MA553 – High Performance Scientific Computing

Fall 2022, Dr. Khanal

Final Project

Quickly Computing the Mandelbrot Numbers

Tyler Procko

Preface

I am deviating from the project final report guidelines because my initial proposal was infeasible, and the project had to change. I give a brief background on why the initial proposal was abandoned, and then resume the normal report format for my current project, which was implemented successfully.

Also, this project should be considered with respect to the Windows operating system. Against all advice, I did manage to get *NumPy*, *Numexpr*, *Numba*, *Cython* and *multiprocessing* working on Windows.

The outline of this report is as follows:

1.1: Initial Proposal

Background

The proposal initially generated for this class project was generated on October 4th of this year. The proposal entailed the optimization of two vectorization techniques, namely, Term Frequency-Inverse Document Frequency (TF-IDF) and Naïve Bayes, used in Natural Language Processing (NLP). NLP is a subset of Machine Learning (ML) that has the purview of deriving understanding from natural, human language, generally in the form of written text.

NLP begins with a representation and vectorization of the text. TF-IDF, for example, is used mostly for categorizing documents or pulling out keywords, as it takes into account the novelty, or uniqueness, of words over a set of documents. Its calculation is as follows:

$TF = \frac{num.occurrences\ of\ the\ word\ in\ the\ document}{num.words\ in\ the\ document}$	$IDF = \log\left(\frac{num.\ documents}{num.\ documents\ containing\ the\ word}\right)$
$TF\_IDF = TF * IDF$	

These calculations are not overly complicated, but for larger datasets where each document is a *sentence,* the number of division operations, in addition to logarithmic calculations, can be quite costly. It was proposed that the *NumPy* and *Numexpr* libraries could be used heavily to optimize these operations, in addition to chunking up the text dataset for use with *multiprocessing*.

Additionally, I intended to attempt an optimization of the vectorization algorithm, Naïve Bayes. Naïve Bayes is a vectorization algorithm used primarily for binary classification tasks, like sentiment analysis. For a given document, its log-likelihood is calculated, using the following formula:

$$\sum_{i=1}^{n} log\frac{P(w_i \mid pos)}{P(w_i \mid neg)}$$

The numerator of the equation is the probability a word is positive given the total set of words, and the denominator is the probability a word is negative given the total set of words.

Implementation

Python's Natural *Language Toolkit (NLTK)* was used for its Tweets dataset, which contains 10,000 Tweets for use in testing NLP approaches. The *sklearn* library, which has a TF-IDF function built-in, was used as a baseline. The TF-IDF implementation in pure Python was successful.

```
def tf(wordDict, bow):
    tfDict={}
    numWordsInDoc=len(bow)
    for word,numOccurrencesInDoc in wordDict.items():
        tfDict[word]=numOccurrencesInDoc/numWordsInDoc
    return tfDict

def idf(docList):
    idfDict={}
    docCount=len(docList)
    idfDict=dict.fromkeys(docList[0].keys(),0)
    for doc in docList:
        for word,val in doc.items():
            if val>0:
                idfDict[word]+=1
    for word,numDocsContainingWord in idfDict.items():
        idfDict[word]=math.log10(docCount/numDocsContainingWord)
    return idfDict

def tfidf(tfBow,idfs):
    tfidf={}
    for word,val in tfBow.items():
        tfidf[word]=val*idfs[word]
    return tfidf
```

The code for TF-IDF, encapsulated in functions.

For a very small sample size of 3 Tweets, my pure Python code was significantly faster than *sklearn*.

```
TF-IDF pure python: 0.00012080000000000000 s
TF-IDF sklearn: 0.00133320000000000010 s
```

However, when scaling the corpus size up to the full 10,000 *NLTK* Tweets, *sklearn* was far faster than pure Python.

```
TF-IDF 10k tweets pure python: 92.33102979999999604388 s
TF-IDF 10k tweets sklearn: 0.35004790000000002292 s
```

Hence, my pure Python code would provide a low-end baseline of worst-case speed, and *sklearn* would provide a best-case speed to compare all optimization attempts against.

1.2: Issues with Proposal

Not Enough Math Calculations

The principal issue with the original proposal was the lack of numerical operations. Much of the work had to do with string manipulation and data structure linking. Nearly all NLP Python code makes use of dictionaries, or hashtables. This is a universal truth. Dictionaries allow very easy linking between document, word and vector representation. While these are, relatively speaking, fast because they can make use of lookup functions, they cannot make use of *NumPy*, Numexpr or, as I found out, Numba. Much of my time was spent attempting to map my string dictionaries into *NumPy* arrays, and my code blew up in complexity as a result. Consider the dictionary, *d*, containing a document ID, word and its vector representation of n dimensions:

$$d = \{document\_id : \{word : [\# \dots n]\}\}$$

In this simple example we have a dictionary containing a dictionary as value, whose value is a large one-dimensional array containing the word's vector representation. Optimizing this with *NumPy* is not possible unless the dictionaries are broken up into arrays and related on their indices, which I attempted. But the code became difficult to manage.

The only obvious optimization technique was using multiple threads, through *multiprocessing*, or by annotating the code with *Numba's* decorators and hoping the magic worked. As was discussed, no performance gain was perceptible. The code itself was too complex, and also unfit for the mathematical optimization techniques learned in class. It is possible that *tensorflow* or *CUDA* could be used to optimize the code, but these are very complicated techniques that would require more time to learn than to implement for TF-IDF code. I did attempt to use the alternative Python interpreter, *PyPy*, to run my code, as it uses a just-in-time (JIT) compiler by default, but the code would have to be changed considerably. For instance, *NumPy* was not working for me, as *PyPy* expects *NumPyPy*; or, I would have to use a virtual environment manager like *Conda* or *PyEnv* so I would not break my Python installation and potentially render my code useless. Also, I encountered issues just trying to install *PyPy*, so this was abandoned.

The *multiprocessing* and *mpi4py* libraries could certainly be used here by chunking up the dictionaries, but to me, these are last-resort, all-out speed measures only undertaken when everything else has been done. If *NumPy* cannot even be used properly, going straight to parallelization does not seem appropriate.

Scalability

Also, with the dataset being used (NLTK Tweets), I only had a corpus of size 10,000. In other words: my limit for testing was 10,000 unique documents. To get more data, I would have to seek out other datasets and my code would have to change considerably as a result. This was unacceptable. The new project needed to be infinitely scalable so I would be able to test with progressively larger values to see where the different optimization techniques do better than others.

Visualizations

Moreover, visualizations of the initial proposal would be trivial. The output of TF-IDF is words and their vector representations. The most obvious visualization would be a scatter plot to show word clusters, but for this course, that is not a very interesting or challenging visualization.

Acceptance

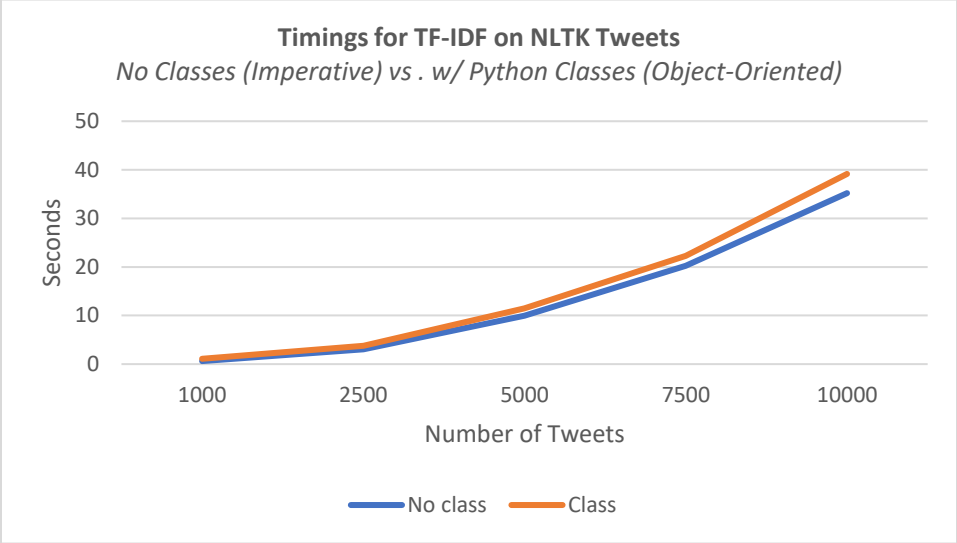
I worked on the TF-IDF code, attempting to bring the data into *NumPy* arrays, but, as everything relied on strings up-front, it was difficult to manage. There is very little mathematical computation until the latter stages of the code execution. On December 2nd, this initial project idea was abandoned in favor of a simpler, deeply mathematic calculation that could make use of the optimization techniques learned in class.

