Lab 19

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.

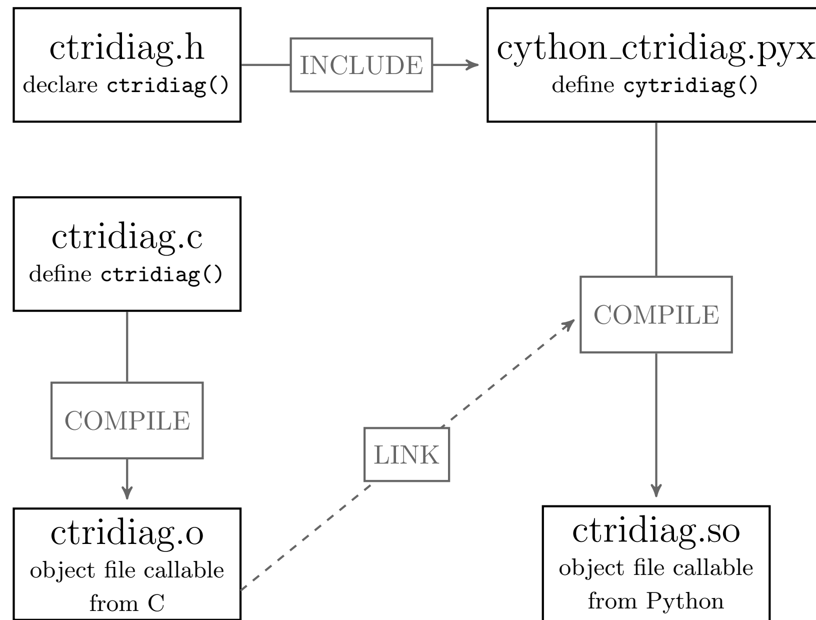

Figure 19.1: This diagram shows the relationships between the files used to write a Cython wrapper for `ctridiag()`. In fact, the Cython file `cython_ctridiag.pyx` compiles first to C code and then to the shared object file, but we have omitted this step from the diagram.

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 19.1.

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;}}
```

```c
    // 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:

```python
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 19.2.

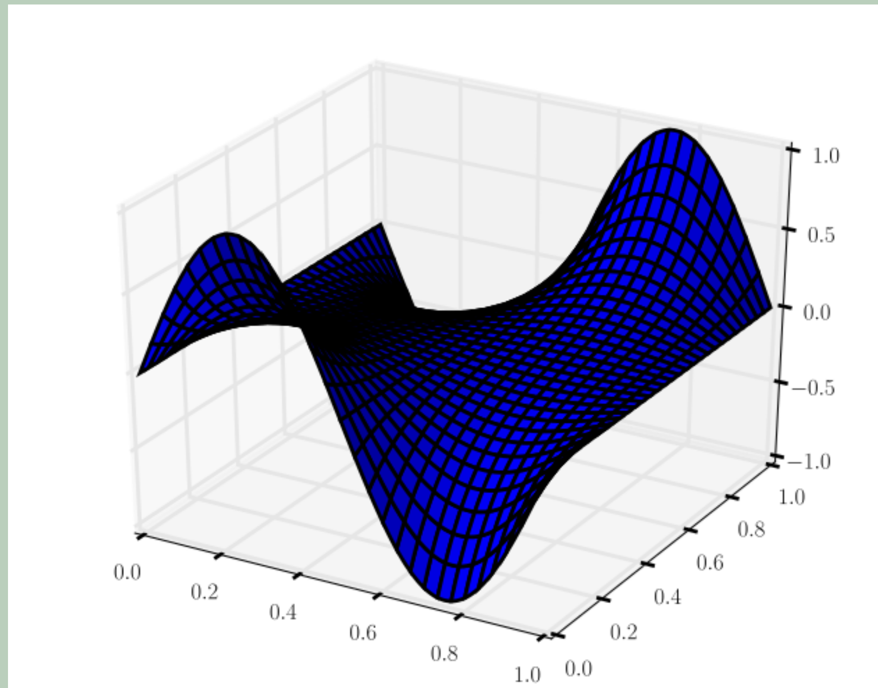

Figure 19.2: Correct output for the test script of Problem 1.

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))
                maxerr = max(abs(U(i,j) - temp), maxerr)
            enddo
        enddo
!       Break the outer loop if within
!       the desired tolerance.
        if (maxerr < tol) exit
    enddo
!   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) then
        info = 0
    else
        info = 1
    end if
end subroutine fssor

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.

```
# 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
# Get the directory for the general Python installation.
from sys import prefix

# 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.
# 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