Lab 15

Profiling Python Code

Lab Objective: *Learn how to efficiently optimize Python code.*

The best code goes through multiple drafts. In a first draft, you should focus on writing code that does what is is supposed to and is easy to read. After writing a fist draft, you may find that your code does not run as quickly as you need it to. Then it is time to *optimize* the most time consuming parts of your code so that they run as quickly as possible.

In this lab we will optimize the function `qr1()` that computes the QR decomposition of a matrix via the modified Gram-Schmidt algorithm (see Lab 7).

```
import numpy as np
from scipy import linalg as la

def qr1(A):
    ncols = A.shape[1]
    Q = A.copy()
    R = np.zeros((ncols, ncols))
    for i in range(ncols):
        R[i, i] = la.norm(Q[:, i])
        Q[:, i] = Q[:, i]/la.norm(Q[:, i])
        for j in range(i+1, ncols):
            R[i, j] = Q[:, j].dot(Q[:, i])
            Q[:,j] = Q[:,j]-Q[:, j].dot(Q[:, i])*Q[:,i]
    return Q, R
```

What to optimize

Python provides a *profiler* that can identify where code spends most of its runtime. The output of the profiler will tell you where to begin your optimization efforts.

In IPython[1], you can profile a function from the command line with `%prun`. Here we profile `qr1()` on a random 300×300 array.

[1]If you are not using IPython, you will need to use the `cProfile` module documented here: `https://docs.python.org/2/library/profile.html`.

```
>>> A = np.random.rand(300, 300)
>>> \%prun qr1(A)
```

On this computer, we get the following output.

```
     97206 function calls in 1.343 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.998    0.998    1.342    1.342 profiling_hw.py:4(qr1)
 89700    0.319    0.000    0.319    0.000 {method 'dot' of 'numpy.ndarray' objects}
   600    0.006    0.000    0.012    0.000 function_base.py:526(asarray_chkfinite)
   600    0.006    0.000    0.009    0.000 linalg.py:1840(norm)
  1200    0.005    0.000    0.005    0.000 {method 'any' of 'numpy.ndarray' objects}
   600    0.002    0.000    0.002    0.000 {method 'reduce' of 'numpy.ufunc' objects}
  1200    0.001    0.000    0.001    0.000 {numpy.core.multiarray.array}
  1200    0.001    0.000    0.002    0.000 numeric.py:167(asarray)
     1    0.001    0.001    0.001    0.001 {method 'copy' of 'numpy.ndarray' objects}
   600    0.001    0.000    0.022    0.000 misc.py:7(norm)
   301    0.001    0.000    0.001    0.000 {range}
     1    0.001    0.001    0.001    0.001 {numpy.core.multiarray.zeros}
   600    0.001    0.000    0.001    0.000 {method 'ravel' of 'numpy.ndarray' objects}
   600    0.000    0.000    0.000    0.000 {method 'conj' of 'numpy.ndarray' objects}
     1    0.000    0.000    1.343    1.343 <string>:1(<module>)
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

The first line of the output tells us that executing `qr1(A)` results in almost 100,000 function calls. Then we see a table listing these functions along with data telling us how much time each takes. Here, `ncalls` is the number of calls to the function, `tottime` is the total time spent in the function, and `cumtime` is the amount of time spent in the function including calls to other functions.

For example, the first line of the table is the function `qr1(A)` itself. This function was called once, it took 1.342s to run, and 0.344s of that was spent in calls to other functions. Of that 0.344s, there were 0.319s spent on 89,700 calls to `np.dot()`.

With this output, we see that most time is spent in multiplying matrices. Since we cannot write a faster method to do this multiplication, we may want to try to reduce the number of matrix multiplications we perform.

How to Optimize

Once you have identified those parts of your code that take the most time, how do you make them run faster? This section lists a few ideas. Always, you should use the profiling and timing functions to help you decide when an optimization is actually useful.

Avoid recomputing values

In our function `qr1()`, we can avoid recomputing `R[i,i]` in the outer loop and `R[i,j]` in the inner loop. The rewritten function is as follows:

