Lab 19

Profiling

Lab Objective: *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. Once you have working code, you may need to speed it up to meet the demands of the application. In this lab we learn to identify the parts of code that take the most time to run and how speed up slow code.*

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

```
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
```

Profiling Slow Code

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 with `%prun`. Here we profile `qr1()` on a random 300×300 array.

```
In [1]: A = np.random.random((300, 300))
```

[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`.

```
In [2]: %prun qr1(A)
```

On the author's 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.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.

Speeding Up Code

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
- Compiling Using 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.

Problem 1. In this lab, we will perform many comparisons between the runtimes of various functions. To help with these comparisions, implement the following function:

```
def compare_timings(f, g, *args):
    """Compare the timings of 'f' and 'g' with arguments '*args'.

    Inputs:
        f (func): first function to compare.
        g (func): second function to compare.
        *args: arguments to use when callings functions 'f' and 'g',
            i.e., call f with f(*args).
    Returns:
        comparison (str): The comparison of the runtimes of functions
            'f' and 'g' in the following format:
                Timing for <f>: <time>
                Timing for <g>: <time>
    """
```

(Hint: You can gain access to the name of many functions by using its `func_name()` method. However, this method does not exist for all functions we will be interested in timing. Therefore, even though it is not as clean, use `str(f)` to print a string representation of f.)

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] # this line changed
        for j in range(i+1, ncols):
            R[i, j] = Q[:, j].dot(Q[:, i])
            Q[:,j] = Q[:,j]-R[i, j]*Q[:,i] # this line changed
    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.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 its loops. Nested loops 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.

Problem 2. The code below is an inefficient implementation of the LU algorithm. Write a function `LU_opt()` that is an optimized version of `LU()`. Look for ways to avoid recomputing values and avoid nested loops by using array slicing instead. Print a comparison of the timing of the original function and your optimized function using your `compare_timings()` function.

```
def LU(A):
    """Return 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 range(1, n):
        for j in range(i):
            L[i,j] = U[i,j] / U[j,j]
            for k in range(j, n):
                U[i,k] -= L[i,j] * U[j,k]
    return L, U
```

Use Built-in Functions

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.

Problem 3. Without using any builtin functions, implement the following function.

```
def mysum(x):
    """Return the sum of the elements of X without using a built-in ↩
        function.

    Inputs:
        x (iterable): a list, set, 1-d NumPy array, or another iterable.
    """
```

Compare `mysum()` to Python's builtin `sum()` function and NumPy's `np.sum()` using your `compare_timings()` function.

Use Generators

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.

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

Though the example below is contrived, it demonstrates the benefits of using generators.

```
# both these functions will return the first iterate of a loop of length 10^8.
def list_iter():
    for i in range(10**8):              # Use range()
        return i

def generator_iter():
    for i in xrange(10**8):             # Use xrange()
        return i
```

```
>>> compare_timings(list_iter,generator_iter)
Timing for <function list_iter at 0x7f3deb5a4488>: 1.93316888809
Timing for <function generator_iter at 0x7f3deb5a4500>: 1.19209289551e-05
```

It is also possible to write your own generators. Say we have a function that returns an array. And say we want to iterate through this array later in our code. In situations like these, it is valuable to consider turning your function into a generator instead of returning the whole list. The benefits of this approach mirror the benefits of using `xrange()` instead of `range()`. The only thing that needs to be adjusted is to change the `return` statement to a `yield` statement. Here is a quick example:

```
def return_squares(n):
    squares = []
    for i in xrange(1,n+1):
        squares.append(i**2)
    return squares

def yield_squares(n):
    for i in xrange(1,n+1):
        yield i**2
```

The `yield` statement "returns" the single value, but all the local variables for the function are stored away until the next iteration. To iterate step-by-step through a generator, use the builtin `next()` function..

```
>>> squares = yield_squares(3)
>>> next(squares)
1
>>> next(squares)
4
>>> next(squares)
9
```

We can also iterate through a generator using a for loop.

```
>>> for s in yield_squares(3):
...     print s,
...
1 4 9
```

Problem 4. The *Fibonacci sequence* is defined by the recurrence relation $F_{n+1} = F_n + F_{n-1}$, where $F_1 = F_2 = 1$. Write a generator that yields the first n Fibonacci numbers.

If you are interested in learning more about writing your own generators, see `https://docs.python.org/2/tutorial/classes.html#generators` and `https://wiki.python.org/moin/Generators`.

