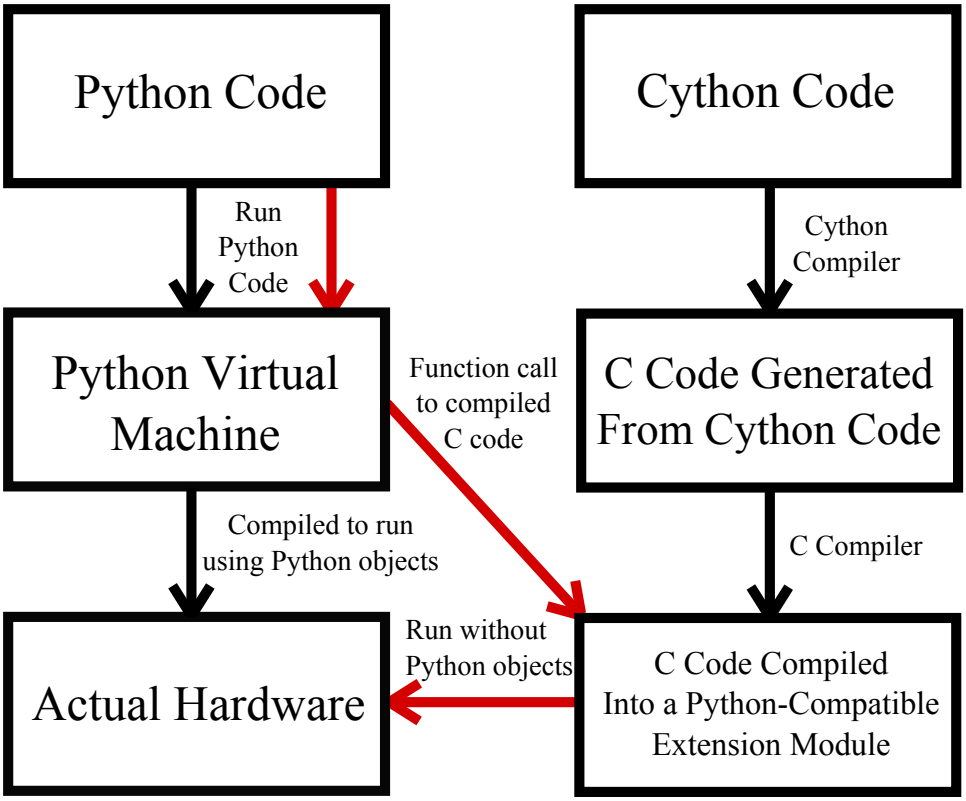

Figure 1.1: A diagram of how Cython compilation and calls to Cython functions work. The path from using a Cython function in a Python file to the actual evaluation is shown in red.

compiled to C, and then the C is compiled to machine code. The resulting file is a Python extension that can be imported from the Python interpreter. Many of the same built in features, data types, and functions from Python work in Cython, but keep in mind that too many calls to Python based functions may slow down a Cython program. Figure 1.1 shows how a Cython file is compiled and how a function call to a Cython module works. Regardless of what method you use to compile a Cython file, this is more or less how it works.

There are a variety of ways to import Cython functions and classes into Python. For example, consider the following cython file

```
1 def hello():
2     print "Hello from Cython!"
```

hello.pyx

We can compile this module using the following Python script:

```
1 from distutils.core import setup
2 from Cython.Build import cythonize
3
4 setup(name="hello", ext_modules=cythonize('hello.pyx'))
```

setup.py

This script should be used from the command prompt using the command:

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

After compiling the module, if the Python interpreter is running from the directory where the module was compiled, in Python the function can now be imported and run like any other Python function.

```
from hello import hello
hello()
```

For larger projects and pieces of code that need special compilation options, using some sort of setup script is the best option.

This code can be compiled and imported in the IPython Notebook all in one step using the Cython magic function. IPython must first import the corresponding functions so that you can use them. With older versions of IPython and Cython you can import this magic function by running the command

```
%load_ext cythonmagic
```

With newer versions of IPython and Cython, you should use the command

```
%load_ext Cython
```

Once this magic function has been loaded, you compile the the `hello` function and import it by running the cell

```
%%cython
def hello():
    print "Hello from Cython!"
```

The IPython magic function for Cython is good for compiling and experimenting with short snippets of code.

Note

When Cython generates the C file that is used to create the Python extension, it can show which portions of the code compile to Python operations and which portions compile to C operations. This is called code annotation. From the command line, if you were compiling `hello.pyx`, you could run the command

```
cython -a hello.pyx
```

This will generate a `html` file that you can open with your web browser. It will show the lines that use Python in bright yellow and lines that use C in white. Lines that use Python have a `+` to the left that expands the line of Cython code to show the relevant portion of C code that was generated.

The IPython Notebook magic can also show this information below a Cython-compiled cell. This is done by replacing `%%cython` with `%%cython -a`.

Type Declarations

One of the easiest ways improve performance in Cython is to declare types on variables. Python does not enforce strict type checking. Any variable can be any type and that type can change at any time. To allow this behavior, an extra interface layer is needed. Cython allows us to declare a variable of a certain type. These typed variables are not Python objects, but native machine-types.

In Python, you can iterate over a list, or use a generator object. In C, for-loops are managed by indexing an integer and checking to see if it is within a range of allowed values. This is a good example of how using Python objects for everything can slow things down. In Python a for-loop from 0 to 1000000 would look something like this:

```
for i in range(1000000):
    pass
```

This for-loop creates a Python list of 1000000 objects and then iterates through it. This can be very slow. It also uses a sizeable chunk of memory for no particular reason. We can improve on this by using the `xrange` function as follows:

```
for i in xrange(1000000):
    pass
```

Rather than making a list, `xrange` makes a generator object that returns the values as needed. For large lists, this can give be a few times faster. Cython will run this loop roughly the same way that Python would. If, however, we pre-define the type of the loop variable, Cython will change the loop so that it is run in C. A `cdef` statement can be used to define `i` as a C integer type with the syntax `cdef <type> <name>`. The empty for-loops above may be written in Cython as follows:

```
cdef int i
for i in range(1000000):
    pass
```