```
def qr2(A):
    ncols = A.shape[1]
    Q = A.copy()
    R = np.zeros((ncols, ncols))
```

```python
    for i in range(ncols):
        R[i, i] = la.norm(Q[:, i])
        Q[:, i] = Q[:, i]/R[i, i]
        for j in range(i+1, ncols):
            R[i, j] = Q[:, j].dot(Q[:, i])
            Q[:,j] = Q[:,j]-R[i, j]*Q[:,i]
    return Q, R
```

Profiling `qr2()` on a 300×300 matrix produces the following output.

```
     48756 function calls in 1.047 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.863    0.863    1.047    1.047 profiling_hw.py:16(qr2)
 44850    0.171    0.000    0.171    0.000 {method 'dot' of 'numpy.ndarray' objects}
   300    0.003    0.000    0.006    0.000 function_base.py:526(asarray_chkfinite)
   300    0.003    0.000    0.005    0.000 linalg.py:1840(norm)
   600    0.002    0.000    0.002    0.000 {method 'any' of 'numpy.ndarray' objects}
   300    0.001    0.000    0.001    0.000 {method 'reduce' of 'numpy.ufunc' objects}
   301    0.001    0.000    0.001    0.000 {range}
   600    0.001    0.000    0.001    0.000 {numpy.core.multiarray.array}
   600    0.001    0.000    0.001    0.000 numeric.py:167(asarray)
   300    0.000    0.000    0.012    0.000 misc.py:7(norm)
     1    0.000    0.000    0.000    0.000 {method 'copy' of 'numpy.ndarray' objects}
   300    0.000    0.000    0.000    0.000 {method 'ravel' of 'numpy.ndarray' objects}
     1    0.000    0.000    1.047    1.047 <string>:1(<module>)
   300    0.000    0.000    0.000    0.000 {method 'conj' of 'numpy.ndarray' objects}
     1    0.000    0.000    0.000    0.000 {numpy.core.multiarray.zeros}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Our optimization reduced almost every kind of function call by half, and reduced the total run time by 0.295s.

Some less obvious ways to eliminate excess computations include moving computations out of loops, not copying large data structures, and simplifying mathematical expressions.

Avoid nested loops

The best way to do this is to use NumPy array operations instead of iterating through arrays. If you must use nested loops, focus your optimization efforts on the innermost loop, which gets called the most times.

Use existing functions instead of writing your own

If there is an intuitive operation you would like to perform on an array, chances are that NumPy or another library already has a function that does it. Python and NumPy functions have already been optimized, and are usually many times faster than the equivalent you might write. We saw an example of this in Lab **??** where we compared NumPy array multiplication with our own matrix multiplication implemented in Python.

Use generators when possible

When you are iterating through a list, you can often replace the list with a *generator*. Instead of storing the entire list in memory, a generator computes each item as it

is needed. For example, the code

```
>>> for i in range(100):
>>>     print i
```

stores the numbers 0 to 99 in memory, looks up each one in turn, and prints it. On the other hand, the code

```
>>> for i in xrange(100):
>>>     print i
```

uses a generator instead of a list. This code computes the first number in the specified range (which is 0), and prints it. Then it computes the next number (which is 1) and prints that.

It is also possible to write your own generators. See `https://docs.python.org/2/tutorial/classes.html#generators` and `https://wiki.python.org/moin/Generators` for more information.

In our example, replacing each `range` with `xrange` does not speed up `qr2()` by a noticeable amount.

Avoid excessive function calls

Function calls take time. Moreover, looking up methods associated with objects takes time. Removing "dots" can significantly speed up execution time.