1.3: Useful Observations Gained

This initial effort was not without benefit, however. The code was written to be a very accommodating test suite. The TF-IDF code was put into its own Python class, and the methods named, such that the *getattr()* function could be used to pick them out individually, or iterate through them in a loop. This code was very clean. In order to allow profiling using *line_profiler*, I pulled the *tf()*, *idf()* and *tfidf()* functions back into the code body for each optimization function.

What was learned from this is that using Python classes slows down execution time considerably. My initial code even had function calls to *tf()*, *idf()* and *tfidf()* in the same file, and the execution time was much

faster than when using a class. Also, I was careful to not include Python's __init()__ method in the timing. Despite the code being literally identical aside from the classing, placing the TF-IDF code into a class, and accessing it by an object, slowed down execution.



There is a clear trend here. Given more data, it is probable that this gap will continue to widen. Having learned this, I made it a requirement of the next project to use no Python classes, as I wanted as much speed as possible.

With my TF-IDF code, I also produced a testing suite, so translating to my new problem and testing it thoroughly is not as difficult, as, with the TF-IDF Code, I had to do it from scratch.

1.4: Shift of Focus

Considering the dilemma encountered, and with a little under two weeks remaining before the project was due, I researched the Web for common math problems in computational optimization work. Many problems were encountered, including:

- Fast Fourier transform
- Discrete Laplacian edge detection (for images)
- The traditional sorting algorithms (e.g., bubble, selection, insertion, merge, heap, quick, etc.)

However, what piqued my interest was an image of a repeating pattern called a fractal. Specifically, it was the Julia set, named after the French mathematician, Gaston Julia. This led me to the Julia programming language, which is purpose-built for high-performance computing, similar to Fortran. Julia is very interesting because its code looks very much like Python but is as fast, or faster than, Fortran. It has operators similar to Python's *NumPy* library that are built into the language. One of the Julia group's language benchmarks is the computation of a set of numbers called the Mandelbrot set, which is quite famous. Searching the Web further, I found many resources for computing the Mandelbrot set in Python, which was a welcome change from the desert that is the NLP optimization Web search space. So, I had a good basis for the new project, as many have implemented the Mandelbrot set in various languages and with varying degrees of optimization, so I now had some things to learn from. As it was with the initial NLP proposal, I could not locate anything helpful about optimizing dictionary-heavy code.

2.1: The Mandelbrot Set: Introduction and Problem Description

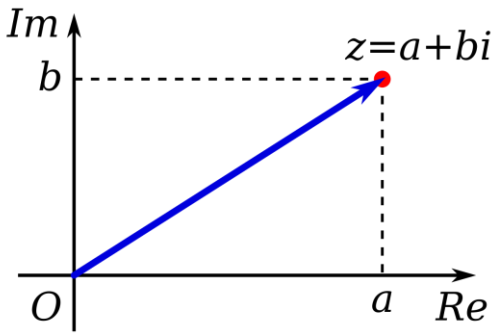
The Mandelbrot set is a set of complex numbers, whose boundary forms an interesting pattern when depicted on the complex plane. A complex number, *c*, is of the form:

$$c = a + bi$$

Where *a* and *b* are real numbers, and *i* is an imaginary number. Complex numbers extend real numbers with a specific element, *i*, called the imaginary unit. The imaginary unit satisfies the equation:

$$i^2 = -1$$

Because no real number can ever satisfy this equation, *i* was called the imaginary number by René Descartes. What is important to note about complex numbers is that they reside in a typical Euclidian vector space of two dimensions, called the complex plane. Complex numbers can be represented as a pair of numbers *(a, b)*, which can be plotted on the complex plane.



A complex number, (a,b). Re is the real axis, Im is the imaginary axis.

The Mandelbrot set (or Mandelbrot sequence) is a set of complex numbers, *c*, for which an infinite sequence of numbers, *z*, remains bounded. The calculation for determining if a complex number is part of the Mandelbrot set is as follows:

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

To know whether some complex number, *c*, belongs to the Mandelbrot set, it is given to the formula above. The complex number *c* now remains constant as the last element, z_n is squared and added to *c* to get the next number, in a loop. The complex number *c* is part of the Mandelbrot set if, for all $n > 0$, the absolute value of z_n remains bounded, i.e., does not diverge to infinity. There is a limit that the magnitude of each complex number in *z* never exceeds: this magnitude is the divergence point. The number of iterations it takes for z to diverge is referred to as the escape count. On the complex plane, the Mandelbrot set fits in a circle of radius 2 (the radius of divergence), so any number outside of it means it is not part of the Mandelbrot set, and computation can cease. So, in code, the number 2 is used as the limit for stopping the Mandelbrot inclusion calculation for a given z_n.

2.2: Baseline: Pure Python Implementation – 3.6 seconds

I am not showing the visualization and testing code in this report as it is rather complicated and does not lend to an understanding of the problem. If the reader is interested, see the Appendix section. Every value of c passed into a Mandelbrot implementation is part of a set of values, defined by the user, that are generated in a double for loop. In my testing, I used:

Code / Graphical Setting	Mathematical Equivalent	Value
Minimum X	Smallest real value a of complex $c = (a, b)$	-2.0
Maximum X	Largest real value a of complex $c = (a, b)$	0.5
Minimum Y	Smallest imaginary value b of complex $c = (a, b)$	-1.2
Maximum Y	Largest imaginary value b of complex $c = (a, b)$	-1.2
Image Width	Number of real values	1000
Image Height	Number of imaginary values	1000
Maximum Iterations	Limit for c values that do not diverge to infinity so code stops	80

Table 2.2a: Initial values used in the computations.

So, in code, there are 1000 real values between -2.0 and 0.5, and 1000 imaginary values between -1.2 and 1.2. I.e., there are *1,000,000 complex numbers* passed into the Mandelbrot implementations every execution. This is not a trivial calculation. The *NumPy.linspace()* function is used to generate arrays of real and imaginary numbers, and Python's *complex()* function to generate complex numbers from them. Every time the Mandelbrot check is performed, this is for "one pixel" of the image. In pure Python, this looks like:

```
import numpy as np

def mandelbrot_purepython(c, maxIterations):
  z = 0
  for n in range(maxIterations):
    if abs(z) > 2:
      return n
    z = z ** 2 + c
  return maxIterations

def mandelbrot_set_purepython(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums = np.linspace(xMin, xMax, width)
  imagNums = np.linspace(yMin, yMax, height)
  escapeCounts = []
  for r in realNums:
    for i in imagNums:
      escapeCounts.append(mandelbrot_purepython(complex(r, i), maxIterations))
  return (realNums, imagNums, escapeCounts)
```

The initial, pure Python implementation of the Mandelbrot set calculation.

Initial timings for this code were approximately 5.3 seconds. Before trying to optimize with *NumPy*, I tried to optimize this code itself, and I found an interesting quirk of Python. Replacing the line:

$$z = z ** 2 + c$$

with:

$$z = z * z + c$$

… improved speed by up to 40%, to 3.6 seconds! This was an interesting optimization that was carried over to all the successive optimizations.

2.3: Profiling

Before moving on to further optimizations, profiling must be done. Below are the results of running the *line_profiler* library against the two pure Python Mandelbrot functions.

```
Function: mandelbrot_purepython at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    10                                           def mandelbrot_purepython(c, maxIterations):
    11   1000000    2143867.0      2.1      1.0    z = 0                           # z always starts as 0
    12  25871042   59485330.0      2.3     27.5    for n in range(maxIterations):  # When testing a complex, c, if we hit the max numb
    13  25130132   83202494.0      3.3     38.5      if abs(z) > 2:                # 2 is the radius of the Mandelbrot set circle (div
    14    740910    1849150.0      2.5      0.9        return n                    # If we hit this limit, kick out
    15  25130132   68641877.0      2.7     31.8      z = z * z + c                 # Mandelbrot (z ** 2 is MUCH slower; about 40%)
    16    259090     598708.0      2.3      0.3    return maxIterations            # Otherwise, return max iterations

Total time: 37.9639 s
File: E:\OneDrive - Embry-Riddle Aeronautical University\college\2022zfall\MA453\project2\mandelbrot1_purepython.py
Function: mandelbrot_set_purepython at line 18

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    18                                           def mandelbrot_set_purepython(xMin, xMax, yMin, yMax, width, height, maxIterations):
    19         1        973.0    973.0      0.0    realNums = np.linspace(xMin, xMax, width)
    20         1        435.0    435.0      0.0    imagNums = np.linspace(yMin, yMax, height)
    21         1          2.0      2.0      0.0    escapeCounts = []
    22      1000       4391.0      4.4      0.0    for r in realNums:
    23   1000000    3320744.0      3.3      0.9      for i in imagNums:
    24                                                 # Uses complex(), which is the same as real + imaginary * 1j
    25   1000000  376312281.0    376.3     99.1        escapeCounts.append(mandelbrot_purepython(complex(r, i), maxIterations))
    26         1          5.0      5.0      0.0    return (realNums, imagNums, escapeCounts)
```

The results of using line_profiler on the pure Python Mandelbrot implementation.