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      # This reduces the number of function lookups.
    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)]
```

We can accomplish the same thing using the `map()` function, which 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:
```

Lastly, it is more efficient to iterate through lists by iterating over the elements instead of iterating over the indices.

```
# Bad
for i in xrange(len(my_list)):
    print my_list[i],

# Good
for x in my_list:
    print x,
```

However, there are situations where you will need to know the indices of the elements over which you are iterating. In these situations, use `enumerate`.

Problem 5. Consider the following poorly-written function.

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

Walk through the code line by line to determine what the code is accomplishing. Using `%prun`, find out which portions of the code below require the most runtime. Rewrite the function to perform the same task in a more efficient way using the optimization techniques we have discussed.

Numba

Though it is much easier to write simple, readable code in Python, it is also much slower than compiled languages such as C. Compiled languages, in general, are much faster. *Numba* is a tool that uses *just-in-time* (JIT) compilation to optimize code, meaning 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 3.

```
from numba import jit
@jit
def numba_sum(A):
    total = 0
    for x in A:
        total += x
    return total
```

Though this code looks very simple, a lot is going on behind the scenes. One of the reasons compiled languages like C are so much faster than Python is because they have explicitly defined datatypes. The main strategy used by Numba is to speed up the Python code by assigning datatypes to all the variables. Rather than requiring us to define the datatypes explicitly as we would need to in any compiled language, 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 an array 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. There are times that Numba is unable to infer all the datatypes correctly.

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.

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. Using this, you can see if all the datatypes are being assigned as you would expect.

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

If you would like to have more control, you may specify datatypes explicity as demonstrated in the code below.

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
# The values inside 'dict' will be specific to your function.
@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.

Problem 6. The code below defines a Python function which takes a matrix to the nth power.

```
def pymatpow(X, power):
    """Return X^{power}, the matrix product XX...X, 'power' times.

    Inputs:
        X ((n,n) ndarray): A square matrix.
        power (int): The power to which to raise X.
    """
    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. Create a function `numba_matpow()` that is the compiled version of `pymatpow()` using Numba.

2. Write a function `numpy_matpow()` that performs the same task as `pymatpow` but uses `np.dot()`. Compile this function using Numba.

3. Compare the speed of `pymatpow()`, `numba_matpow()` and the `numpy_matpow()` function. Remember to time `numba_matpow()` and `numpy_matpow()` on the second pass so the compilation process is not part of your timing. Perform your comparisons using your `compare_timings()` function.

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

Achtung!

NumPy's array methods are often faster than a Numba equivalent that you

could code yourself. If you are unsure which method is fastest, time them.

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 7.

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 Numba when and where it is necessary. In other words, Numba will not always save you from a poor algorithm design.

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

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

One way to do this is with the `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 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 ((n-1,) ndarray): first subdiagonal of A.
        b ((n,) ndarray): main diagonal of A.
        c ((n-1,) ndarray): first superdiagonal of A.
        d ((n,) ndarray): the right side of the linear system.

    Returns:
        x ((n,) array): solution to the tridiagonal system Ax = d.
    """
    n = len(b)

    # 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-1] / 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(b)
    x[-1] = dd[-1] / bb[-1]
    for i in reversed(xrange(n-1)):
        x[i] = (dd[i] - cc[i]*x[i+1]) / bb[i]

    return x
```

Problem 7.

1. Write a function that is a compiled version of `pytridiag()`.
2. Compare the speed of your new function with `pytridiag()` and `scipy.linalg.solve()`. When comparing `numba_tridiag()` and `pytridiag()`, use 1000000×1000000 sized systems. When comparing `numba_tridiag()` and the SciPy algorithm, use a 1000×1000 systems.

You may use the code below to generate the arrays `a`, `b`, and `c`, as well as the matrix A.

```
def init_tridiag(n):
    """Construct a random nxn tridiagonal matrix A by diagonals.

    Inputs:
        n (int): The number of rows / columns of A.

    Returns:
        a ((n-1,) ndarray): first subdiagonal of A.
        b ((n,) ndarray): main diagonal of A.
        c ((n-1,) ndarray): first superdiagonal of A.
        A ((n,n) ndarray): the tridiagonal matrix.
    """
    a = np.random.random_integers(-9, 9, n-1).astype("float")
    b = np.random.random_integers(-9 ,9, n  ).astype("float")
    c = np.random.random_integers(-9, 9, n-1).astype("float")

    # Replace any zeros with ones.
    a[a==0] = 1
    b[b==0] = 1
    c[c==0] = 1

    # Construct the matrix A.
    A = np.zeros((b.size,b.size))
    np.fill_diagonal(A, b)
    np.fill_diagonal(A[1:,:-1], a)
    np.fill_diagonal(A[:-1,1:], c)

    return a, b, c, A
```

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.