

Lab 1

Profiling and Optimizing Python Code

Lab Objective: *Identify which portions of the code are most time consuming using a profiler. Optimize Python code using Cython and Numba and other tools.*

The best code goes through multiple drafts. In a first draft, you should focus on writing code that does what it is supposed to and is easy to read. After writing a first 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 ??).

```
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[:, i].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 IPython¹, you can profile a function with `%prun`. Here we profile `qr1()` on a random 300×300 array.

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

```
In [1]: A = np.random.rand(300, 300)
In [2]: %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? Here are some of the techniques we will address in this lab:

- Avoid Recomputing Values
- Avoid Nested Loops
- Use Existing Functions Instead of Writing Your Own
- Use Generators When Possible
- Avoid Excessive Function Calls
- Write Pythonic Code

- Use Cython or Numba
- Use a More Efficient Algorithm

You should always 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))
    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	totttime	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

For many algorithms, the temporal complexity of an algorithm is determined by the loops. Nested loop quickly increase the temporal complexity. The best way to avoid

nested loops 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. These are particularly useful when building lists inside a loop. 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:

```

Problem 1. Using the profiling techniques discussed in this lab, find out which portions of the code below require the most runtime. Then, rewrite the function using some of the optimization techniques we have discussed thus far.

```
def foo(n):
    my_list = []
    for i in xrange(n):
        num = np.random.randint(-9,9)
        my_list.append(num)
    evens = 0
    for j in xrange(n):
        if j%2 == 0:
            evens += my_list[j]
    return mylist, evens
```

Hint: If you are unsure where to begin optimizing, walk through the code line by line to determine what the code is accomplishing. Then, write your own function to perform the same task in a more efficient way.

Use Cython or Numba

Though it is much easier to write simple, readable code in Python, it is also much closer than compiled languages such as C. Compiled languages, in general, are much faster. Cython and Numba are two tools that you can use to optimize your code. Cython code is compiled before execution whereas Numba uses *just-in-time* (JIT) compilation. We will first discuss Cython.

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 section we will introduce Cython as a language and discuss how it can be used to speed up Python code.

Compilation in IPython

Cython code can be compiled and imported to IPython (and IPython Notebooks) with the Cython magic function. Load this function with the following command:²

```
In [1]: %load_ext Cython
```

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

```
In [2]: %%cython
...: import numpy as np
...: from scipy import linalg as la
...: def qr(A):
...:     norm = la.norm
...:     ncols = A.shape[1]
...:     Q = A.copy()
```

²In older versions of IPython and Cython, you may need to use the command `%load_ext cythonmagic`.

```

...:     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
...:

```

Note the import statements on lines 2 and 3. Even if these modules were already loaded in the current IPython session, these lines would still be necessary. Now the `qr()` function has been compiled in Cython and can be called from IPython just as any other function.

It is best to use this method of compilation if you are only using your optimized function once or you are in the stages of developing your optimized algorithm. It can be a little frustrating to fix errors in your function if you are using the IPython shell. The IPython Notebook interface is much easier to use for these types of problems.

Compilation from the Command Line

Compiling Cython from the command line is particularly useful if you need an optimized module that you will use over and over again. As we will discuss in this section, once a function is written and compiled in Cython, it can be imported as a module.

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 1.1 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.

```

1 from distutils.core import setup
2 from Cython.Build import cythonize
4 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.

NOTE

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

```
$ cython -a cymodule.pyx
```

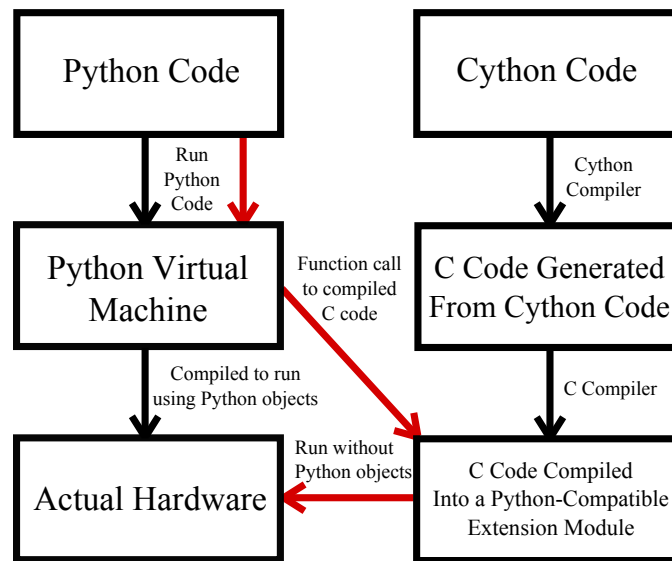


Figure 1.1: A diagram showing how Cython code is compiled and called. The path from calling a Cython function in a Python file to its evaluation is shown in red.