For example, we could rewrite our function to reduce the number of times we need to look up the function `la.norm()`.

```
def qr2(A):
    norm = la.norm
    ncols = A.shape[1]
    Q = A.copy()
    R = np.zeros((ncols, ncols))
    for i in range(ncols):
        R[i, i] = norm(Q[:, i])
        Q[:, i] = Q[:, i]/R[i, i]
        for j in range(i+1, ncols):
            R[i, j] = Q[:, j].dot(Q[:, i])
            Q[:,j] = Q[:,j]-R[i, j]*Q[:,i]
    return Q, R
```

Once again, an analysis with `%prun` reveals that this optimization does not help significantly in this case.

Write Pythonic code

Several special features of Python allow you to write fast code easily. First, list comprehensions are much faster than for loops. For example, replace

```
>>> mylist = []
>>> for i in xrange(100):
>>>     mylist.append(math.sqrt(i))
```

with

```
>>> mylist = [math.sqrt(i) for i in xrange(100)]
```

When it can be used, the function `map()` is even faster.

```
>>> mylist = map(math.sqrt, xrange(100))
```

The analog of a list comprehension also exists for generators, dictionaries, and sets.

Second, swap values with a single assignment.

```
>>> a, b = 1, 2
>>> a, b = b, a
>>> print a, b
2 1
```

Third, many non-Boolean objects in Python have truth values. For example, numbers are `False` when equal to zero and `True` otherwise. Similarly, lists and strings are `False` when they are empty and `True` otherwise. So when `a` is a number, instead of

```
>>> if a != 0:
```

use

```
>>> if a:
```

Use Cython

We will discuss Cython at the end of this lab.

Use a more efficient algorithm

The optimizations discussed thus far will speed up your code at most by a constant. They will not change the complexity of your code. In order to reduce the complexity (say from $O(n^2)$ to $O(n\log(n))$), you typically need to change your algorithm.

When to Stop

You don't need to apply every possible optimization to your code. When your code runs acceptably fast, stop optimizing.

Cython

Cython code is basically Python with extra type declarations. This code is then compiled into C, which—depending on the details—can run much faster than the Python equivalent. In this lab we will introduce Cython as a language and discuss how it can be used to speed up Python code.

Compiling Cython

With a few exceptions, every Python program is also a Cython program. For example, suppose you save the following script as `cymodule.pyx`.

```
import numpy as np
from scipy import linalg as la

def qr(A):
    norm = la.norm
    ncols = A.shape[1]
    Q = A.copy()
    R = np.zeros((ncols, ncols))
    for i in range(ncols):
        R[i, i] = norm(Q[:, i])
        Q[:, i] = Q[:, i]/R[i, i]
        for j in range(i+1, ncols):
            R[i, j] = Q[:, j].dot(Q[:, i])
            Q[:,j] = Q[:,j]-R[i, j]*Q[:,i]
    return Q, R
```

To compile this code as Cython, we need to write another script, call it `setup.py`.

```
from distutils.core import setup
from Cython.Build import cythonize

setup(name="cymodule", ext_modules=cythonize('cymodule.pyx'))
```

setup.py

To run the setup script, type the following in the command line. [2]

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

Now start up IPython in the same directory where you ran the setup script. You can import `cymodule` just as if it were a Python module.

```
>>> import cymodule
>>> cymodule.qr(A)
```

Speeding up Cython with type declarations

So far, our Cythonized `qr()` function does not run any faster than the Python version. [3] The simplest way to take advantage of the C-compilation is to declare data types, as you would in a C program. All Cython data types and their NumPy equivalents are listed in Table 15.1.

In our example, when we declare the index `j` to be an `int`, the inner for loop is pushed into C (instead of Python), speeding up our code immensely. We do this by modifying `cymodule.pyx` as follows.

[2] Note that `python` in this line will need to refer to your version of Python that comes with Cython. Often, you will need to replace `python` with a path pointing to a specific distribution of Python.

[3] Sometimes you will see speedup immediately after Cythonizing a Python function.

