PySHop Session 1

Introduction to Python and Open Source Software

Tyler Abbot

Department of Economics
Sciences Po

Fall 2015

Outline

Does Python have a speed problem?

Depends on your perspective.

- Native Python is slower than Matlab
- NumPy is about as fast as Matlab
- Python is slower than C++ or Fortran
- What drives this speed issue? Is it an issue?

The Interpreter

Ease means speed squeeze (pathetic rhymeing)

- Each time Python encounters an object it dynamically checks its type
- This is not a problem in MatLab, C, R, or Fortran
- Global Interpreter Lock (GIL)
- Ways around this problem are numerous...

How to reclaim speed.
Vroom

- NumPy
- Cython
- Just-in-time (JIT) complitation: Numba
- Multiprocessing
- PyCUDA

Monte Carlo Moment Estimation

- Take a random variable x distributed according to the pdf $g(x)$
- We will look to estimate the following moment

$$\phi = \mathbb{E}[f(x)] = \int_X f(x)g(x)dx$$
$$X \subset \mathbb{R}^n$$

- Estimating this integral by quadrature rules is often impossible in high dimensions (n)
- For instance, $n = 10$ implies that, for 10 quadrature points in each direction, $10^{10} = 10,000,000,000$ function evaluations

Monte Carlo Integration

Sorry, but I'll skip derivations

- Take a uniform sample of M points $\{x_i\}_{i=1}^{M}$ from the support of the integral
- Calculate an approximation of the integral as the following

$$\mathbb{E}[f(x)] \approx \hat{\phi} = \frac{V}{M} \sum_{i=1}^{M} f(x_i) g(x_i)$$

where $V = \int_X dx =$ Volume of Support

- Can also get a variance estimate for our moment estimator

$$\sigma^2 \approx \hat{\sigma^2} = \frac{V^2}{M(M-1)} \sum_{i=1}^{M} \left(f(x_i) g(x_i) - \frac{1}{V} \hat{\phi} \right)^2$$

A Concrete Example

What moment and what distribution?

- As an example we'll take the L^2 norm and the N-dimensional uniform distribution with support being a hypercube in $\mathbb{R}^n$:

$$f(x) = \|x\|^2 = \sum_1^N x_i^2$$

$$g(x) = Uni(X) = \prod_1^N \frac{1}{\overline{X_i} - \underline{X_i}}$$

$$X = \times_{i=1}^n [\underline{X_i}, \overline{X_i}]$$

- This moment admits a closed form solution:

$$\phi = \mathbb{E}[f(x)] = \frac{n}{3}$$

A loop based solution
Slow and Steady

```
def monte_carlo_expectation_serial(f, g, M, X):
    """
    A loop based monte carlo integration.

    inputs:
        f     :    function; the moment to be estimated
        g     :    function; the distribution function of x
        M     :    scalar; the number of points to sample
        X     :    ndarray; bounds in R^n for the support

    """
    # Generate points uniformly from the sample space
    N = X.shape[0]
    points = np.random.rand(M, N)

    #Scale the points
    points = points*(np.array([X[:, 1], ]*M)
                      - np.array([X[:, 0], ]*M))
                      +  np.array([X[:, 0], ]*M)
```

A loop based solution II

Slow and Steady

```
    # Calculate the volume.  Assume a hyperrectangle for support.
    V = np.prod(np.abs(X[:,1] - X[:,0]))

    # Calculate the sum as a loop
    sum1 = 0.0
    for i in range(0, M):
        sum1 += f(points[i, :])*g(points[i, :], X)
    sum1 /= M

    # Calculate the variance
    var1 = 0.0
    for i in range(0, M):
        var1 += (f(points[i, :])*g(points[i, :], X) - sum1)**2

    # Return the result
    return V*sum1, V**2*var1/(M*(M - 1))
```

A loop based solution III

Define f and f

Our function takes a moment, f, and a pdf, g, as arguments. So we define those here.

```python
def f(x):
    return np.linalg.norm(x)**2

def g(x, X):
    return np.prod(1/(X[:, 1] - X[:, 0]))
```

A loop based solution IV
Running the code

Now we can run the code to see how we did:

```
M = 10
N = 2
X = np.array((0*np.ones(N), 1*np.ones(N))).T

monte_carlo_expectation_serial(f, g, M, X)
```

```
Out[5]:  (0.83253221789825815, 0.02161839613412718)
```