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.

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 1.1.

WARNING

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

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.

```

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

```

³Sometimes 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 1.1: Numeric types available in Cython.

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):
    ...
```

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.

Putting it All Together

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 by running the following code in IPython:

```
%cython
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 1.2 compares the speed of `pydot()`, `cydot()`, and the `dot()` method of a NumPy array.

Problem 2.

1. Write the following function in Python without using the `sum()` builtin function.

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

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

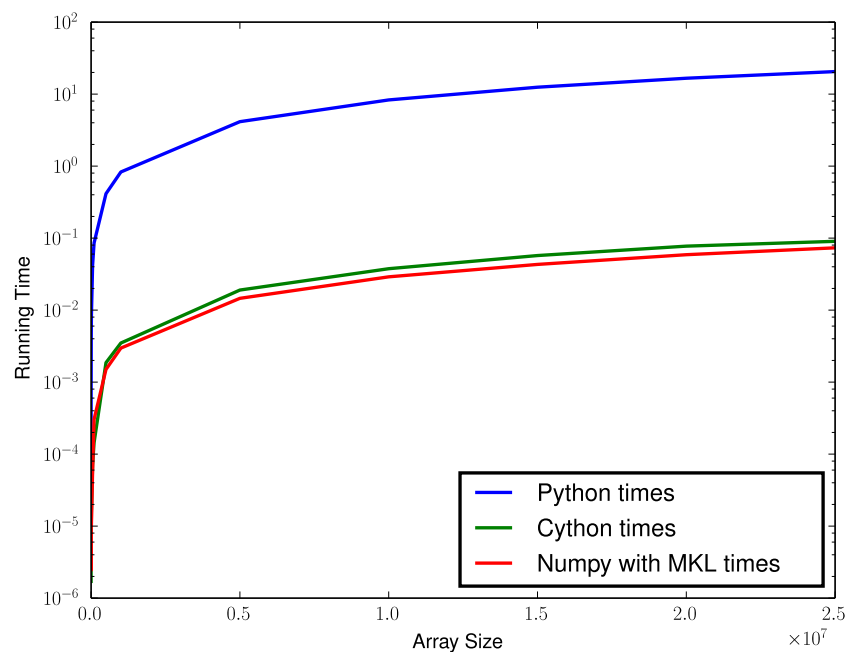


Figure 1.2: The running times of `pydot()`, `cydot()`, and the `dot()` method of a NumPy array on vectors of length “Array Size.” The Cython version runs almost as fast as the version built into NumPy.

2. Rewrite the same function in Cython using a typed for-loop, and a typed memoryview of `x`. Call this function `cysum()`. Assume the input array `x` is an array of double precision floating point numbers.
3. Compare the `pysum()` function, the `cysum()`, the Python builtin `sum()` function, and the `np.sum()` function. Print the results to the terminal.

Problem 3. In the Affine Transformations Lab (Lab ??) you wrote a function to compute the LU decomposition of a matrix. One possible implementation is found in the following code:

```
def LU(A):
    """Returns the LU decomposition of a square matrix."""
    n = A.shape[0]
    U = np.array(np.copy(A), dtype=float)
    L = np.eye(n)

    for i in xrange(1,n):
        for j in xrange(i):
```

```
L[i,j] = U[i,j]/U[j,j]
U[i,j:] -= L[i,j]*U[j,j:]
return L,U
```

1. Cython does not recognize NumPy array slicing. To be able to optimize this function using Cython, all array slicing must be removed. Rewrite the LU decomposition function so it performs every operation element-by-element instead of using NumPy array operations.
2. Port your function from Part 1 to Cython. Use typed for-loops and typed memoryviews. You may assume that you are only dealing with real arrays of double precision floating point numbers.
3. Compare the speeds of the original code, your solution to Part 1, and your solution to Part 2. Print the results to the terminal.