You can use a `cdef` statement to define multiple multiple variables of multiple types at once and initialize each of them to different values.

```
# Declare integer variables i, j, and k.
# Set k equal to 2.
cdef int i, j, k=2

cdef:
    # Declare and initialize m equal to 4 and n equal to 5.
    int m=4, n=5
    # Declare and initialize e equal to 2.71.
    double e = 2.71
    # Declare a double precision complex number a.
    double complex a
```

Cython Type	NumPy Type	Description
float	float32	32 bit floating point number
double	float64	64 bit floating point number
float complex	complex64	64 bit floating point complex number
double complex	complex128	128 bit floating point complex number
char	int8	8 bit signed integer
unsigned char	uint8	8 bit unsigned integer
short	int16	16 bit signed integer
unsigned short	uint16	16 bit unsigned integer
int	int32	32 bit signed integer
unsigned int	uint32	32 bit unsigned integer
long	int32 or int64	32 or 64 bit signed integer (size depends on platform)
unsigned long	uint32 or uint64	32 or 64 bit unsigned integer (size depends on platform)
long long	int64	64 bit signed integer
unsigned long long	uint64	64 bit unsigned integer

Table 1.1: Numeric types available in Cython.

Because Cython uses the extra type information, it can translate the loop into C, resulting in a speed-up of a factor of 50 over Python. Cython achieves this speed-up by removing the cost of working with Python integer objects. Similarly, when summing a large array of double precision floating point values, declaring the variable type will speed up the computation. It is much easier for a computer to perform arithmetic operations without having to infer what datatypes are being used when each operation is performed. Generally, whenever a large number of computations involve the same data type, adding type declarations should speed things up considerably.

Table 1.1 shows the different numeric types that are available in Cython. Cython uses the same type naming conventions as C and C++. On most normal modern systems, the sizes listed for integers will apply, though some are still dependent on the operating system.

Warning

Integer types in C are different from Python's integer type. Python's integer type works with arbitrarily large integers. C integers can overflow. When you declare the type of a variable in Cython to be some sort of integer, it is declared as a C integer type and can overflow.

You can also declare types for arrays. Declaring types for arrays allows some operations involving arrays to be performed *much* faster. The `__getitem__` method (equivalent to array access using `[ ]`) of NumPy arrays is written for Python and requires roughly the same cost as a Python function call. There are several ways to avoid this extra cost. Cython does include an interface for C arrays and pointers, but

the interface is not terribly convenient, so we will not discuss it here. Fortunately, Cython includes a C-level interface for Numpy-like array objects. Cython's arrays are called typed memoryviews.

A typed memoryview can be declared like this:

```
cdef double[:,:] X = ...
```

Declaring arrays in this manner allows us to access individual items in a NumPy array at roughly the same speed we could access items in a C array. Unfortunately, this fast array access only works when accessing *one item at a time* from the array. As of this writing, Cython's memoryviews only support accessing and writing to their entries. Any mathematical operations involving arrays must be performed by numpy or be run element-by-element in a loop that passes over the array. Memoryviews do allow you to pass slices of arrays from one Cython-defined function to another without the cost of a Python operation.

> **Note**
>
> Although typed memoryviews cannot be used directly for arithmetic, they can be passed as arguments to most numpy functions. Numpy includes functions for all common arithmetic operations, all of which allow the specification of an output array. For example, the operation `c = a + b` can be performed on typed memoryviews using the expression `c = np.add(a, b)` and the operation `c[:] = a * b` can be performed as `np.multiply(a, b, out=c)`. This does require the cost of calling a Python function, as well as all the checks that NumPy runs on all of the arguments passed to its functions, but, when working with large arrays, it can be worth it.

Compiler Directives

There are several compiler directives which may be passed to the Cython compiler to speed up array access. By default, Cython checks to see if the indices used in array accesses are within the bounds of the array. Cython also allows negative indexing the same way Python does. These features incur some performance loss and may be removed after a program has been carefully debugged. Compiler directives in Cython can be included as comments or as function decorators. Directives included in comments will apply to the whole file, while function decorators will only apply to the function or method immediately following the decorators. The comments to turn off bounds checking and negative indices are

```
# cython: boundscheck=False
# cython: wraparound=False
```

To use the function decorators, first import the `cython` module by including the line `cimport cython` in your import statements. The decorators are

```
cimport cython
```

```
@cython.boundscheck(False)
@cython.wraparound(False)
```

When using these compiler directives, make absolutely sure that your program is *not* using negative indices or accessing any array out of bounds. Doing so could access and possibly modify or access some portion of memory that has not been allocated as part of an array. That can either crash Python or cause unexpected behavior in a program. It is usually best to only include these directives after debugging a program thoroughly.

Cython has many other compiler directives. One to be aware of is the `cdivision` option. In Python, the `%` operator returns a number with the sign of the second argument, while C keeps the sign of the first argument. In Python, `-1%5` returns `4`, while in C, this returns `-1`. Cython, by default, will behave like Python. Cython will also check for zero division and raise a `ZeroDivisionError` when necessary. This feature does have some cost. If needed, you can make the `%` operator behave like it would in C and turn off the check for zero division by setting `cdivision` to `True`.

Functions in Cython

Type declarations can also be used for function arguments. You can declare the types of the inputs for the function to ensure that it receives the right arguments.

```
def myfunction(double[:] X, int n, double h, items):
    ...
```

Notice that the type declarations were not included for all of the arguments. The untyped arguments are expected to be Python objects with the corresponding methods. Computations involving these untyped arguments will use Python instead of C. Keyword arguments are allowed for both typed and untyped arguments.

Cython also allows you to make C functions that are only callable within the C extension you are currently building. These functions are declared using the same syntax as in Python, except the keyword `def` is replaced with `cdef`. These functions can be called within the module, but are not actually imported into the Python namespace of the Python module.

Cython functions may be declared using the `cpdef` statement. These functions are accesible within Cython without the cost of a Python function call, but can still be called from Python. The return type for functions declared using `cdef` and `cpdef` may also be specified, for example:

```
cpdef int myfunction(double[:] X, int n, double h, items):
    ...
```

Some Examples

To show how to use Cython, we will take the dot product of two one-dimensional arrays. This can be done in Python as follows:

```
def pydot(A, B):
    tot = 0.
    for i in xrange(A.shape[0]):
        tot += A[i] * b[i]
    return tot
```

A C-compiled equivalent can be made using the following Cython code

```
%%cython
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def cydot(double[:] A, double[:] B):
    cdef double tot=0.
    cdef int i
    for i in xrange(A.shape[0]):
        tot += A[i] * B[i]
    return tot
```

This function may be timed by evaluating the following cell in the IPython Notebook.

```
from numpy.random import rand
n = 10000000
A = rand(n)
B = rand(n)
%timeit cydot(A, B)
```

Figure 1.2 compares the timings of this new dot-product function with other possible implementations.