A loop based solution V

Running the code II

```
for M in [100, 1000, 10000]:
    print(monte_carlo_expectation_serial(f, g, M, X))
```

```
Out:  (0.6556763952214224, 0.0017492598946346404)
      (0.6534139829243473 8, 0.00017119601191807105)
      (0.6615490162460285 8, 1.739551947398236 6e-05)
```

```
for M in [100, 1000, 10000]:
    %timeit monte_carlo_expectation_serial(f, g, M, X)
```

```
Out:  100 loops, best of 3: 1.84 ms per loop
      100 loops, best of 3: 18.6 ms per loop
      1 loops, best of 3: 182 ms per loop
```

A loop based solution V

Assesing

- Success!
- Our function conferges to the correct value. We can do this with different values of `M` and `N`, but I'll leave that for the notes
- But why are we using loops!? Didn't you tell me loops are bad?
- For comparison reasons.Now, let's do it in NumPy!

Why is NumPy fast?

A brief recap.

- Typing
- Pre-compiled code
- Numerical algorithms
- Magic

A NumPy based solution

Arrays!

```python
def monte_carlo_expectation_numpy(f, g, M, X):
    """
    A loop based monte carlo integration.

    inputs:
        f    :    function; the moment to be estimated
        g    :    function; the distribution function of x
        M    :    scalar; the number of points to sample
        X    :    ndarray; bounds in R^n for the support

    """
    # Generate points uniformly from the sample space
    N = X.shape[0]
    points = np.random.rand(M, N)

    #Scale the points
    points = points*(np.array([X[:, 1], ]*M)
                     - np.array([X[:, 0], ]*M))
                     +  np.array([X[:, 0], ]*M)
```

A NumPy based solution II

Arrays! part deux

```
    # Calculate the volume.  Assume a hyperrectangle for support.
    V = np.prod(np.abs(X[:,1] - X[:,0]))

    # Calculate the sum as a dot product
    sum1 = np.dot(f_vec(points), g_vec(points, X))/M

    # Calculate the variance
    var1 = np.linalg.norm(f_vec(points)*g_vec(points, X) - sum1)**2

    # Return the result
    return V*sum1, V**2*var1/(M*(M - 1))
```

A NumPy based solution III

Redefine f and g

We also need to redefine our functions f and g to take array arguments. This entails only changing the axis argument, as they were already numpy functions.

```python
def f_vec(x):
    return np.linalg.norm(x, axis = 1)**2

def g_vec(x, X):
    return np.prod(np.ones((x.shape[0], x.shape[1]))
                   /(X[:, 1] - X[:, 0]), axis=1)
```

A NumPy based solution IV

Running the code

Now we can run the code to see how we did, comparing the serial and NumPy answers:

```
M = 100
N = 2
X = np.array((0*np.ones(N), 1*np.ones(N))).T

print(monte_carlo_expectation_serial(f, g, M, X))
print(monte_carlo_expectation_numpy(f_vec, g_vec, M, X))
```

```
Out[5]:  (0.6134337201688502‌3, 0.0016344965630260006)
         (0.6008107609243664‌7, 0.0014187899662249886)
```

NOTE: The two use different samples, so the numbers will differ, but we are not too far off!

A NumPy based solution V

Running the code II

```
M = 100
N = 2
X = np.array((0*np.ones(N), 1*np.ones(N))).T
for M in [100, 1000, 10000]:
    print("M = %s" %M)
    %timeit monte_carlo_expectation_serial(f, g, M, X)
    %timeit monte_carlo_expectation_numpy(f_vec, g_vec, M, X)
```

```
Out:  M = 100
The slowest run took 6.25 times longer than the fastest.
This could mean that an intermediate result is being cached
1000 loops, best of 3: 1.8 ms per loop
10000 loops, best of 3: 114 $\mu$s per loop
M = 1000
100 loops, best of 3: 18 ms per loop
1000 loops, best of 3: 712 $\mu$s per loop
M = 10000
10 loops, best of 3: 181 ms per loop
100 loops, best of 3: 6.6 ms per loop
```

A NumPy based solution V

Assesing

- No big suprise.
- NumPy is 20-30 times faster than native Python.
- We can go beyond NumPy to try to squeeze out even more speed.
- The easiest way: types.

Typing Speed

- Several ways to boost speed using types.
- Need to tell the interpreter what type your objects are.
- Most powerful: Cython. We will not do this.
- Easiest: Just-in-Time (JIT) compiler
- Black box? Very complicated and very challenging problem combining compilation and interpretation (kind of like combining C and Python TOGETHER!)
- In a nut shell: the first time Python encounters a variable, it remembers the type and reuses this information later.