Numba

As mentioned before, Numba relies on a just-in-time (JIT) compiler. This means that the code is compiled right before it is executed. We will discuss this process a bit later in this section. The API for using Numba is incredibly simple. All one has to do is import Numba and add the `@jit` function decorator to your function. The following code would be a Numba equivalent to Problem 2.

```
from numba import jit
@jit
def numba_sum(A):
    total = 0
    for i in xrange(len(A)):
        total += A[i]
    return total
```

Though this code looks very simple, a lot is going on behind the scenes. As we saw with Cython, our code will run faster if datatypes are assigned to all our variables. With Cython, we had to explicitly define the datatypes of each variable. Rather than requiring us to define the datatypes explicitly, Numba attempts to *infer* the correct datatypes based on the datatypes of the input.

In the code above, for example, say that our array `A` was full of integers. Though we have not explicitly defined a datatype for the variable `total`, Numba will infer that the datatype for `total` should also be an integer.

Once all the datatypes have been inferred and assigned, the code is translated to machine code by the LLVM library. Numba will then cache this compiled version of our code. This means that we can bypass this whole inference and compilation process the next time we run our function.

More Control Within Numba

Though the inference engine within Numba does a good job, it's not always perfect. If you would like to have more control, you may specify datatypes explicitly as demonstrated in the code below.

Also, if you add the keyword argument, `nopython=True` to the `jit` decorator, an error will be raised if Numba was unable to convert everything to explicit datatypes.

In this example, we will assume that the input will be doubles. Note that is necessary to import the desired datatype from the Numba module.

```
from numba import double
@jit(nopython=True, locals=dict(A=double[:], total=double))
def numba_sum(A):
    total = 0
    for i in xrange(len(A)):
        total += A[i]
    return total
```

Notice that the `jit` function decorator is the only thing that changed. Note also that this means that we will not be allowed to pass an array of integers to this function. If we had not specified datatypes, the inference engine would allow us to pass arrays of any numerical datatype. In the case that our function sees a datatype that it has not seen before, the inference and compilation process would have to be repeated. As before, the new version will also be cached.

If your function is running slower than you would expect, you can find out what is going on under the hood by calling the `inspect_types()` method of the function.

```
# Due to the length of the output, we will leave it out of the lab text.
>>> numba_sum.inspect_types()
```

If you are trying to achieve the fastest runtimes possible, it will be worth trying both Numba and Cython. Since Numba relies on an LLVM compiler and Cython relies on a C compiler, it is possible that these two techniques will produce different results.

Problem 4. The code below defines a Python function which takes a matrix to the n th power.

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

Inputs:

- X (array) - A square 2-D NumPy array
- power (int) - The power to which we are taking the matrix X.

Returns:

- prod (array) - 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(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 and typed memoryviews. Call your function `cymatpow()`.
2. Create a compiled version of `pymatpow()` using Numba. Call this function `numbamatpow()`.
3. Compare the speed of `pymatpow()`, `cymatpow()`, `numbamatpow()` and the `np.dot()` function. Remember to time `numbamatpow()` on the second pass so the compilation process is not part of your timing. Print your results to the terminal.

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

A Caution

NumPy's array methods are often faster than a Cython or Numba 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 or Numba when and where it is necessary.

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. We will address the benefits of using more efficient algorithms in Problem 5.

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

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & \cdots & 0 \\ 0 & a_3 & b_3 & c_3 & \cdots & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & c_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ \vdots \\ \vdots \\ d_n \end{bmatrix}$$