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 (depends on platform)
unsigned long	uint32 or uint64	32 or 64 bit unsigned integer (depends on platform)
long long	int64	64 bit signed integer
unsigned long long	uint64	64 bit unsigned integer

Table 15.1: Numeric types available in Cython.

```
...
def qr():
    cdef int j
    ncols = A.shape[1]
    Q = A.copy()
    ...
```

We could also declare `i` to be an int, but the effect of doing so is negligible.

After modifying `cymodule.pyx`, we must recompile it by running the script `setup.py`, and then re-import it into Python. When we profile this function, we see that it is indeed a good deal faster than its Python cousin, and all the speed up is in the function itself (where the for loops are).

```
...
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.949    0.949    0.961    0.961 {cymodule.qr}
...
```

Similarly, you can speed up function calls in Cython by declaring the types of some or all of the arguments.

```
def myfunction(double arg1, int arg2, arg3):
    ...
```

A caution

NumPy's array methods are usually faster than a Cython equivalent you could code yourself. If you are unsure which method is fastest, time them.

Moreover, a good algorithm written with a slow language (like Python) is faster than a bad algorithm written in a fast language (like C). Hence, focus on writing fast algorithms with good Python code, and only use Cython when and where it is necessary.

Problem 1. Practice profiling and optimizing functions you have already written. Some suggested functions are listed below.

- LU decomposition (Lab 5).
- Householder triangularization or Hessenburg decomposition (Lab 7).
- Givens triangularization (Lab 8)
- Image Segmentation (Lab 9)
- Eigenvalue Solvers (Lab 10)

Your solution should include a before and after profiling of the function. It should also include a list of changes, the reasoning behind the changes, and the effect of the changes on runtime.

It is possible that you will not be able to speed up the function significantly. Remember it is more important for code to be readable than to execute quickly.

More on Cython (Optional)

This section has a more complete introduction to Cython. For another reference, see `http://docs.cython.org/src/tutorial/cython_tutorial.html`.

Compilation

Cython code is usually written in a `.pyx` file, which is then compiled to C. Next the C is compiled to a Python extension written in machine code. Figure 15.1 shows how Cython code is compiled and called.

These two compilations (first to C and then to a Python extension) are accomplished by the script `setup.py` discussed earlier in this lab.

```
from distutils.core import setup
from Cython.Build import cythonize

setup(name="cymodule", ext_modules=cythonize('cymodule.pyx'))
```

setup.py

The call to `cythonize()` compiles `cymodule.pyx` to C, and the call to `setup()` converts the C code to a Python extension.

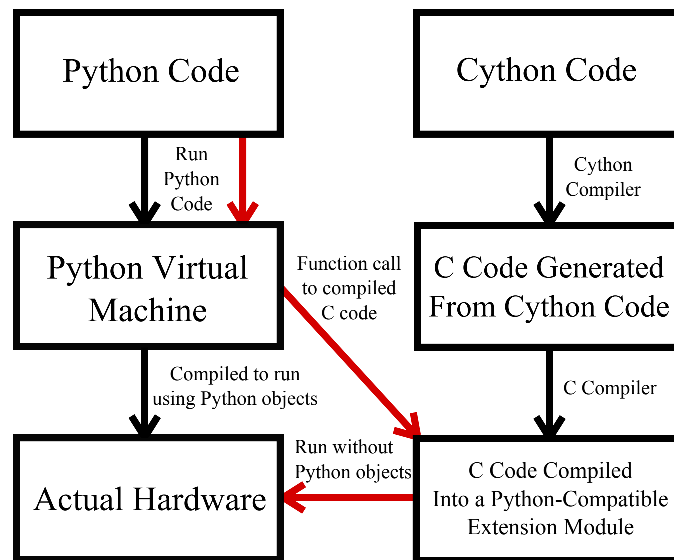

Figure 15.1: A diagram showing how Cython code is compiled and called. The path from calling a Cython function in a Python file to its evaluation is shown in red.