Numba
Huh?

- Continuum sponsored project (like Anaconda)
- Makes JIT incredibly easy
- Has much more functionality that we won't cover.
- Let's see how it works!

A JITed solution

What?! So easy!

```
# Simply add the jit decorator before the function
@jit
def jitted_monte_carlo_serial(f, g, M, X):
...
```

That's it! You add the decorator before the function and you're done!
Do the same thing to the NumPy function and then run them both using `%timeit`...

A JITed II

Running the code

```
M = 100
N = 2
X = np.array((0*np.ones(N), 1*np.ones(N))).T

%timeit jitted_monte_carlo_serial(f, g, M, X)
%timeit jitted_monte_carlo_numpy(f_vec, g_vec, M, X)
```

```
The slowest run took 332.66 times longer than the fastest.
This could mean that an intermediate result is being cached
1 loops, best of 3: 1.45 ms per loop
The slowest run took 1625.85 times longer than the fastest.
This could mean that an intermediate result is being cached
1 loops, best of 3: 160 $\mu$s per loop
```

A JITed solution III

Intermediate Assessment

- The first loop is much slower than the rest...
- The result is actually not that much faster...
- One possible reason: we call un-JITed functions f and g
- Add the `@jit` decorator to these functions and try again

NOTE: I'm ommitting the code, but it's so simple!

A JITed solution IV

Running Again

```
M = 100
N = 2
X = np.array((0*np.ones(N), 1*np.ones(N))).T

%timeit jitted_monte_carlo_serial(jit_f, jit_g, M, X)
%timeit jitted_monte_carlo_numpy(jit_f_vec, jit_g_vec, M, X)
```

```
The slowest run took 58.47 times longer than the fastest.
This could mean that an intermediate result is being cached
1000 loops, best of 3: 1.62 ms per loop
10000 loops, best of 3: 153 $\mu$s per loop
```

A JITed solution V

Failure!

- Nothing! Really?!
- Our functions are NumPy functions.
- NumPy functions use precompiled C and types are pre-defined
- Conclusion: JITing a NumPy function is kind of redundant...
- Only really get gain from JIT if have many operations
- Let's see an example that gives some more promising results

A JITed Example

Something that works

```python
def test_func(x):
    return np.sum(x)

@jit
def jit_test_func(x):
    return np.sum(x)

x = np.arange(1000)
%timeit test_func(x)
%timeit jit_test_func(x)
```

```
The slowest run took 16.57 times longer than the fastest.
This could mean that an intermediate result is being cached
100000 loops, best of 3: 2.2 $\mu$s per loop
The slowest run took 43725.24 times longer than the fastest.
This could mean that an intermediate result is being cached
1000000 loops, best of 3: 590 ns per loop
```

Speed From Typing

Taking stock

- Numba is so easy you should just try it
- When it works best is when object types are checked often
- In our example it is not useful because NumPy is already so fast
- To get the biggest benefit use simple functions (this becomes more and more important)
- If you are interested in further speed from types, check out Cython

Introduction to Parallelism

What is it?

- Parallelism works by splitting up a series of problems into its components
- A problem is "parallelizable" when it can be seperated into smaller problems that do not rely on entirely on each other
- This is called "task parallelism"
- There are other types of parallelism that we won't discuss (too low level)

Threading v. Multiprocessing

Depends on abstraction

- There are two main types of parallelization routines: threading and multiprocessing
- A threaded process uses a single instance of the Python interpretter and sprouts many "threads" (more on this later)
- A multiprocesser routine sprouts many instances of the Python interpretter and runs them on seperate cores of the CPU
- The GIL blocks CPU based threading (there are advanced ways to side step this, but I have no idea how they work)

How does a computer work?

Just in case.

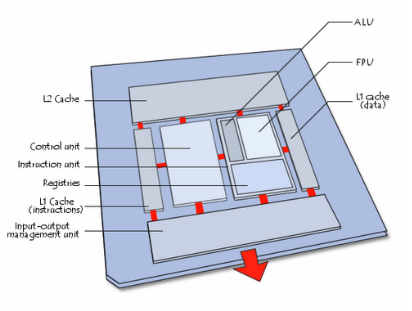
There are 3 main components (that we care about) to a computer:

- CPU
- Ram
- Hard disk

The CPU carries out calculation.
The RAM stores short term memory for immediate use.
The hard disk stores long term data.