Looking at the results of the profiling, we can, for now, ignore the second function, as there is not much optimization that can be done yet, short of working with more *NumPy* arrays. The greater interest lies in the first function, which actually performs the Mandelbrot checks for all given *c* values. Interestingly, the *if statement* takes up most of the computation time! This is followed by the calculation for updating *z*, and then the for loop itself. This is perhaps the most insightful discovery and will be used to optimize the code throughout this project. There are various ways to improve the speed of the absolute value function, which actually is very computationally inefficient. Moreover, as will be discussed, the calculation to update *z* can be optimized as well.

In the second function, the best optimization that is apparent is using a *NumPy* array in place of the Python list for the escapeCounts variable. If this is done, then there will be no need for the double for loop, as *NumPy* can perform operations over entire arrays at once. There is also some less obvious optimization that can be performed on the instantiation of Python complexes (through *complex()*); these will be discussed later.

2.4: NumPy Optimization – 1.23 seconds

```
import numpy as np

def mandelbrot_numpy(c, maxIterations):
  escapeCount = np.resize(np.array(0,), c.shape)
  z = np.zeros(c.shape, np.complex64)

  for iteration in range(maxIterations):
    z = z * z + c
    done    = np.greater(abs(z), 2.0)
    c       = np.where(done, 0 + 0j, c)
    z       = np.where(done, 0 + 0j, z)
    escapeCount = np.where(done, iteration, escapeCount)
  return escapeCount

def mandelbrot_set_numpy(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums     = np.linspace(xMin, xMax, width, dtype = np.float32)
  imagNums     = np.linspace(yMin, yMax, height, dtype = np.float32)
  complexNums  = np.ravel(realNums + imagNums[:,None] * 1j)
  escapeCounts = mandelbrot_numpy(complexNums, maxIterations)
  escapeCounts = escapeCounts.reshape((width, height))
  return (realNums, imagNums, escapeCounts.T)
```

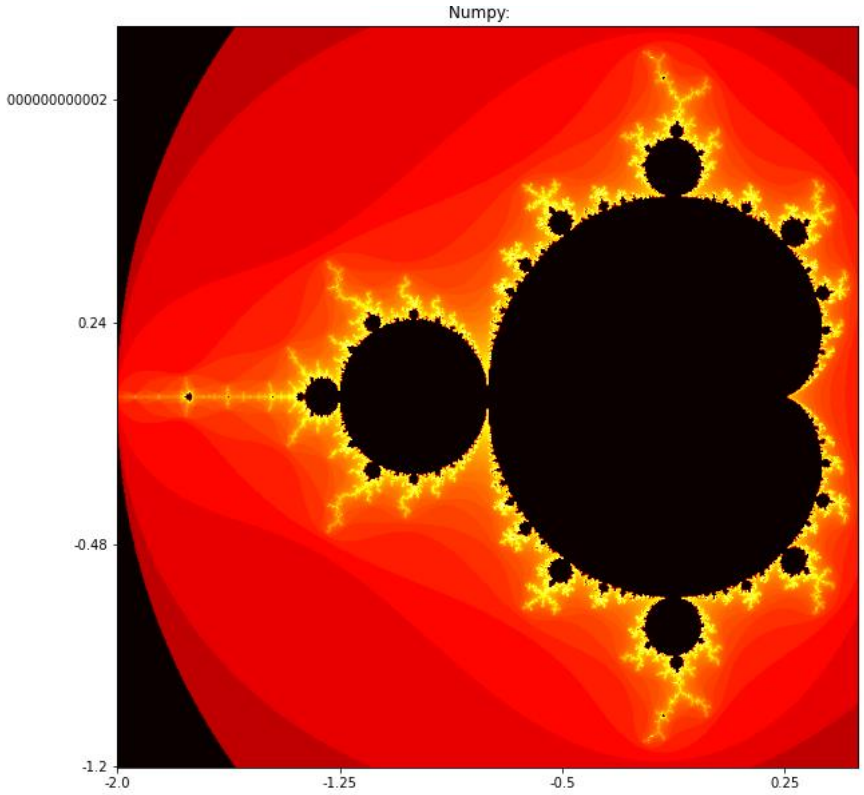
The initial Numpy implementation.

The *NumPy* optimized code makes use of the *NumPy.where()* function, which is like an if statement but faster. Other than that, it is much the same as the pure Python code, except, notice, the input complex number does not use Python's *complex()* function, but is built literally, using the form *real + imaginary * 1j*. This is defined in Python's documentation as a proper way to construct a complex number. In my tests this seemed to be slightly faster (but not substantially so). Also, I had issues using *complex()* in the *ravel()* function, and this seemed to work better. Anyhow, *NumPy's* power lies in array operations on an arbitrary number of elements, so building out complexes using *ravel()* is more ideal than using *complex()* in a double for loop like the pure Python code.

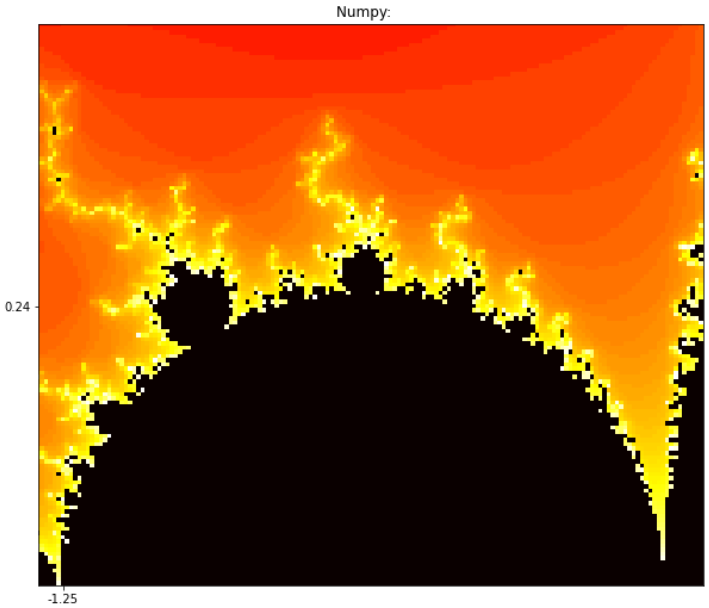
Interestingly, again, I found another quirk of Python. The timings for this code were approximately 1.85 seconds. Adding *dtype* declarations of *np.float32* to the *np.linspace()* commands improved timings to 1.23 seconds, an improvement of 40%!

```
realNums = np.linspace(xMin, xMax, width, dtype = np.float32)
imagNums = np.linspace(yMin, yMax, height, dtype = np.float32)
```

This is because the default datatype is *np.float64*, which takes longer in computations. In my visualization tests, there was no attendant loss of visual clarity with this simple optimization. Below is provided the *NumPy* and *Matplotlib* visualization of the Mandelbrot set!