Now we will do a more advanced example using memoryviews. We will write functions which, given a two-dimensional array `A`, make a new array `B` such that `B[i,j] = dot(A[i],A[j])`.

Here's a purely iteration-based solution in Python:

```
def pyrowdot(A):
    B = np.empty((A.shape[0], A.shape[0]))
    for i in xrange(A.shape[0]):
        for j in xrange(i):
            temp = pydot(A[i], A[j])
            B[i,j] = temp
        B[i,i] = pydot(A[i], A[i])
    for i in xrange(A.shape[0]):
        for j in xrange(i+1, A.shape[0]):
            B[i,j] = B[j,i]
    return B
```

To do this same sort of thing in Cython we can compile the following file:

```
import numpy as np
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
```

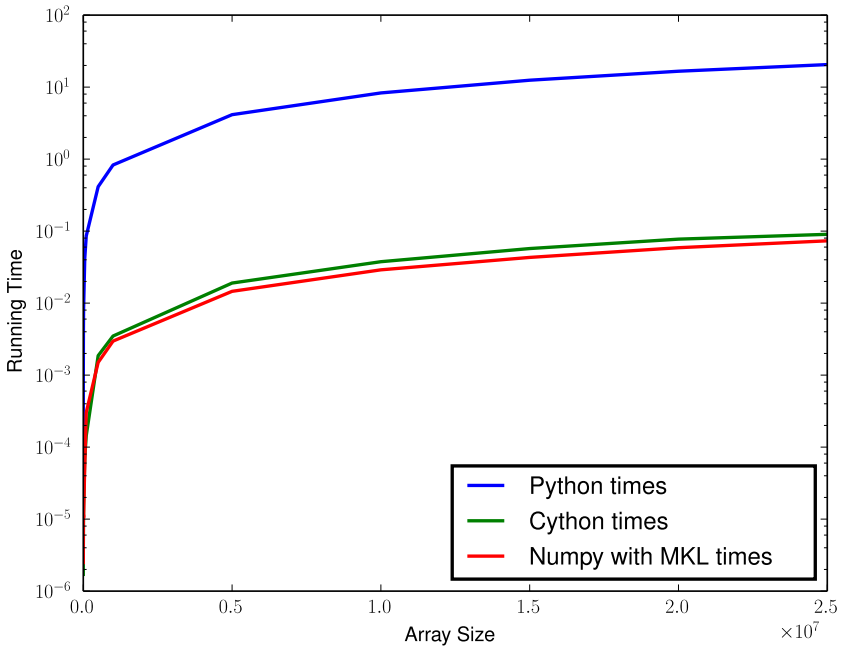

Figure 1.2: The running times of the pure Python dot-product, the Cython based dot-product, and the dot-product built into NumPy which links to the Intel MKL. The Cython version can run nearly as fast as the version built into NumPy.

```
cdef double cydot(double[:] A, double[:] B):
    cdef double tot=0.
    cdef int i, n=A.shape[0]
    for i in xrange(n):
        tot += A[i] * B[i]
    return tot

@cython.boundscheck(False)
@cython.wraparound(False)
def cyrowdot(double[:,:] A):
    cdef double[:,:] B = np.empty((A.shape[0], A.shape[0]))
    cdef double temp
    cdef int i, j, n=A.shape[0]
    for i in xrange(n):
        for j in xrange(i):
            temp = cydot(A[i], A[j])
            B[i,j] = temp
        B[i,i] = cydot(A[i], A[i])
    for i in xrange(n):
        for j in xrange(i+1, n):
            B[i,j] = B[j,i]
    return np.array(B)
```

This can also be done in NumPy by running `A.dot(A.T)`. The timings of the Python and Cython versions of this function are shown in Figure 1.3.

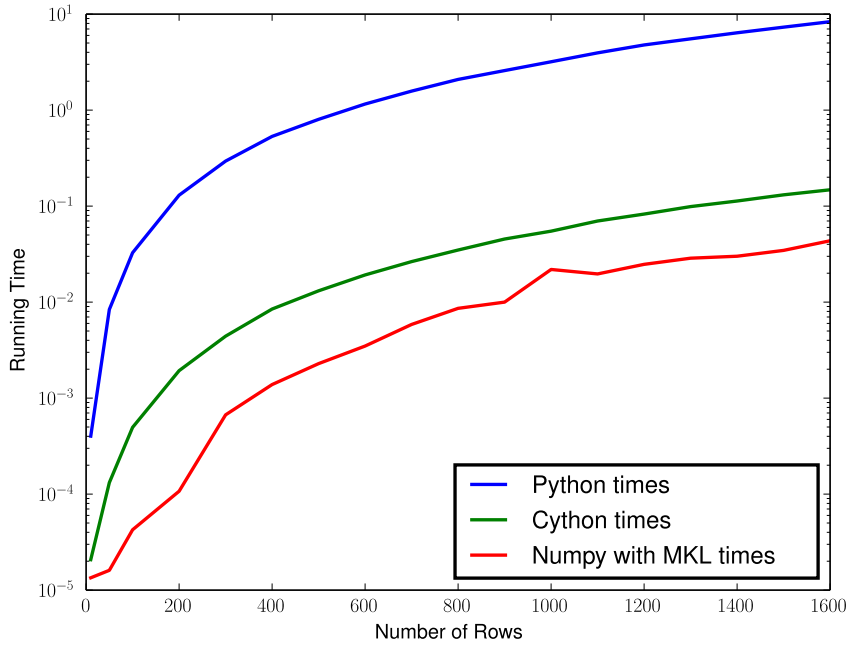

Figure 1.3: The times of the Python, Cython, and NumPy (with MKL) versions of the `rowdot` function that we used as an example. The arrays used for testing were $n \times 3$ where n is shown along the horizontal axis.

Problem 1. Write a function in Python that takes the sum of a one-dimensional array by iterating over it.