Components of a CPU

The brains of the operation



Each "core" has the following

- Arithmetic Logic Unit (ALU) - carries out calculations
- Control Unit - executes instructions
- Registers - stores data on the device

A note on classes
One last thing to learn about Python!

To define our parallelized solution, we'll use a class. But what is a class?! Here is an example:

```
class a_test_class:
        """
        A useless class.
        """
        def __init_(self, args):
        # When you define an object it runs this code automatically
                self.something = args

        def a_useless_attribute(self):
                return self.something


obj = a_test_class(2)
obj.a_useless_attribute
```

```
Out: 2
```

How can we split up our problem?

Parellelizing!

- We can naturally split the estimator by splitting the sum
- Our problem then becomes to calculate a series of partial sums
- When completed we combine these sums into a result

Some useful vocab:

- worker - an instance of the interpreter
- queue - an object to store output
- process - the function to caluculate on a worker

A multiprocessing solution

A classic approach

```python
class cpu_parallel_monte_carlo:
    """
    A class containing CPU based parallelization of the serial monte c

    inputs:
        f     :    function; the moment to be estimated
        g     :    function; the distribution function of x
        M     :    scalar; the number of points to sample
        X     :    ndarray; bounds in R^n for the support

    """
    def __init__(self, f, g, M, X, workers):
        if M%workers is not 0:
            raise ValueError('The vector of points must '
                             'be evenly divisible'
                             ' by the number of workers.')
        # Initialize the object attributes
        self.f = f
        self.g = g
```

A multiprocessing solution II

A classic approach

```
        self.M = M
        self.X = X
        self.workers = workers
        self.queue = mp.Queue()
        self.N =  X.shape[0]
        self.points = np.random.rand(M, N)\
            *(np.array([X[:, 1], ]*M)
              - np.array([X[:, 0], ]*M))\
            +  np.array([X[:, 0], ]*M)
        self.V = np.prod(np.abs(X[:,1] - X[:,0]))
        # When the object is defined we go ahead and estimate.

        self.mc_mean()
        self.mc_var()


    def partial_sum(self, process):
        """
        A function that will calculate the partial sum for """
```

A multiprocessing solution III

A classic approach

```python
        """the monte carlo mean.
        inputs:
            process    :     int; the process number

        """
        # NOTE: This is the length of the slice of points.
        K = int(self.M/self.workers)
        partial_points = self.points[process*K:(process + 1)*K, :]

        # Calculate the sum as a loop
        sum1 = 0.0
        for i in range(0, K):
            sum1 += self.f(partial_points[i, :])\
                    * self.g(partial_points[i, :], self.X)

        self.queue.put(sum1)
```

A multiprocessing solution IV

A classic approach

```
    def partial_var(self, process):
        """
        A function that will calculate the partial sum for
        the monte carlo variance.

        inputs:
            process   :    int; the process number

        """
        K = int(self.M/self.workers)
        partial_points = self.points[process*K:(process + 1)*K, :]

        #Calculate the paritial sums as a loop
        var1 = 0.0
        sum1 = self.mean/self.V
        for i in range(0, K):
            var1 += (f(partial_points[i, :])*g(partial_points[i, :], X
                       - sum1)**2
        self.queue.put(var1)
```

A multiprocessing solution V

A classic approach

```
    def mc_mean(self):
        """
        A method to calculate the mean by parallel montecarlo.

        """
        processes = [mp.Process(target=self.partial_sum,
                                kwargs=dict(process=i))
                     for i in range(0, self.workers)]

        # Run the processes
        for p in processes:
            p.start()

        # When the processes are done, exit
        for p in processes:
            p.join()

        partial_sums = [self.queue.get() for p in processes]
        self.mean = sum(partial_sums)*self.V/self.M
```

A multiprocessing solution VI

A classic approach

```python
    def mc_var(self):
        """
        A method to calculate the variance by parallel montecarlo.

        """
        # Create a list of processes to run
        processes = [mp.Process(target=self.partial_var,
                                kwargs=dict(process=i))
                     for i in range(0, self.workers)]

        # Run the processes
        for p in processes:
            p.start()

        for p in processes:
            p.join()

        partial_sums = [self.queue.get() for p in processes]
        self.var = sum(partial_sums)*self.V**2/(self.M*(self.M - 1))
```

Review

What the heck was that?

- We just created a class for monte carlo simulation
- It contains methods for partial sums and multiprocessor monte carlo
- Upon definition it automatically runs the estimation
- The results are NOT returned
- Let's see if we get the same result!