Note

You can learn more about the Python/C balance in your Cython file with the following line.

```
>>> cython -a cymodule.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.

Cython code can be compiled and imported to IPython with the Cython magic function. Load this function with the following command. [4]

```
>>> \%load_ext Cython
```

Now you can define any Cython function in the command line by prefacing it with `%%cython`.

```
>>> \%\%cython
>>> import numpy as np
>>> from scipy import linalg as la
>>> def qr():
>>>     cdef int j
>>> ...
```

[4] In older versions of IPython and Cython, you may need to use the command `%load_ext cythonmagic`.

Type Declarations

Type declarations make Cython faster than Python because a computer can process numbers faster when it knows ahead of time what type they are. Declaring a variable type in Cython makes the variable a native machine type instead of a Python object.

You can initialize a variable when it is declared or afterwards. It is possible to initialize several variables at once, as demonstrated below.

```
# 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
```

Warning

Unlike Python integers, which can be arbitrarily large, Cython integers can overflow.

Arrays in Cython

You can also declare types for arrays. Doing so produces a *typed memoryview*, or Cython array. As with ordinary variables, typed memoryviews can be initialized when they are declared or later.

```
# Define a NumPy array
A = np.linspace(0,1,6)

# Create typed memoryviews on A and B
cdef double[:] cA = A
cdef int[:,:] cB

# Initialize cB
cB = np.array([[1,2],[3,4],[5,6]], dtype=np.dtype("i"))
```

Accessing single entries from a memoryview is faster than accessing entries of a NumPy array. Also, passing slices of a memoryview to a Cython function is fast. However, memoryviews do not support mathematical operations. These must be performed on NumPy objects, or by looping through the arrays.

Note

Memoryviews can be passed as arguments to most NumPy functions. Numpy

includes functions for all common arithmetic operations. For example, you can add typed memoryviews with `c = np.add(a, b)`. This may or may not be faster than looping through the array.

Compiler Directives

When you access elements of an array, Cython checks that the indices are within bounds. Cython also allows negative indexing the same way Python does. These features slow down code execution. After a program has been carefully debugged, they may be removed via *compiler directives*.

Compiler directives in Cython can be included as comments or function decorators. Directives included in comments will apply to the whole file, while function decorators will only apply to the function immediately following. 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)
```

Cython has many other compiler directives, including `cdivision`. When `cdivision` is set to `True`, the `%` operator returns a number with the sign of the first argument (like in C). Also, division by 0 will no longer raise a `ZeroDivisionError`, which will increase the speed of your program.

More on functions in Cython

Speed up function calls in Cython by declaring the types of some or all of the arguments.

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

If we pass in a NumPy array for the argument `X`, Cython will convert it to a typed memoryview before it enters the function. However, if we pass in a NumPy array for `items`, it will remain a NumPy array in the function. Both typed and untyped arguments can be made into keyword arguments.

Cython also allows you to make C functions that are only callable within the C extension you are currently building. They are not ported into the Python namespace. These functions are declared using the same syntax as in Python, except the keyword `def` is replaced with `cdef`. The keyword `cpdef` combines `def` and `cdef` by creasting two versions of the function: one for Python and one for C.

You can also specify the return type for functions declared using `cdef` and `cpdef`, as in the code below.

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

Some examples

The following Python function computes the dot product of two 1-D arrays.

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

A C equivalent can be compiled from the following Cython code.

```
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
```

Figure 15.2 compares the speed of `pydot()`, `cydot()`, and the `dot()` method of a NumPy array.

For a second example, we will write functions which compute AA^T from A. That is, given `A`, these functions compute `B` where `B[i,j] = dot(A[i],A[j])`.

Here is the Python solution.