Write three Cython versions: one using a typed for-loop to iterate over the array, another using a typed for-loop and optimized array access, and another using a typed for-loop, optimized array access, and special compiler directives to further speed up array access.

Compare the speed of the functions you just wrote, the builtin `sum()` function and NumPy's `sum()` function. What do you see? Notice that when you specify the types for your variables in ways that don't work well, you may actually slow things down.

Problem 2. In an earlier lab, you wrote a function to compute the LU decomposition of a matrix. Write one version in pure Python that performs every operation element-by-element instead of using any sort of vector operation. Port your new loop-based version to Cython. Use typed for-loops, typed memoryviews, and the additional compiler directives in your optimized

solution. You may assume, in this case, that you are only dealing with real, two-dimensional arrays of double precision floating point numbers and that, as the computation proceeds, the diagonal elements will always be nonzero. Compare the speed of your new solutions to the speed of the vectorized Python version you wrote earlier.

Problem 3. (optional) The code below defines a Python function which takes a matrix to the nth power. Port it to Cython. Use typed for-loops, typed arrays, and the special compiler directives.

```
import numpy as np
def pymatpow(X, power):
    prod = X.copy()
    temparr = np.empty_like(X[0])
    size = X.shape[0]
    for n in xrange(1, power):
        for i in xrange(X.shape[0]):
            for j in xrange(X.shape[1]):
                tot = 0.
                for k in xrange(size):
                    tot += prod[i,k] * X[k,j]
                temparr[j] = tot
            prod[i] = temparr
    return prod
```

Compare the speed of the Python function, the function you just wrote, and the `np.dot()` function. The NumPy function should still be faster, but your solution should be much faster than the pure Python version. NumPy and SciPy do this computation and other computations by calling BLAS and LAPACK. BLAS and LAPACK are heavily optimized libraries for linear algebra. The BLAS are usually implemented using a mix of C and assembly, while LAPACK is written exclusively in Fortran. This is probably one of the best-optimized portions of NumPy and Scipy. The difference in performance should be particularly clear in this case because of the high order of complexity of the algorithm.

It is important to recognize that the choice of algorithm and fast implementation can both drastically affect the speed of your code. A simple example is the tridiagonal algorithm which is used to solve systems of the form:

$$\begin{bmatrix} b_0 & c_0 & 0 & 0 & 0 & \cdots & \cdots & 0 \\ a_0 & b_1 & c_1 & 0 & 0 & \cdots & \cdots & 0 \\ 0 & a_1 & b_2 & c_2 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & a_2 & b_3 & c_3 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & c_{n-1} \\ 0 & 0 & 0 & 0 & 0 & \cdots & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ \vdots \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_n \end{bmatrix}$$

The following Python code solves this system. (`x` and `c` are modified in place) The final result is stored in `x`, and `c` is used to store temporary values.

```
def pytridiag(a, b, c, x):
    # Note: overwrites c and x.
    n = x.size
    temp = 0.
    c[0] /= b[0]
    x[0] /= b[0]
    for i in xrange(n-2):
        temp = 1. / (b[i+1] - a[i] * c[i])
        c[i+1] *= temp
        x[i+1] = (x[i+1] - a[i] * x[i]) * temp
    x[n-1] = (x[n-1] - a[n-2] * x[n-2]) / (b[n-1] - a[n-2] * c[n-2])
    for i in xrange(n-2, -1, -1):
        x[i] = x[i] - c[i] * x[i+1]
```

Problem 4. Port the above code to Cython using typed for-loops, optimized array accesses and the special compiler directives. Compare the speed of your new function with the pure Python version given above. Now compare the speed of both of these functions with the `solve()` function in `scipy.linalg`. For your first two functions a good starting point for computation will be to consider 1000000×1000000 sized systems and then adjust the size so you get good results on your particular machine. When testing the SciPy algorithm, you will probably want to start with systems involving a 1000×1000 matrix and then go up from there. Keep in mind that the SciPy function is heavily optimized, but that it uses a much more general algorithm.

What does this example tell you about the relationship between good implementation and proper choice of algorithm? In this case, the LU decomposition method used by the SciPy function has complexity $O(n^3)$, while the tridiagonal algorithm is $O(n)$.

Note

Cython has predefined interfaces for many functions and classes in the C standard library. These functions can be imported using a `cimport` statement.

```
from libc.math cimport fabs, sin, cos, ...
```

Notice that we imported `fabs`. This is the absolute value function from C. The 'f' in `fabs` is short for floating point, since this function operates on floating point numbers. `min` and `max` are also renamed the same way.

Lab 2

Interfacing With Other Programming Languages Using Cython

Lab Objective: *Learn to interface with object files using Cython. This lab should be worked through on a machine that has already been configured to build Cython extensions using gcc or MinGW.*

Suppose you are writing a program in Python, but would like to call code written in another language. Perhaps this code has already been debugged and heavily optimized, so you do not want to simply re-implement the algorithm in Python or Cython. In technical terms, you want Python to *interface* (or communicate) with the other language. For example, NumPy's linear algebra functions call functions from LAPACK and BLAS, which are written in Fortran.

One way to have Python interface with C is to write a Cython *wrapper* for the C function. The wrapper is a Cython function that calls the C function. This is relatively easy to do because Cython compiles to C. From Python, you can call the wrapper, which calls the C function (see Figure TODO). In this lab you will learn how to use Cython to wrap a C function.

Wrapping with Cython: an overview

When you use Cython to wrap a C function, you just write a Cython function that calls the C function. To actually do this, we need to understand a little bit more about how C works.

After you write a program in C, you pass it to a compiler. The compiler turns your C code into machine code, i.e., instructions that your computer can read. The output of the compiler is an *object file*.

In our scenario, we have C code defining a single function that we would like to wrap in Cython. This C code has already been compiled to an object file. The protocol for calling the C code from Cython is similar to calling it from C:

1. we need to *include a header file* for the C code in our Cython code, and
2. we must *link to the object file* compiled from the C code when we compile the Cython code.