A multiprocessing solution VII

Output

```python
workers = 2
M = 1000
N = 2
X = np.array((0*np.ones(N), 1*np.ones(N))).T

test = cpu_parallel_monte_carlo(f, g, M, X, workers)

print(monte_carlo_expectation_serial(f, g, M, X))
print(monte_carlo_expectation_numpy(f_vec, g_vec, M, X))
print((test.mean, test.var))
```

```
(0.6489662258370581, 0.00018176915454179822)
(0.67652119525959598, 0.0001748470226532903)
(0.64822527708, 0.000166748010145)
```

A multiprocessing solution VIII
Time

```
workers = 4
M = 1000
N = 2
X = np.array((0*np.ones(N), 1*np.ones(N))).T

# We can also time these three side by side
%timeit monte_carlo_expectation_serial(f, g, M, X)
%timeit monte_carlo_expectation_numpy(f_vec, g_vec, M, X)
%timeit cpu_parallel_monte_carlo(f, g, M, X, workers)
```

```
10 loops, best of 3: 19.2 ms per loop
1000 loops, best of 3: 718 $\mu$s per loop
10 loops, best of 3: 51.8 ms per loop
```

Still slower?!

What's going on?

- First, we pay a price in creating the class - fix this by removing the automated estimation (code in the notes)
- Second, we are using native python... not cool - fix this by replacing with Numpy (code in the notes)
- Third, the creation of the seperate Python instances is costly. Only when $M >> 10000$ do we see gains comparing parallel to serial (example code in notes)
- Since we're short on time, I'll simply give you the output of the comparisons, but check out the notes for the full code with descriptions and comments

A multiprocessing solution IX

Comparison

```
workers = 4
for M in [100, 1000, 10000]:
    print("M = %s" %M)
    test = cpu_parallel_monte_carlo(f, g, M, X, workers)
    test_numpy = cpu_parallel_monte_carlo_numpy(f_vec, g_vec, M, X, wor
    %timeit monte_carlo_expectation_serial(f, g, M, X)
    %timeit monte_carlo_expectation_numpy(f_vec, g_vec, M, X)
    %timeit (test.mc_mean(), test.mc_var())
    %timeit test_numpy.output()
    del test
    del test_numpy
```

A multiprocessing solution X

Comparison Output

```
M = 100
100 loops, best of 3: 1.9 ms per loop
10000 loops, best of 3: 119 $\mu$s per loop
10 loops, best of 3: 41.3 ms per loop
10 loops, best of 3: 42.2 ms per loop
M = 1000
100 loops, best of 3: 19 ms per loop
1000 loops, best of 3: 727 $\mu$s per loop
10 loops, best of 3: 48.9 ms per loop
10 loops, best of 3: 40.3 ms per loop
M = 10000
10 loops, best of 3: 192 ms per loop
100 loops, best of 3: 6.6 ms per loop
10 loops, best of 3: 137 ms per loop
10 loops, best of 3: 40.8 ms per loop
```

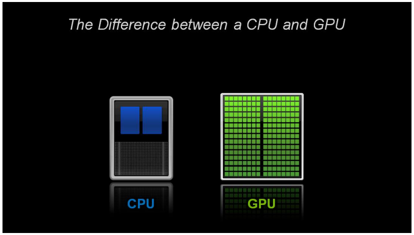
Only with $M >> 100000$ does multiprocessor Numpy beat serial Numpy

Multiprocessing
Taking stock

- Moving from serial to parallel is tough
- Learning curve is steep and coding is more involved
- Only useful for large scale problems
- Methodology similar every time, just have to understand parallelizing your problem

Introduction to GPU's

Finally!



- GPU stands for "Graphics Processing Unit"
- Created to handle complex graphics, which involve many simple calculations(eg translations or rotations)
- Removed all the nonsense from the processor and just crammed in ALUs

Introduction to GPU's II

What's so special?

- For $\approx$ \$100 can buy an Nvidia graphics card with 512 cores
- That means a cost of about 20 cents per core.
- Can run them in parallel
- 3 top-of-the-line cards (4992 cores each) can perform 746 times as many calculations per second as the Deep Blue supercomputer that beat Gary Kasparov (one game took only 19 moves)

Threading

A bunch of idiots

- As opposed to multiprocessing, GPUs use multithreading
- Have to think of the GPU as 512 infants who just learned addition and multiplication
- You have to do all the work
- Proper management necessary for efficiency gains