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

Here is the Cython solution. We changed the function `cydot()` to a C function since it will only be called by `cyrowdot()`. Also, the Cython solution uses a typed memoryview of its input.

```
import numpy as np
cimport cython
# cython: boundscheck=False
# cython: wraparound=False

cdef double cydot(double[:] A, double[:] B):
    cdef double tot=0.
```

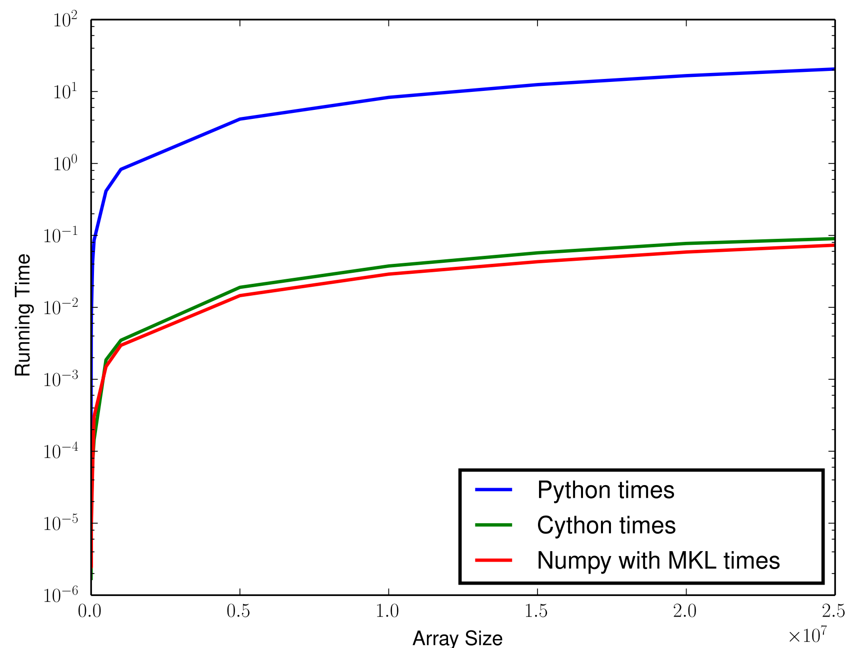

Figure 15.2: The running times of`pydot()`, `cydot()`, and the `dot()` method of a NumPy array on vectors of length "Array Size." The Cython version runs almost as fast as the version built into NumPy.

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

def cyrowdot(double[:,:] A):
    cdef double[:,:] B = np.empty((A.shape[0], A.shape[0]))
    cdef int i, j, n=A.shape[0]
    for i in xrange(n):
        for j in xrange(i):
           B[i,j] = cydot(A[i], A[j])
        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 `pyrowdot()`, `cyrowdot()`, and the NumPy command `A.dot(A.T)` are shown in Figure 15.3.

In both of these examples, NumPy's implementation was faster than the version we wrote in Cython.