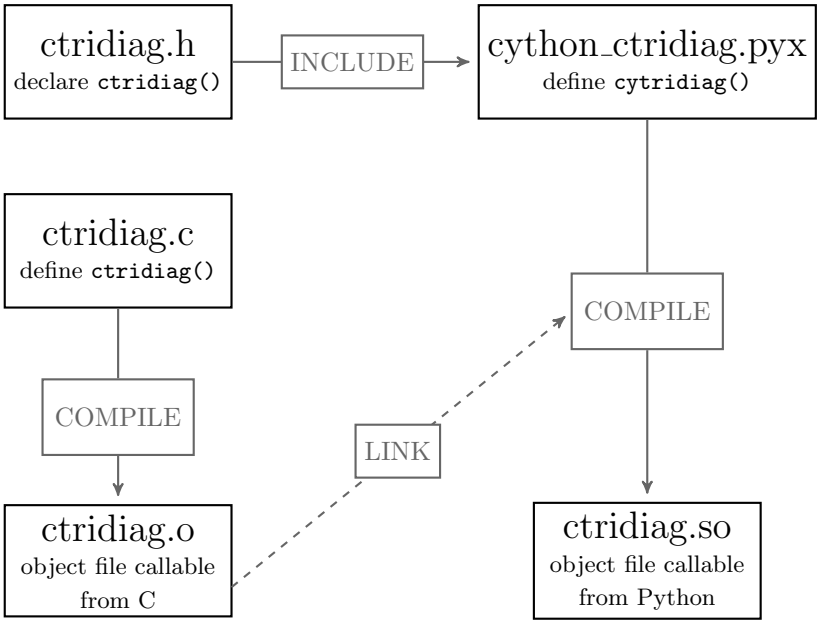

Figure 2.1: This diagram shows the relationships between the files used to write a Cython wrapper for `ctridiag()`. In fact, the Cython file `cython_ctridiag.pyx` compiles first to C code and then to the shared object file, but we have omitted this step from the diagram.

A *header* for a file contains the declaration of the function defined in the file. We include the header in the Cython code so that the compiler can check that the function is called with the right number and types of arguments. Then, when we tell the compiler to *link to* the object file, we simply tell it where to find the instructions defining the function declared in the header.

The Cython code will compile to a second object file. This object file defines a module that can be imported into Python. See Figure 2.1.

Wrapping with Cython: an example

As an example, we will wrap the C function `ctridiag()` below. This function computes the solution to a tridiagonal system $A\mathbf{v} = \mathbf{x}$. Its parameters are pointers to four 1-D arrays, which satisfy the following:

- Arrays `a` and `c` have length `n-1` and contain the first subdiagonal and superdiagonal of A, respectively.
- Arrays `b` and `x` have length `n` and represent the main diagonal of A and $\mathbf{x}$, respectively.

The array `c` is used to store temporary values and `x` is transformed into the solution of the system.

```
1 void ctridiag(double *a, double *b, double *c, double *x, int n){
```

```
    // Solve a tridiagonal system inplace.
    // Initialize temporary variable and index.
    int i;
    double temp;
    // Perform necessary computation in place.
    // Initial steps
    c[0] = c[0] / b[0];
    x[0] = x[0] / b[0];
    // Iterate down arrays.
    for (i=0; i<n-2; i++){
        temp = 1.0 / (b[i+1] - a[i] * c[i]);
        c[i+1] = c[i+1] * temp;
        x[i+1] = (x[i+1] - a[i] * x[i]) * temp;
        }
    // Perform last step.
    x[n-1] = (x[n-1] - a[n-2] * x[n-2]) / (b[n-1] - a[n-2] * c[n-2]);
    // Perform back substitution to finish constructing the solution.
    for (i=n-2; i>-1; i--){
        x[i] = x[i] - c[i] * x[i+1];
        }
    }
```

ctridiag.c

This terminal command tells the C-compiler gcc to compiles `ctridiag.c` to an object file `ctridiag.o`.

```
>>> gcc -fPIC -c ctridiag.c -o ctridiag.o
```

The `-fPIC` option is required because we will later link to this object file when compiling a shared object file. The `-c` flag prevents the compiler from raising an error even though `ctridiag.c` does not have a `main` function.

Write a header for `ctridiag.c`

The header file essentially contains a function declaration for `ctridiag()`. It tells Cython how to use the object file `ctridiag.o`.

```
extern void ctridiag(double* a, double* b, double* c, double* x, int n);
// This declaration specifies what arguments should be passed
// to the function called `ctridiag.'
```

ctridiag.h

Write a Cython wrapper

Next we write a Cython file containing a function that "wraps" `ctridiag()`. This file must include the header we wrote in the previous step.

```
# Include the ctridiag() function as declared in ctridiag.h
# We use a cdef since this function will only be called from Cython
cdef extern from "ctridiag.h":
    void ctridiag(double* a, double* b, double* c, double* x, int n)

# Define a Cython wrapper for ctridiag().
```

```
# Accept four NumPy arrays of doubles
# This will not check for out of bounds memory accesses.
cpdef cytridiag(double[:] a, double[:] b, double[:] c, double[:] x):
    cdef int n = x.size
    ctridiag(&a[0], &b[0], &c[0], &x[0], n)
```

cython_ctridiag.pyx

Some comments on this code are in order. First, including the header for `ctridiag()` allows us to call this function even though it was not defined in this file. This is a little like importing NumPy at the start of your Python script.

Second, the arguments of the Cython function `cytridiag()` are not in bijection with the arguments of `ctridiag()`. In fact, Cython does not need to know the size of the NumPy arrays it accepts as inputs because these arrays are objects that carry that information with them. We extract this information and pass it to the C function inside the Cython wrapper.

However, it is possible to unintentionally access memory outside of an array by calling this function. For example, suppose the input arrays `a`, `b`, `c`, and `x` are of sizes 4, 5, 3, and 5. Since `x` is used to determine the size of the system, the function `ctridiag()` will expect `c` to have length 4. At some point, `ctridiag()` will likely try to read or even write to the 4th entry of `c`. This address does exist in memory, but it does not contain an entry of `c`! Therefore, this function must be called by a responsible user who knows the sizes of her arrays. Alternatively, you could check the sizes of the arrays in the Cython wrapper before passing them to the C function.

Finally, the C function expects a parameter `double* a`, meaning that `a` is a *pointer to* (i.e., the address of) a `double`. The funtion `ctridiag()` expects this double to be the first entry of the array `a`. So instead of passing the object `a` to `ctridiag()`, we find the first entry of `a` with `a[0]`, and then take its address with the `&` operator.