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. This is called the Thomas algorithm.

```
def pytridiag(a,b,c,d):
    """Solve the tridiagonal system  $Ax = d$  where  $A$  has diagonals  $a$ ,  $b$ , and  $c$ .

    Inputs:
        a, b, c, d (array) - All 1-D NumPy arrays of equal length.
    Returns:
        x (array) - solution to the tridiagonal system.
    """
    n = len(a)

    # Make copies so the original arrays remain unchanged
    aa = np.copy(a)
    bb = np.copy(b)
    cc = np.copy(c)
    dd = np.copy(d)

    # Forward sweep
    for i in xrange(1, n):
        temp = aa[i]/bb[i-1]
        bb[i] = bb[i] - temp*cc[i-1]
        dd[i] = dd[i] - temp*dd[i-1]

    # Back substitution
    x = np.zeros_like(a)
    x[-1] = dd[-1]/bb[-1]
    for i in xrange(n-2, -1, -1):
        x[i] = (dd[i]-cc[i]*x[i+1])/bb[i]

    return x
```

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. You may use the code below to generate the arrays, a , b , and c , along with the corresponding tridiagonal matrix A .

```
def init_tridiag(n):
    """Initializes a random nxn tridiagonal matrix A.

    Inputs:
        n (int) : size of array

    Returns:
        a (1-D array) : (-1)-th diagonal of A
```

```

b (1-D array) : main diagonal of A
c (1-D array) : (1)-th diagonal of A
A (2-D array) : nxn tridiagonal matrix defined by a,b,c.
"""
a = np.random.random_integers(-9,9,n).astype("float")
b = np.random.random_integers(-9,9,n).astype("float")
c = np.random.random_integers(-9,9,n).astype("float")

A = np.zeros((b.size,b.size))
np.fill_diagonal(A,b)
np.fill_diagonal(A[1:,-1:],a[1:])
np.fill_diagonal(A[:-1,1:],c)
return a,b,c,A

```

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()`.

When to Stop Optimizing

You don't need to apply every possible optimization to your code. When your code runs acceptably fast, stop optimizing. There is no need spending valuable time making optimizations once the speed is sufficient.

Moreover, remember not to prematurely optimize your functions. Make sure the function does exactly what you want it to before worrying about any kind of optimization.

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.

More on 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
```

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 `cddivision`. When `cddivision` 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 creating 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):
    ...
```

More Examples

For another example, we will write functions which compute AA^T from A . That is, given A , these functions compute B where $B[i,j] = \text{dot}(A[i], A[j])$.

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

cpdef 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

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 1.3.

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

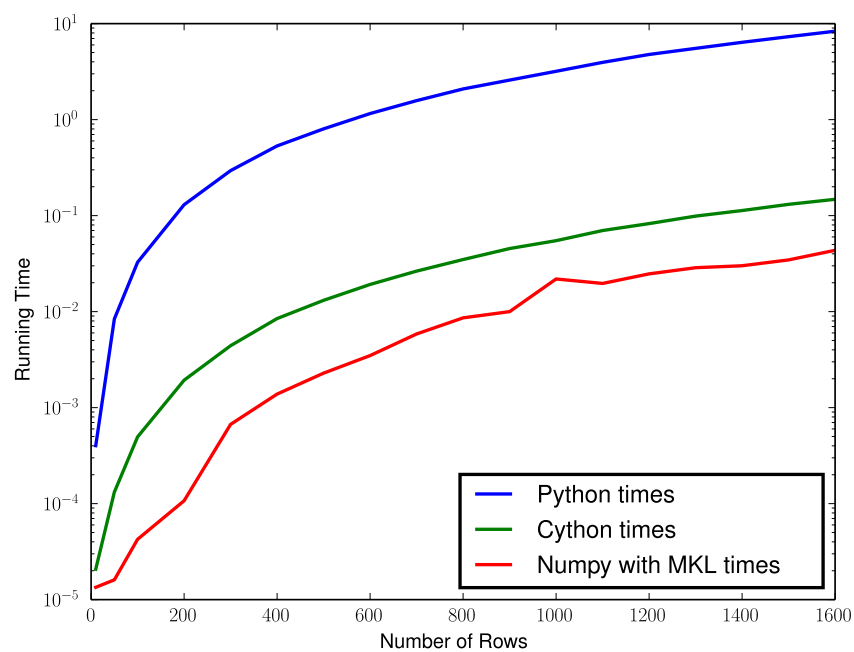


Figure 1.3: The timings of `pyrowdot()`, `cyrowdot()`, and the NumPy command `A.dot(A.T)`. The arrays used for testing were $n \times 3$ where n is shown along the horizontal axis.