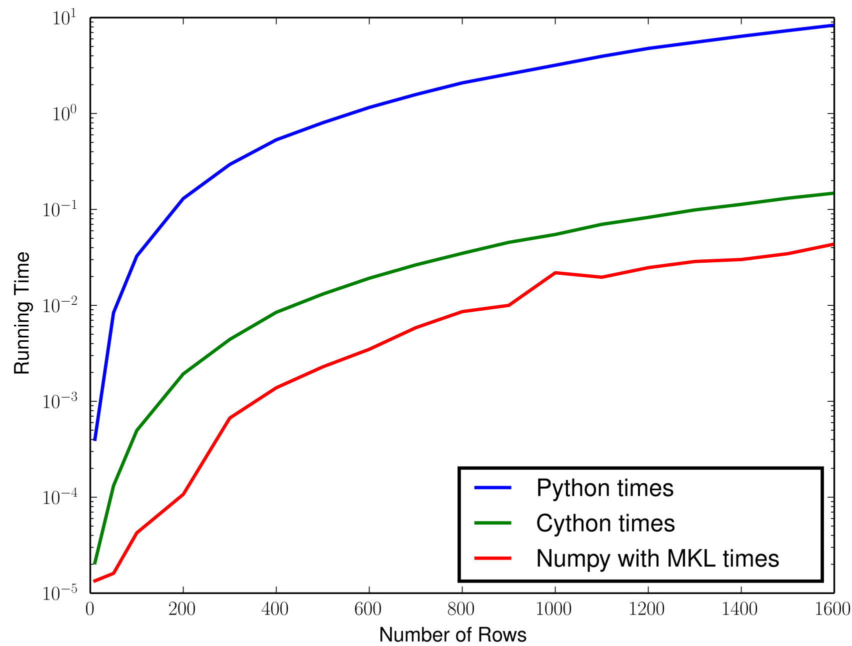

Figure 15.3: The timings of `pyrowdot()`, `cyrowdot()`, and the NumPy command `A.dot(A.T)`. The arrays used for testing were $n \times 3$ where n is shown along the horizontal axis.

Problem 2.

1. Write the following function in Python.

```
def pysum(X):
    ''' Return the sum of the elements of X.

    INPUTS:
    X - a 1-D NumPy array
    ''''
```

2. Rewrite the same function in Cython using a typed for-loop, a typed memoryview of `X`, and appropriate compiler directives.

3. Compare the speed of the functions you just wrote, the builtin `sum()` function, and NumPy's `sum()` function.

Problem 3. The code below defines a Python function which takes a matrix to the nth power.

```
def pymatpow(X, power):
    ''' Return X^{power}.

    INPUTS:
    X       - A square 2-D NumPy array
    power   - An integer
    '''
    prod = X.copy()
    temparr = np.empty_like(X[0])
    size = X.shape[0]
    for n in xrange(1, power):
        for i in xrange(size):
            for j in xrange(size):
                tot = 0.
                for k in xrange(size):
                    tot += prod[i,k] * X[k,j]
                temparr[j] = tot
            prod[i] = temparr
    return prod
```

1. Port `pymatpow()` to Cython using typed for-loops, typed arrays, and appropriate compiler directives.
2. Compare the speed of `pymatpow()`, the function you just wrote, and the `np.dot()` function.

NumPy takes products of matrices by calling BLAS and LAPACK, which are heavily optimized linear algebra libraries written in C, assembly, and Fortran.

Problem 4. In Lab 5 you wrote a function to compute the LU decomposition of a matrix.

1. Rewrite this function so it performs every operation element-by-element instead of using NumPy array operations.
2. Port the function from part (a) to Cython. Use typed for-loops, typed memoryviews, and appropriate compiler directives. You may assume that you are only dealing with real arrays of double precision floating point numbers.
3. Compare the speed of your new solutions to the speed of the NumPy version you wrote earlier.

The correct choice of algorithm is more important than a fast implementation. For example, suppose you wish to solve the following tridiagonal system.

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

One way to do this is with the general solve method in SciPy's `linalg` module. Alternatively, you could use an algorithm optimized for tridiagonal matrices. The code below implements one such algorithm in Python.

The final result is stored in `x`, and `c` is used to store temporary values.

```python
def pytridiag(a, b, c, x):
    '''Solve the tridiagonal system Ad = x where A has diagonals a, b, and c.

    INPUTS:
    a, b, c, x - All 1-D NumPy arrays.

    NOTE:
    The final result is stored in `x` and `c` is used to store temporary values.
    '''
    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 5.

1. Port the above code to Cython using typed for-loops, typed memoryviews and appropriate compiler directives.
2. Compare the speed of your new function with `pytridiag()` and `scipy.linalg.solve()`. To compare the first two functions, start with 1000000×1000000 sized systems. When testing the SciPy algorithm, start with 1000×1000 systems.
3. What do you learn about good implementation versus proper choice of algorithm?

Note that an efficient tridiagonal matrix solver is implemented by `scipy`.

```
sparse.linalg.spsolve().
```