Compile the Cython wrapper

Now we can compile `cython_ctridiag.pyx` to build the Python extension. The following setup file uses distutils to compile the Cython file, and may be run on Windows, Linux, or Macintosh-based computers. Notice that in line 28, we link to the existing object file `ctridiag.o`.

```
# Import needed setup functions.
from distutils.core import setup
from distutils.extension import Extension
# This is part of Cython's interface for distutils.
from Cython.Distutils import build_ext
# We still need to include the directory
# containing the NumPy headers.
from numpy import get_include
# We still need to run one command via command line.
from os import system

# Compile the .o file we will be accessing.
# This is independent building the Python extension module.
shared_obj = "gcc ctridiag.c -fPIC -c -o ctridiag.o"
print shared_obj
system(shared_obj)
```

```
# Tell Python how to compile the extension.
ext_modules = [Extension(
                         # Module name:
                         "cython_ctridiag",
                         # Cython source file:
                         ["cython_ctridiag.pyx"],
                         # Other compile arguments
                         # This flag doesn't do much this time,
                         # but this is where it would go.
                         extra_compile_args=["-fPIC", "-O3"],
                         # Extra files to link to:
                         extra_link_args=["ctridiag.o"])]

# Build the extension.
setup(name = 'cython_ctridiag',
      cmdclass = {'build_ext': build_ext},
      # Include the directory with the NumPy headers when compiling.
      include_dirs = [get_include()],
      ext_modules = ext_modules)
```

ctridiag_setup_distutils.py

This setup file can be called from the command line with the following command.

```
python ctridiag_setup_distutils.py build_ext --inplace
```

The `--inplace` flag tells the script to compile the extension in the current directory. The appendix at the end of this lab contains setup files that build the Python extension by hand on various operating systems.

Test the Python extension

After running the setup file, you should have a Python module called `cython_cytridiag` that defines a function `cytridiag()`. You can import this module into IPython as usual. However, if you modify `ctridiag()` and then try to recompile the extension, you may get an error if the module is currently in use. Hence, if you are frequently recompiling your extension, it is wise to test it with a script.

The following script tests the module `cython_cytridiag`.

```
import numpy as np
from numpy.random import rand
# Import the new module.
from cython_ctridiag import cytridiag as ct

# Construct arguments to pass to the function.
n=10
a, b, c, x = rand(n-1), rand(n), rand(n-1), rand(n)
# Construct a matrix A to test that the result is correct.
A = np.zeros((n,n))
A.ravel()[A.shape[1]::A.shape[1]+1] = a
A.ravel()[::A.shape[1]+1] = b
A.ravel()[1::A.shape[1]+1] = c
# Store x so we can verify the algorithm returned the correct values
x_copy = x.copy()
# Call the function.
ct(a, b, c, x)
```

```
# Test to see if the result is correct.
# Print the result in the command line.
if np.absolute(A.dot(x) - x_copy).max() < 1E-12:
    print "Test Passed"
else:
    print "Test Failed"
```

ctridiag_test.py

Warning

When passing arrays as pointers to C or Fortran functions, be *absolutely sure* that the array being passed is contiguous. This means that the entries of the array are stored in *adjacent* entries in memory. Passing one of these functions a strided array will result in out of bounds memory accesses and could crash your computer. For an example of how to check an array is contiguous in a Cython wrapper, see Problem 1.

Another example

The C function `cssor()` below implements the Successive Over Relaxation algorithm for solving Laplace's equation, which in two dimensions is

$$\frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} = 0.$$

In the Successive Over Relaxation algorithm, an input array is iteratively modified until it converges to the solution $F(x, y)$.

```
void cssor(double* U, int m, int n, double omega, double tol, int maxiters, int* info){
    // info is passed as a pointer so the function can modify it as needed.
    // Temporary variables:
    // 'maxerr' is a temporary value.
    // It is used to determine when to stop iteration.
    // 'i', 'j', and 'k' are indices for the loops.
    // lcf and rcf will be precomputed values
    // used on the inside of the loops.
    double maxerr, temp, lcf, rcf;
    int i, j, k;
    lcf = 1.0 - omega;
    rcf = 0.25 * omega;
    for (k=0; k<maxiters; k++){
        maxerr = 0.0;
        for (j=1; j<n-1; j++){
            for (i=1; i<m-1; i++){
                temp = U[i*n+j];
                U[i*n+j] = lcf * U[i*n+j] + rcf * (U[i*n+j-1] + U[i*n+j+1] + U[(i-1)*n+j] + U[(i+1)*n+j]);
                maxerr = fmax(fabs(U[i*n+j] - temp), maxerr);}}
        // Break the outer loop if within
        // the desired tolerance.
        if (maxerr < tol){break;}}
```

```
    // Here we have it set status to 0 if
    // the desired tolerance was attained
    // within the the given maximum
    // number of iterations.
    if (maxerr < tol){*info=0;}
    else{*info=1;}}
```

cssor.c

The function `cssor()` accepts the following inputs.

- U is a pointer to a Fortran-contiguous 2-D array of double precision numbers.
- m is an integer storing the number of rows of U.
- n is an integer storing the number of columns of U.
- ω is a double precision floating point value between 1 and 2. The closer this value is to 2, the faster the algorithm will converge, but if it is too close the algorithm may not converge at all. For this lab just use 1.9.
- tol is a floating point number storing a tolerance used to determine when the algorithm should terminate.
- maxiters is an integer storing the maximum allowable number of iterations.
- info is a pointer to an integer. This function will set info to 0 if the algorithm converged to a solution within the given tolerance. It will set info to 1 if it did not.

Problem 1. Wrap `cssor()` so it can be called from Python, following these steps.

1. Write a C header for `cssor.c`.
2. Write a Cython wrapper `cyssor()` for the function `cssor()` as follows.
   (a) Have `cyssor()` accept parameters `tol` and `maxiters` that default to $10e^{-8}$ and 10,000, respectively. What other arguments does `cyssor()` need to accept?
   (b) Check that the input `U` is a C-contiguous array (this means that the array is stored in a certain way in memory). You can check if `U` is C-contiguous with the code `U.is_c_contig()`, which returns a Boolean value. If `U` is not C-contiguous, raise a ValueError as follows:

```
raise ValueError('Input array U is not C-contiguous')
```

   When raising a ValueError, you may replace the string `'Input array U is not C-contiguous'` with any desired text. For more about errors and exceptions in Python, see `https://docs.python.org/2/tutorial/errors.html`.