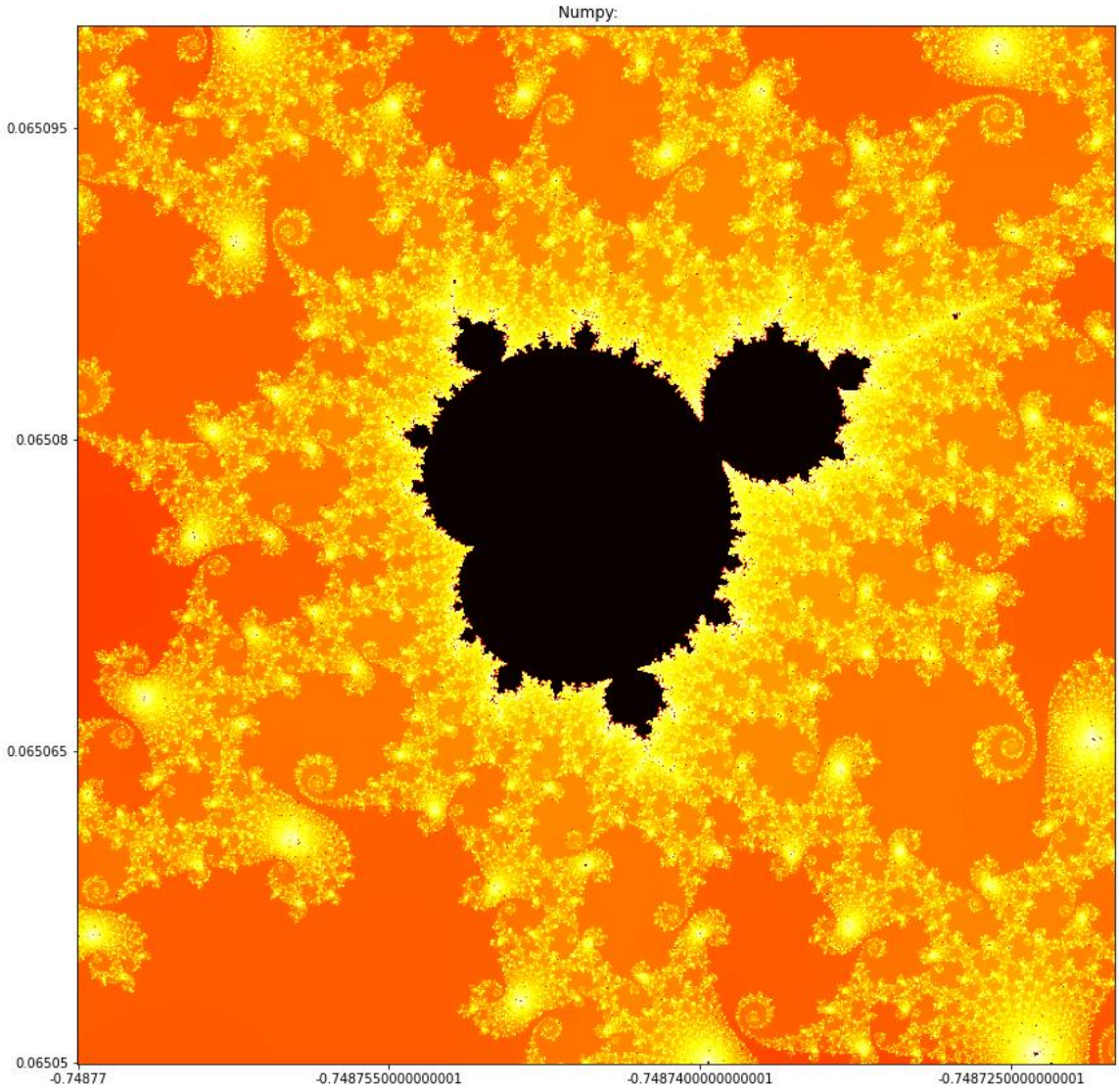


Matplotlib visualization of the Mandelbrot set, Numpy version.



Zoomed in portion of the Mandelbrot border. The code allows adjustment of image resolution; this is not discussed in this report to any great extent.

Changing the input values, we can get totally different (and beautiful) Mandelbrot visual:



Tweaking values for Mandelbrot input results in vastly different visuals.

2.5: Better NumPy Optimization - .85 seconds

```python
import numpy as np

def mandelbrot_numpy_better(c, maxIterations):
  escapeCount = np.zeros(c.shape)
  z = np.zeros(c.shape, np.complex64)

  for iteration in range(maxIterations):
    notFinished = np.less(z.real * z.real + z.imag * z.imag, 4.0)
    escapeCount[notFinished] = iteration
    z[notFinished] = z[notFinished] * z[notFinished] + c[notFinished]

  escapeCount[escapeCount == maxIterations - 1] = 0
  return escapeCount

def mandelbrot_set_numpy_better(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums = np.linspace(xMin, xMax, width, dtype = np.float32)
  imagNums = np.linspace(yMin, yMax, height, dtype = np.float32)
  complexNums = realNums + imagNums[:, None] * 1j
  escapeCounts = mandelbrot_numpy_better(complexNums, maxIterations)
  return (realNums, imagNums, escapeCounts.T)
```

Better NumPy implementation.

The first optimization performed is getting rid of the *NumPy ravel()* call and directly using *NumPy* to add the real and imaginary arrays into a complex array. This is much faster than the old *NumPy* code. In the Mandelbrot function itself, the *resize()* function is removed and the *output* array is directly asserted to be a zero *NumPy* array of shape *c*.

In the iteration of *z*, we speed up the *if (abs(z) > 2* check. This is perhaps the greatest breakthrough, derived from this article:

- https://medium.com/swlh/visualizing-the-mandelbrot-set-using-python-50-lines-f6aa5a05cf0f

… which allowed the diverging check to be expressed using the Pythagorean theorem. Because *c* diverges when the distance from the origin is greater than 2, the Pythagorean theorem can be used to define this as:

$$x^2 + y^2 > 4$$

The *NumPy.less()* function is used to track if the escape point has been reached. In this implementation, *Boolean indexing* is used heavily. This is because *NumPy* allows Boolean indexing to select elements of an array based on a condition. Since this function only needs to return the iteration numbers that did not diverge, *Boolean indexing* is sensible.

Initial timings for this implementation are at .85 seconds, which is an increase in speed of about 38% over the original Numpy code.

2.6: Numba Optimization - .24 seconds

Numba is… essentially magic. It takes very little effort and gives massive benefit through the JIT compiler.

```python
import numpy as np
from numba import jit

@jit
def mandelbrot_numba(c, maxIterations):
  z = 0
  for n in range(maxIterations):
    if z.real * z.real + z.imag * z.imag > 4.0:
      return n
    z = z * z + c
  return 0

@jit
def mandelbrot_set_numba(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums = np.linspace(xMin, xMax, width)    # Declaring datatypes here breaks the code :)
  imagNums = np.linspace(yMin, yMax, height)   # ^^^
  escapeCounts = np.empty((width, height))
  for i in range(width):
    for j in range(height):
      escapeCounts[i, j] = mandelbrot_numba(realNums[i] + imagNums[j] * 1j, maxIterations)
  return (realNums, imagNums, escapeCounts)
```

Numba implementation.

This *Numba* code is very similar to the better *NumPy* implementation and the pure Python code. Very little was changed. The only observation made was that declaring datatypes in the *NumPy.linspace()* like before broke the code.

Reported timings for this code are at .24 seconds, which is a 112% increase over the better *NumPy* implementation.

2.7: Better Numba Optimization - .14 seconds

```python
import numpy as np
from numba import jit

@jit
def mandelbrot_numba_better(cReal, cImag, maxIterations):
  real = cReal
  imag = cImag
  for n in range(maxIterations):
    realNew = real * real
    imagNew = imag * imag
    if realNew + imagNew > 4.0:
      return n
    imag = 2 * real * imag + cImag
    real = realNew - imagNew + cReal
  return 0

@jit
def mandelbrot_set_numba_better(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums = np.linspace(xMin, xMax, width)    # Declaring datatypes here breaks the code :)
  imagNums = np.linspace(yMin, yMax, height)  # ^^^
  complexNums = np.empty((width, height))
  for i in range(width):
    for j in range(height):
      complexNums[i, j] = mandelbrot_numba_better(realNums[i], imagNums[j], maxIterations)
  return (realNums, imagNums, complexNums)
```

Improved Numba implementation.

This implementation does very little differently than the last, save for no longer using Python's built-in complex type, or the standard form, *real + imaginary * 1j*. Instead, the real and imaginary numbers are passed into the Mandelbrot function as entire *NumPy* arrays. This is done so the complex number calculations can be done literally.

First, the squares of each real and imaginary number are calculated and stored. Pre-calculating them seemed to save some computation time, though not much. Then, in the if statement, the Pythagorean theorem trick is used again. The last two lines before the return statement have to do with complex number math. Take some complex number, $a + bi$. Squaring it produces $(a^2 - b^2) + (2ab)i$. So, we re-calculate z (here, represented by *imag* and *real*), where:

$$(a^2 - b^2) \equiv realNew - imagNew + cReal$$

$$(2ab)i \equiv 2 * real * imag + cImag$$

This is a really fast way to compute $z = z * z + c$. Timings for this are at .14 seconds, an improvement of 52% over the regular *Numba* implementation.

2.8: Betterer (guvectorized) Numba Optimization - .01 seconds

```
import numpy as np
from numba import jit, vectorize, guvectorize, complex64, int32

@jit(int32(complex64, int32))
def mandelbrot_numba_betterer3(c, maxIterations):
  realNew = 0
  real  = 0
  imag  = 0
  for n in range(maxIterations):
    realNew = real * real - imag * imag + c.real
    imag = 2 * real * imag + c.imag
    real = realNew
    if real * real + imag * imag > 4.0:
      return n
  return 0

@guvectorize([(complex64[:], int32[:], int32[:])], '(n),()->(n)', target = 'parallel')
def mandelbrot_numba_betterer2(c, maxIterations, output):
  maxIterationsTemp = maxIterations[0]
  for i in range(c.shape[0]):
    output[i] = mandelbrot_numba_betterer3(c[i], maxIterationsTemp)

def mandelbrot_set_numba_betterer(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums = np.linspace(xMin, xMax, width, dtype = np.float32)
  imagNums = np.linspace(yMin, yMax, height, dtype = np.float32)
  complexNums = realNums + imagNums[:, None] * 1j
  escapeCounts = mandelbrot_numba_betterer2(complexNums, maxIterations)
  return (realNums, imagNums, escapeCounts.T)
```

Further improved Numba implementation.