A simple example

PyCUDA

Elementwise multiplication kernel

```
mod = SourceModule("""
    __global__ void elementwise_multiply(float *a, float *b)
    {
        //the thread identifier to reference the original index
        const int t = threadIdx.x + blockIdx.x * 512 ;

        a[t] *= b[t];

    }
    """)
```

- Written in C
- For simple applications you can just reuse other kernels
- Instructions for individual threads

A simple example II
PyCUDA

```python
gpu_elementwise = mod.get_function("elementwise_multiply")

def super_fast_elementwise_multiply(a, b, NUMBER_OF_BLOCKS,
                                    THREADS_PER_BLOCK):
    """
    A GPU implementation of an elementwise multiplicaiton
    of two vectors.

    Inputs:

        a,b     :     ndarray

    """
    ### ALLOCATE MEMORY OIN THE CPU
    # Allocate memory to fill with solution
    c = np.zeros(a.shape[0]).astype(np.float32)
    ### ALLOCATE MEMORY ON THE GPU
    a_gpu = cuda.mem_alloc(a.nbytes)
    b_gpu = cuda.mem_alloc(b.nbytes)
```

A simple example III
PyCUDA

```
    ### TRANSFER DATA TO THE DEVICE
    cuda.memcpy_htod(a_gpu, a)
    cuda.memcpy_htod(b_gpu, b)

    ### EXECUTE THE FUNCTION
    gpu_elementwise(a_gpu, b_gpu,
                    block=(THREADS_PER_BLOCK, 1, 1),
                    grid=(NUMBER_OF_BLOCKS, 1))

    ### RETRIEVE SOLUTION
    cuda.memcpy_dtoh(c, a_gpu)

    # Free up the memory
    del a_gpu
    del b_gpu

    return c
```

A simple example IV
PyCUDA

```
# How big do you want your vectors?  They will be of length M*512
M = 10
M *= 512

### DEFINE THE THREAD STRUCTURE
# Define GPU size parameters
NUMBER_OF_BLOCKS = int(M/512)
THREADS_PER_BLOCK = 512

# Generate some data
A = np.ones((M)).astype(np.float32)*3
B = np.ones((M)).astype(np.float32)*2

# Calculate the elementwise multiplication
C = super_fast_elementwise_multiply(A, B, NUMBER_OF_BLOCKS,
                                   THREADS_PER_BLOCK)
print(C)
print(A*B)
```

A simple example V
PyCUDA

```
Out:
[ 6.  6.  6. ...,  6.  6.  6.]
[ 6.  6.  6. ...,  6.  6.  6.]
```

- Every PyCUDA function follows a similar structure $\Rightarrow$ Re-use old kernels
- If need something special, combine old kernels
- Need to define the thread structure appropriately
- Be VERY careful for the thread id's. These are very important.

A Simple Example VI

Speed

```
for M in [1000, 10000, 100000, 200000]:
    M *= 512

    NUMBER_OF_BLOCKS = int(M/512)
    THREADS_PER_BLOCK = 512

    # Generate some data
    a = np.ones((M)).astype(np.float32)*3
    b = np.ones((M)).astype(np.float32)*2

    print("\nNumber of Elements: %s" % M)
    %timeit a*b
    %timeit super_fast_elementwise_multiply(a, b,
                                            NUMBER_OF_BLOCKS,
                                            THREADS_PER_BLOCK)


    del a
    del b
```

A Simple Example VII

Speed II

```
Number of Elements: 512000
1000 loops, best of 3: 426 s per loop
100 loops, best of 3: 2.79 ms per loop

Number of Elements: 5120000
100 loops, best of 3: 5.76 ms per loop
100 loops, best of 3: 17 ms per loop

Number of Elements: 51200000
10 loops, best of 3: 59.7 ms per loop
10 loops, best of 3: 169 ms per loop

Number of Elements: 102400000
10 loops, best of 3: 121 ms per loop
1 loops, best of 3: 337 ms per loop
```

Slow again!?

This class is bogus.

- As with other methods we've seen, we aren't getting speed
- Typical problem: If this were truly faster, we wouldn't use NumPy!
- High latency causes slowdown on data transfer
- "Latency" (in a network sense) refers to the speed with which messages/data are sent
- To minimize the price paid by data transfer, do more calculation on the device

For exposition sake, I'll show two GPU solutions, one using NumPy to calculate the L^2 norm and one calculating it directly on the GPU using a 2-D thread structure. Finally, we'll discuss why this doesn't work and I'll attempt (and fail) to fix it.

GPU Monte Carlo

Far too much work for what you get.

((Go to notebook. This is too long for slides...))