(c) Raise a ValueError if `cssor()` fails to converge—i.e., if `cssor()` sets `info` to 1.

3. Compile `cssor.c` to an object file and compile your Cython wrapper to a Python extension. Be aware that you may compile `cssor.c` correctly and still receive warnings from the compiler.

4. Write a test script for your Cython function.

   (a) You can run the function with the following code.

```
import numpy as np
from cython_fssor import cyssor
resolution = 501
U = np.zeros((resolution, resolution))
X = np.linspace(0, 1, resolution)
U[0] = np.sin(2 * np.pi * X)
U[-1] = - U[0]
cyssor(U, 1.9)
```

   (b) Have your test script run the code above and plot the modified array `U` on the domain $[0,1] \times [0,1]$. Note that U is a 501×501 array. Your plot should look like Figure 2.2.

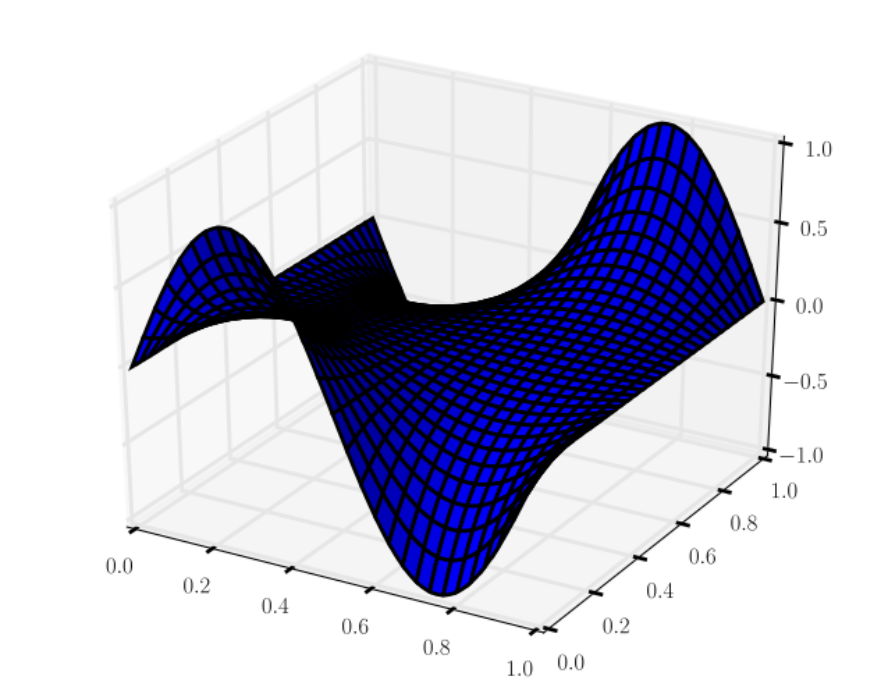

Figure 2.2: Correct output for the test script of Problem 1.

Wrapping a Fortran function (Optional)

We can also use Cython to wrap Fortran. As an example, we will wrap the Fortran function below, which implements the same algorithm as `ctridiag()`.

We have used the C library `iso_c_binding` to make the function accept pointers to native C types. If we were wrapping a function or subroutine that we did not write ourselves, we would have to define a Fortran function that uses the `iso_c_binding` library to accept pointers from C and then uses the values it receives to call the original function.

```
module tridiag

use iso_c_binding, only: c_double, c_int

implicit none

contains

! The expression bind(c) tells the compiler to
! make the naming convention in the object file
! match the naming convention here.
! This will be a subroutine since it does not
! return any values.
subroutine ftridiag(a, b, c, x, n) bind(c)

!   Here we declare the types for the inputs.
!   Thisis where we use the c_double and c_int types.
!   The 'dimension' statement tells the compiler that
!   the argument is an array of the given shape.
    integer(c_int), intent(in) :: n
    real(c_double),dimension(n),intent(in) :: b
    real(c_double),dimension(n),intent(inout) :: x
    real(c_double),dimension(n-1),intent(in) :: a
    real(c_double),dimension(n-1),intent(inout) :: c

!   Two temporary varaibles.
!   'm' is a temporary value.
!   'i' is the index for the loops.
    real(c_double) m
    integer i

!   Here is the actual computation:
    c(1) = c(1) / b(1)
    x(1) = x(1) / b(1)
!   This is the syntax for a 'for' loop in Fortran.
!   Indexing for arrays in fortran starts at 1
!   instead of starting at 0 like it does in Python.
!   Arrays are accessed using parentheses
!   instead of brackets.
    do i = 1,n-2
        m = 1.0D0 / (b(i+1) - a(i) * c(i))
        c(i+1) = c(i+1) * m
        x(i+1) = x(i+1) - a(i) * x(i)
        x(i+1) = x(i+1) * m
!   Note that you have to explicitly end the loop.
    enddo
    x(n) = (x(n) - a(n-1) * x(n-1)) / (b(n) - a(n-1) * c(n-1))
    do i = n-1,1,-1
```

```
        x(i) = x(i) - c(i) * x(i+1)
    enddo
!   You must also explicitly end the function or subroutine.
end subroutine ftridiag

end module
```

ftridiag.f90

Warning

When interfacing between Fortran and C, you will have to pass pointers to *all* the variables you send to the Fortran function as arguments. Passing a variable directly will probably crash Python.

Write a header for `ftridiag.f90`

Here is a header that tells C how to interface with the function we have just defined.

```
extern void ftridiag(double* a, double* b, double* c, double* x, int* n);
// Notice that we passed n as a pointer as well.
```

ftridiag.h

To compile `ftridiag.f90` to an object file you can run the following command in your command line:

```
gfortran -fPIC -c ftridiag.f90 -o ftridiag.o
```

Write a Cython wrapper, compile, and test it

The Cython wrapper for this function is analogous to `cython_ctridiag.pyx`, except that every variable passed to `ftridiag` should be a pointer. Aside from the use of gfortran instead of gcc, the rest of the compilation and testing process is entirely the same. The setup files specific to Windows and Linux, the setup file using distutils, and the test script are included in this lab folder.