This code is incredibly fast. I cannot pretend to fully know how it works, but I can describe most of it. The actual Mandelbrot function is rather normal. Some modifications were done to the last Numba code, but otherwise, it is mostly the same. The *@guvectorize* decorator is *Numba's* generalized universal function decorator, built specifically for high-dimensional arrays. There is a temporary input to the *@guvectorize* decorated function to convert an integer input into an array, because this is what *@guvectorize* expects. Similar to *Cython*, explicitly declaring datatypes in the decorators results in a large speedup.

Interestingly, *@guvectorize* takes in an input/output *layout* string, of the form *'in -> out'*. In the case of this code, this is *'(n),()->(n)'*, which denotes the input is an n-element 1D array and a scalar, and the output is an n-element 1D array. The last argument is *target*, which was set to *'parallel'* to make use of multiple cores. The default *target* for *@guvectorize* is *'cpu'*, which only makes use of one core.

So, it is obvious to state that the Mandelbrot problem is embarrassingly parallel. Coupling parallelization with a JIT compiler makes for an incredibly fast program.

Timings for this code are .01 seconds, which equates to a speedup of 173% over the last *Numba* implementation.

2.9: Numexpr Optimization - .45 seconds

```python
import numpy as np, numexpr as ne

def mandelbrot_numexpr(c, maxIterations):
  escapeCount = np.zeros(c.shape)
  z = np.zeros(c.shape, np.complex64)

  for iteration in range(maxIterations):
    notFinished = ne.evaluate('z.real * z.real + z.imag * z.imag < 4.0')
    escapeCount[notFinished] = iteration
    z = ne.evaluate('where(notFinished,z ** 2 + c, z)')

  escapeCount[escapeCount == maxIterations - 1] = 0
  return escapeCount

def mandelbrot_set_numexpr(xMin, xMax, yMin, yMax, width, height, maxIterations):
  realNums = np.linspace(xMin, xMax, width, dtype = np.float32)
  imagNums = np.linspace(yMin, yMax, height, dtype = np.float32)
  complexNums  = np.ravel(realNums + imagNums[:,None] * 1j)
  escapeCounts  = mandelbrot_numexpr(complexNums, maxIterations)
  escapeCounts  = escapeCounts.reshape((width, height))
  return (realNums, imagNums, escapeCounts.T)
```

Numexpr implementation.

This is very similar to the original *Numpy* code. Using *Numexpr* allows very complex *NumPy* expressions to be calculated extremely fast. One observation is that declaring the datatypes in the *Numpy.linspace()* function calls like before had very little effect on speed. It is not the fastest and took considerable effort in translating *NumPy* code to string expressions, so it is not the most ideal method, but it is still fast.

Initial timings for this code had .45 seconds, which is a 170% increase from the original, pure Python code.

2.10: Cython Optimization - .06 seconds

```
import cython
import numpy as np

cdef int mandelbrot_cython_func(double cReal, double cImaginary, int maxIterations):
  cdef double real = cReal
  cdef double imaginary = cImaginary
  cdef double real2
  cdef double imaginary2
  cdef int n

  for n in range(maxIterations):
    real2 = real * real
    imaginary2 = imaginary * imaginary
    if real2 + imaginary2 > 4.0:
      return n
    imaginary = 2 * real * imaginary + cImaginary
    real = real2 - imaginary2 + cReal;
  return 0

@cython.cdivision(True)
@cython.nonecheck(False)
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef mandelbrot_set_cython_func(double xMin, double xMax, double yMin, double yMax, int width, int height, int maxIterations):
  cdef double[:] realNums = np.linspace(xMin, xMax, width)
  cdef double[:] imagNums = np.linspace(yMin, yMax, height)
  cdef int[:,:] escapeCounts = np.empty((width, height), np.int)
  cdef int i, j

  for i in range(width):
    for j in range(height):
      escapeCounts[i, j] = mandelbrot_cython_func(realNums[i], imagNums[j], maxIterations)

  return (realNums, imagNums, escapeCounts)
```

The Cython implementation (.pyx file).

The *Cython* optimization went well and was the simplest to do, as I had already done the *NumPy* and *Numba* legwork. The primary issue I encountered initially was in trying to *Cython-ize* my pure Python code. But the glaring issue was that I passed Python's complex datatype to the Mandelbrot calculation function, so that would not work. In my 2nd *Numba* implementation, I instead passed an array for the real numbers and an array for the imaginary numbers. I did the same here and it worked very well. Defining all datatypes with *cdef* is essential for speed in *Cython*.

The *Cython* decorators, *cdivision, nonecheck, boundscheck and wraparound* were all set to allow the fastest execution time.

I also attempted to do parallel *Cython*, using *prange* in the for loops. But, as is common, I got *Global Interpreter Lock (GIL)* errors. I tried appending *nogil* to the functions but got more *GIL* errors. As the current *Cython* code was already exceptionally fast, I abandoned trying to use parallel *Cython*, but it should be noted that it can improve speed, so perhaps in the future it can be investigated.

Initial timings with this *Cython* code came out to .06 seconds, which is nearly as fast as the best *Numba* code!

2.11: Multiprocessing Optimization – 1.34 seconds

All the codes thus far use a double for loop to iteratively generate a complex number (a pixel on the image), which is then checked as being part of the Mandelbrot set, where its escape count is returned and put into a list. The initial *multiprocessing* optimization made use of list comprehension with the *map()* function. In essence, instead of checking every single *c* value through loops, a list of arguments for these checks was built in advance, and the *map()* function used to call the Mandelbrot function on every element in the list. In code, this:

```
escapeCounts = []
for r in realNums:
  for i in imagNums:
    escapeCounts.append(mandelbrot_purepython(complex(r, i), maxIterations))
```

… could become:

```
p = Pool()
complexNums = [complex(real, imag) for imag in imagNums for real in realNums]
escapeCounts = p.map(mandelbrot, complexNums)
escapeCounts = np.reshape(escapeCounts, (width, height))
```

In my tests, the *multiprocessing Pool* kept running the *main.py* file, which houses all of the testing code for every other optimization technique. Obviously, the spawned processes were re-running the input command, *python main.py*. Moreover, I once got an error:

An attempt has been made to start a new process before the current process has finished its bootstrapping phase!

Looking this one up, I realized that, for basic testing, the best solution is to define a main method in the file of the multiprocessing code, otherwise the *multiprocessing* code would grow complicated with *start()* and *join()* calls, among other synchronization constructs. So, I put a main method in my *multiprocessing* code file and it worked.

```
def mandelbrot(c):
  z = 0
  for n in range(maxIterations):
    if abs(z) > 2:
      return n
    z = z * z + c
  return maxIterations

if __name__ == '__main__':
  time1 = time.time()
  realNums = np.linspace(xMin, xMax, width)
  imagNums = np.linspace(yMin, yMax, height)

  p = Pool()
  complexNums = [complex(real, imag) for imag in imagNums for real in realNums]
  escapeCounts = p.map(mandelbrot, complexNums)
  escapeCounts = np.reshape(escapeCounts, (width, height))
```

The final multiprocessing implementation.

Timings for this implementation were at 1.34 seconds, over twice as fast as pure Python.

2.12: MPI Optimization

Due to time constraints, *MPI* could not be implemented as an optimization technique. However, what follows are considerations on how optimizing the Mandelbrot calculations could effectually be done.

The Mandelbrot inclusion calculation, including the divergence check (*abs(z) > 2*) would remain the same as all the other methods, and could be optimized with the Pythagorean theorem trick learned earlier. The low-level Python trick of not using ***2* to square a number, and instead literally multiplying it by itself, could be used to save time as well.

MPI, through the *mpi4py* library, would produce the largest boost in speed by initializing an empty, two-dimensional *NumPy* array for containing integers, which are the escape counts. An *MPI* function like *scan()* could be used to apply an operation against the array element-wise. This array would be broken up into chunks and the Mandelbrot code called on each one. The results of this would have to be collected back into the root rank, using the *gather()* function.

3.1: Initial Timings

Below are provided the initial timings for each algorithm. Timings with increasingly large input values are provided in the next section, 3.2.

Optimization Technique	Initial Timing (seconds)	% decrease from original	x speedup from original
Pure Python	3.6	-	
Numpy	1.23	65.83%	2.93x
Better Numpy	.85	76.39%	4.24x
Numba	.24	93.33%	15x
Better Numba	.14	96.11%	25.71x
Betterer (guvectorized) Numba	.01	99.72%	360x
Numexpr	.45	87.5%	8x
Cython	.06	98.33%	60x
Multiprocessing	1.34	62.78%	2.69x

Table 3.1a: The initial timings for every technique employed. Refer to Table 2.2a for the input values used.

Visualization Cost

Visualizing the Mandelbrot sets for the various algorithms has an additional computational cost, making it a bit slower. More code means more time, obviously; so there is not much insight to be had with this observation, but it is included nonetheless for completeness.

Optimization Technique	Raw Timing (seconds)	Visualization Timing (seconds)
Numpy	1.23	1.56
Better Numpy	.85	.98
Numba	.24	.58
Better Numba	.14	.35
Betterer (guvectorized) Numba	.01	.08
Numexpr	.45	.58
Cython	.06	.10

Table 3.1b: The timings without visualizations, and with.

With the initial input values, the cost of visualization is not significant, but nonetheless, it is evident. For larger images this cost could become worthy of consideration. For this, GPU computing may be a solution for speeding up the visualization code.

3.2: Scalability: Testing Larger Numbers

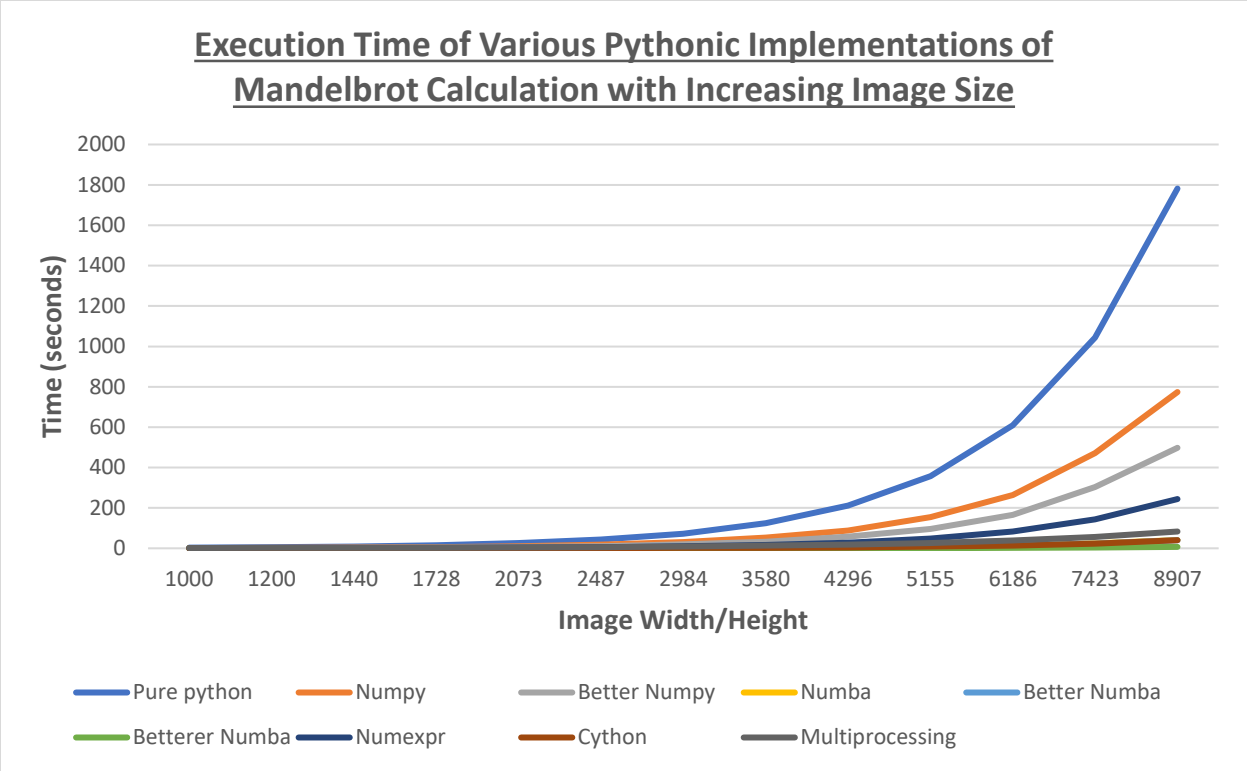
It is not sufficient to simply use one set of numbers to test speed. Different optimization techniques have different levels of scalability. Given larger numbers and more computation, some techniques that lag behind others with small numbers may excel with larger numbers, and some may exhibit curves that are interesting for analysis of their computational efficiency.

So, I wrote a clever bit of Python code to increment the image width, image height and maximum number of iterations input into the Mandelbrot code by 20% each time, to see the effect of progressively larger numbers on each optimization technique. After 13 increases of 20% (equating to a 260% increase in input values size), the code for pure Python took over half an hour each run, so it was decided to stop. Below is a verbatim inclusion of the timings table, with input data values. The *multiprocessing* implementation is not included in these tests because of issues validating the output of the Mandelbrot set; therefore, its timings cannot be regarded as representative.

Image width / height	**Iter.**	**Pure Python**	**Numpy**	**Better Numpy**	**Numba**	**Better Numba**	**Betterer (guvectorized) Numba**	**Numexpr**	**Cython**	**Multiprocessing**
1000	80	3.459	1.238	.769	.310	.180	.014	.450	.067	1.344
1200	96	5.737	2.122	1.335	.116	.112	.022	.741	.114	1.677
1440	115	9.261	3.532	2.164	.196	.188	.035	1.155	.193	2.178
1728	138	15.460	6.094	3.811	.326	.320	.060	1.952	.323	2.986
2073	165	25.723	10.511	6.524	.563	.544	.101	3.375	.558	4.395
2487	198	42.906	17.506	11.025	.959	.939	.171	5.733	.951	6.174
2984	237	72.628	30.218	19.308	1.612	1.585	.289	9.729	1.605	8.658
3580	284	124.331	52.355	32.593	2.765	2.717	.483	17.059	2.755	13.519
4296	340	211.270	88.413	57.287	4.756	4.649	.822	28.010	4.625	18.774
5155	408	356.757	155.205	96.411	7.987	7.833	1.39	47.693	8.013	26.895
6186	489	609.522	264.443	166.003	13.733	13.449	2.39	82.745	13.742	39.302
7423	586	1042.891	471.580	303.482	23.885	23.394	4.06	143.670	23.383	56.779
8907	703	1781.800	773.96	497.83	41.074	41.074	6.986	243.922	40.204	83.266

Table 3.2a: All timings (in seconds).

While these numbers may seem small, note that, in the last round of computations, the image is 8907x8907 pixels, where each pixel is a complex number (a Python object) that is put through 703 rounds of Mandelbrot testing until it either diverges or maxes out at 703 iterations to escape… this is a massive computational load! This is 79,334,649 pixels that must be checked!



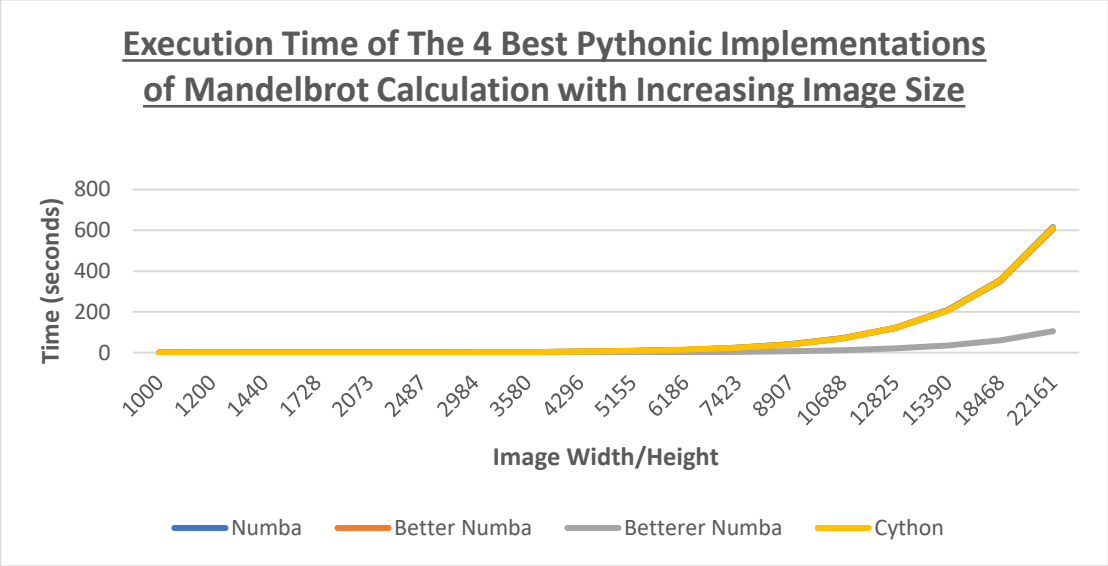
The Pure Python implementation goes exponential very quickly, with the two *Numpy* versions following behind. The *Numexpr* implementation is behind those, and then the *multiprocessing* implementation, and then the two earlier *Numba* implementations and *Cython* are pretty even. The absolute outlier is the best *Numba* implementation (the *guvectorized* one), which is over 5x as fast as the next best solution.