Warning

Fortran differs from C in that the columns of arrays are stored in contiguous blocks of memory instead of the rows. The default for NumPy and for C arrays is to have rows stored in contiguous blocks, but in NumPy this varies. You should make sure any code passing arrays between Fortran, C, and NumPy addresses this discrepancy. Though by default Fortran assumes arrays are arranged in Fortran-order, for many routines you can specify whether to act on an array or its transpose.

> **Note**
>
> The function `ftridiag()` may seem unusual because it is a Fortran function that is callable from C. In fact, this is not an uncommon scenario. Many larger Fortran libraries (for example, LAPACK) provide C wrappers for all their functions.

Another example

Here is a Fortran implementation of the Successive Over Relaxation algorithm for solving Laplace's equation. This function is the Fortran equivalent of `cssor.c`. Its parameters are the same, with the exception that all parameters are *points* to the values they reference.

```
module ssor

use iso_c_binding, only: c_double, c_int

implicit none

contains

! The expression bind(c) tells the compiler to
! make the naming convention in the object file
! match the naming convention here.
! This will be a subroutine since it does not
! return any values.
subroutine fssor(U, m, n, omega, tol, maxiters, info) bind(c)

!   Here we declare the types for the inputs.
!   This is where we use the c_double and c_int types.
!   The 'dimension' statement tells the compiler that
!   the argument is an array of the given shape.
    integer(c_int), intent(in) :: m, n, maxiters
    integer(c_int), intent(out) :: info
    real(c_double), dimension(m,n), intent(inout) :: U
    real(c_double), intent(in) :: omega, tol

!   Temporary variables:
!   'maxerr' is a temporary value.
!   It is used to determine when to stop iteration.
!   'i', 'j', and 'k' are indices for the loops.
!   lcf and rcf will be precomputed values
!   used on the inside of the loops.
    real(c_double) :: maxerr, temp, lcf, rcf
    integer i, j, k

    lcf = 1.0D0 - omega
    rcf = 0.25D0 * omega
    do k = 1, maxiters
        maxerr = 0.0D0
        do j = 2, n-1
            do i = 2, m-1
                temp = U(i,j)
```

```
                U(i,j) = lcf * U(i,j) + rcf * (U(i,j-1) + U(i,j+1) + U(i-1,j) + U↩
                     (i+1,j))
42              maxerr = max(abs(U(i,j) - temp), maxerr)
            enddo
44      enddo
!       Break the outer loop if within
46 !    the desired tolerance.
        if (maxerr < tol) exit
48  enddo
!   Here we have it set status to 0 if
50 !   the desired tolerance was attained
!   within the the given maximum
52 !   number of iterations.
    if (maxerr < tol) then
54      info = 0
    else
56      info = 1
    end if
58 end subroutine fssor

60 end module
```

fssor.f90

Problem 2. Imitate Problem 1 to wrap `fssor()` with Cython.

Appendix: compiling C extensions for Python

If you know more about compilers, you may find it enlightening to manually compile a C extension for Python. Here we show how to manually compile `cython_ctridiag.pyx` on Windows, Linux, and Macintosh machines.

On Windows, using the compiler MinGW (a version of gcc for windows), the compilation can be performed by running the following setup file.

```
1  # The system function is used to run commands
2  # as if they were run in the terminal.
   from os import system
4  # Get the directory for the NumPy header files.
   from numpy import get_include
6  # Get the directory for the Python header files.
   from distutils.sysconfig import get_python_inc
8  # Get the directory for the general Python installation.
   from sys import prefix
10
   # Now we construct a string for the flags we need to pass to
12 # the compiler in order to give it access to the proper
   # header files and to allow it to link to the proper
14 # object files.
   # Use the -I flag to include the directory containing
16 # the Python headers for C extensions.
   # This header is needed when compiling Cython-made C files.
18 # This flag will look something like: -Ic:/Python27/include
   ic = " -I" + get_python_inc()
```

```
# Use the -I flag to include the directory for the NumPy
# headers that allow C to interface with NumPy arrays.
# This is necessary since we want the Cython file to operate
# on NumPy arrays.
npy = " -I" + get_include()
# This links to a compiled Python library.
# This flag is only necessary on Windows.
lb = " \"" + prefix + "/libs/libpython27.a\""
# This links to the actual object file itself
o = " \"ctridiag.o\""

# Make a string of all these flags together.
# Add another flag that tells the compiler to
# build for 64 bit Windows.
flags = "-fPIC" + ic + npy + lb + o + " -D MS_WIN64"

# Build the object file from the C source code.
system("gcc ctridiag.c -fPIC -c -o ctridiag.o")
# Compile the Cython file to C code.
system("cython -a cython_ctridiag.pyx --embed")
# Compile the C code generated by Cython to an object file.
system("gcc -c cython_ctridiag.c" + flags)
# Make a Python extension containing the compiled
# object file from the Cython C code.
system("gcc -shared cython_ctridiag.o -o cython_ctridiag.pyd" + flags)
```

ctridiag_setup_windows64.py

The following file works on Linux and Macintosh machines using gcc.

```
# The system function is used to run commands
# as if they were run in the terminal.
from os import system
# Get the directory for the NumPy header files.
from numpy import get_include
# Get the directory for the Python header files.
from distutils.sysconfig import get_python_inc

# Now we construct a string for the flags we need to pass to
# the compiler in order to give it access to the proper
# header files and to allow it to link to the proper
# object files.
# Use the -I flag to include the directory containing
# the Python headers for C extensions.
# This header is needed when compiling Cython-made C files.
ic = " -I" + get_python_inc()
# Use the -I flag to include the directory for the NumPy
# headers that allow C to interface with NumPy arrays.
# This is necessary since we want the Cython file to operate
# on NumPy arrays.
npy = " -I" + get_include()
# This links to the actual object file itself
o = " ctridiag.o"

# Make a string of all these flags together.
# Add another flag that tells the compiler to make
# position independent code.
flags = ic + npy + o + " -fPIC"

# Build the object file from the C source code.
```

```
system("gcc ctridiag.c -c -o ctridiag.o -fPIC")
# Compile the Cython file to C code.
system("cython -a cython_ctridiag.pyx --embed")
# Compile the C code generated by Cython to an object file.
system("gcc -c cython_ctridiag.c" + flags)
# Make a Python extension containing the compiled
# object file from the Cython C code.
system("gcc -shared cython_ctridiag.o -o cython_ctridiag.so" + flags)
```

ctridiag_setup_linux.py