Optimization Technique	**Timing in seconds (smallest input value)**	**Optimization Technique**	**Timing in seconds (largest input value)**
Pure Python	3.46	Pure Python	1781.76
Multiprocessing	1.34	Numpy	773.96
Numpy	1.24	Better Numpy	497.83
Better Numpy	.77	Numexpr	243.92
Numexpr	.45	Multiprocessing	83.27
Numba	.31	Numba	41.07
Better Numba	.18	Cython	40.20
Cython	.07	Better Numba	39.97
Guvectorized Numba	.01	Guvectorized Numba	6.99

Table 3.2b: Timings for each optimization technique with the largest and smallest input values.

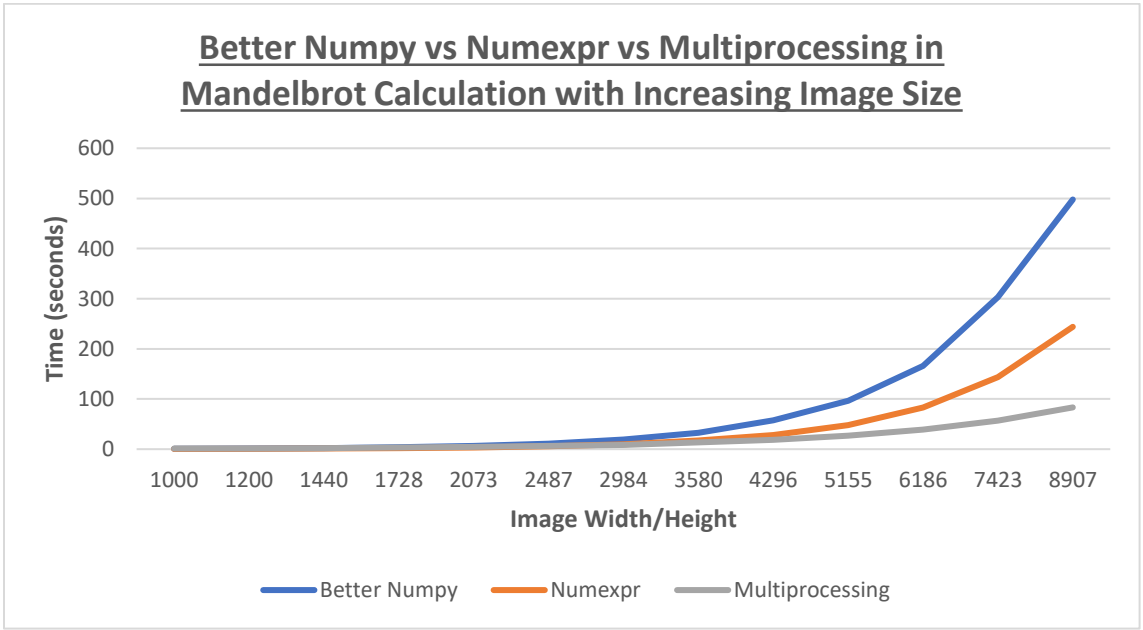
What is most interesting is that the two earlier Numba techniques become nearly identical in speed over time (despite the 2nd method being nearly twice as fast as the 1st with the initial data size). Furthermore, these two earlier Numba techniques begin to closely match the speed of the Cython code, which before had lagged considerably behind them, but seems to close the gap with more computations. Because the other codes became so slow, the three Numba and Cython implementation lines were not very distinguishable as they were so small compared to the rest.

So, I ran just those four for 18 increases of 20% input value size. Even at the 18th iteration of input value size, the largest time of 615 seconds was approximately a *third* of the pure Python implementation at the 13th iteration. *Numba* is very efficient.



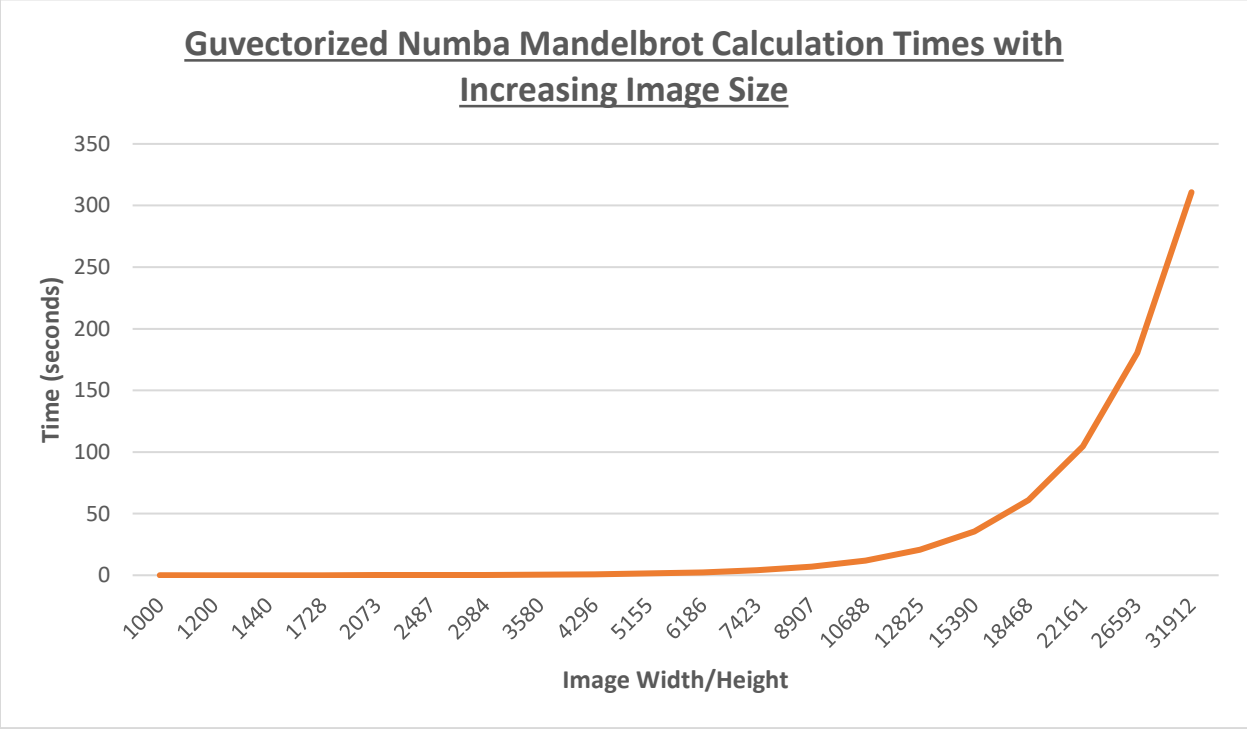
The *Numba*, better *Numba* and *Cython* implementations are all so close that they cannot be distinguished with the Betterer *Numba* implementation on the same graph. The code would have to be run for several hours to potentially find a divergence point. Of the three, the regular *Numba* implementation performed the worst; though, again, the differences between the three were very tight, perhaps no greater than 1%.

Moreover, some techniques beat *multiprocessing* early on, but it catches up to them with larger image sizes and eventually scales far better. Below is that observation:



3.3: Testing Blazing Fast Guvectorized Numba

Because the *guvectorized Numba* implementation is so exceptionally fast, it was decided to run it by itself, to see at what limit it really begins to slow down.



Even with an image width/height of nearly 32,000 pixels, *guvectorized Numba* only takes around 6 minutes. Recall, the pure Python implementation took 6 minutes to compute a Mandelbrot image of only 5000x5000 pixels.

4: Discussion and Conclusions

Mandelbrot Checking: Embarrassingly Parallel

Looking at the Mandelbrot code in pure Python: the checks for inclusion in the Mandelbrot set on any complex number are called through a double for loop. Put simply: for every complex number made up of a real part, *r*, and an imaginary part, *i*, this complex number, *c*, is looped through the Mandelbrot checks until it either diverges (the escape point) or the specified number of iterations has been reached. These escape counts are recorded and then appended to an array. Notice that the Mandelbrot check for a complex number is done *independent of any other complex numbers*. This is the definition of an embarrassingly parallel problem.

```
def mandelbrot_purepython(c, maxIterations):
    z = 0
    for n in range(maxIterations):
        if abs(z) > 2:
            return n
        z = z * z + c
    return maxIterations

def mandelbrot_set_purepython(xMin, xMax, yMin, yMax, width, height, maxIterations):
    realNums = np.linspace(xMin, xMax, width)
    imagNums = np.linspace(yMin, yMax, height)
    escapeCounts = []
    for r in realNums:
        for i in imagNums:
            escapeCounts.append(mandelbrot_purepython(complex(r, i), maxIterations))
    return (realNums, imagNums, escapeCounts)
```

Embarrassingly parallel double for loop and array appending

The simplest way to parallelize this code is to pre-emptively generate an array of complex numbers (ideally, not using Python's built-in complex type, but by passing in the real and imaginary parts themselves, as Python's complex type is an object and slows down computation). Pre-allocating this array of complex numbers allows us to make use of arbitrary array operations like *map()*, which can then be called against the complex number array with the Mandelbrot checks. This was done in the *multiprocessing* optimization, but the *guvectorized Numba* implementation, despite minimal effort, was untouchable in speed. The reason this is possible is because every check for some complex number, *c*, can be done independently of all others. So, making use of parallel processes is an obvious optimization technique.

In performing this project, it is my opinion that for loop-heavy, floating point array-bound operations, the starting point should be a *NumPy* optimization, followed by a *Numba* optimization, preferably with *guvectorize*. These two tactics take minimal effort or mapping from the original pure Python code, and *guvectorize* allows the benefit of parallelization without having to navigate the complexities of managing processes in *multiprocessing* or arranging data structures with *MPI*. Moreover, the programmer should be aware of low-level, language-specific "tricks", such as the faster way of computing a square, e.g., *z * z* as opposed to the common (but slower!) *z**2*; and, more specific mathematical quirks, like computing if *abs(z) > 2*, the faster form of which is derived from the Pythagorean theorem: if *z.real * z.real + z.imag * z.imag > 4*.

Finer Tests

I would like to perform more tests with larger amounts of data. Principally, I am interested in the *Numba*, *Better Numba* and *Cython* implementations, as they were nearly identical in speed, even up to an image size of nearly 10,000x10,000 pixels. It seemed that they were beginning to diverge from one another, however, with Cython seeming to pull ahead; but, more time would be needed to execute this, as each run took over six minutes.

Hardware Dependence

This code was run exclusively on one Windows 10 computer, with the following specifications:

- Intel i9-9900k (8 cores, 3.6 GHz base clock, 5.0 GHz turbo)
- Nvidia GeForce RTX 3080 (10GB GDDR6)
- 32 GB DDR4-3200MHz RAM

Code speeds will vary across computer platforms. Although the 32 GB memory limit was never reached during execution (the most witnessed was 36%), it is probable that the Windows operating system made use of virtual memory, paging or other techniques, through which the solid state storage on the computer is used to house memory operations temporarily. The operating system itself is installed on an M.2 SSD, which has sequential read/write speeds of up to 7,500 MBps and 7,000 MBps, respectively.

Potential Low-Level Mandelbrot Optimizations

Since the Mandelbrot fractal is *symmetrical*, it is technically possible to cut the number of computations in half. When a pixel is computed for rendering on the top, it can be mirrored directly to the bottom, or vice versa, effectively halving computation time. This would resolve in placing an additional if statement check in the Mandelbrot calculation code, and extra code in the visualization function to map pixels from top to bottom.

Also, since the central facet of the Mandelbrot check resolves to an if statement in code, it should be investigated if there are computationally faster ways of performing a Boolean check. Outside of Python, there is likely a body of work regarding this; but in Python, if statements and the *NumPy.where()* function were what were used in this work.

Visualization

Optimizing the visualization code itself may be as complicated as involving GPU computing, or as simple as using another plotting library in Python. For instance, the *pyplot* construct of *Matplotlib* is a very simple and efficient plotting module that, in my tests, seemed to be rather quick in visualizing even large sets of Mandelbrot escape counts.

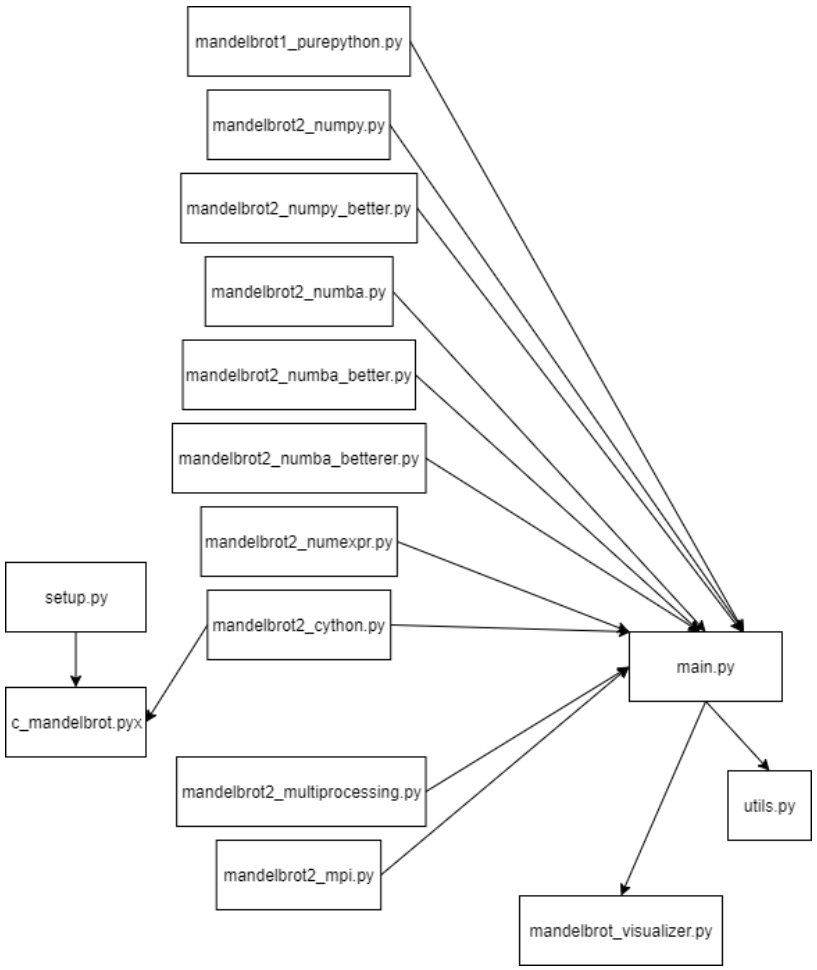
Other Applicable Optimization Libraries

My optimization work extended only to the mathematical Mandelbrot code itself. The code for visualizing the Mandelbrot sets was not touched. This is an area of potential future optimization, probably through the use of GPU computing or some other plotting libraries, as Matplotlib may be lacking in particular features. Plotting places some additional computational load and is therefore slower. This fact leads to the conclusion that plotting of the Mandelbrot set can potentially benefit from GPU computing. Some potential things to investigate include:

- *lru_cache* from the *functools* library
- *PyCUDA*
- *Tensorflow*
- *PyOpenCL*
- *Pythran* (ahead-of-time) compiler
- *PyPy JIT* interpreter

5: References and Appendix

- Description of complex numbers and the image of the complex plane: https://en.wikipedia.org/wiki/Complex_number
- Numba guvectorize: https://coderzcolumn.com/tutorials/python/numba-guvectorize-decorator
- Mandelbrot set in Python: https://realpython.com/mandelbrot-set-python/
- How to draw the Mandelbrot set in Python: https://www.javatpoint.com/how-to-draw-the-mandelbrot-set-in-python
- Making Python as fast as Julia: https://mspyzblog.wordpress.com/2017/09/28/how-to-make-python-run-as-fast-as-julia/
- PyOpenCL Mandelbrot demonstration: https://github.com/inducer/pyopencl/blob/main/examples/demo_mandelbrot.py
- Mandelbrot set in Python with divergence point checking Pythagorean optimization: https://medium.com/swlh/visualizing-the-mandelbrot-set-using-python-50-lines-f6aa5a05cf0f
- Mandelbrot set Python multiprocessing: https://magpi.raspberrypi.com/articles/multiprocessing-with-python
- Mandelbrot set Python dynamic multiprocessing: https://github.com/DipanshuSehjal/Mandelbrot-set/blob/master/mandelbrot.py



Simplistic software architecture diagram.

```python
def timed(func):
  @functools.wraps(func)
  def wrapper(*args, **kwargs):
    start = time.perf_counter()
    ret = func(*args, **kwargs)
    return time.perf_counter() - start
  return wrapper
```

My timer wrapper; pass it a function and it returns how long it takes to run.

```python
for i in range(_LOAD_TIMES):
  print(_WIDTH, 'x', _HEIGHT, ':', _MAX_ITERATIONS, 'iterations')
  for stringName, funcName in utils.funcs.items():
    funcTimed = timed(possibles.get(funcName))
    avgAlgTime = sum(funcTimed(_XMIN, _XMAX, _YMIN, _YMAX, _WIDTH, _HEIGHT, _MAX_ITERATIONS) for _ in range(_EXECUTION_RUNS)) / _EXECUTION_RUNS
    if _NOISY:
      print(stringName + '\t\t' + str(avgAlgTime) + 's')
    else:
      print(str(avgAlgTime))
  _WIDTH          += int(_WIDTH          * _LOAD_AMT)
  _HEIGHT         += int(_HEIGHT         * _LOAD_AMT)
  _MAX_ITERATIONS += int(_MAX_ITERATIONS * _LOAD_AMT)
```

Code snippet that allows running all the optimization implementations with increasingly larger values for testing scalability.

6: Acknowledgements

These are sources I encountered, considered/read fully but did not use. Many of them influenced my project direction or are included as future works.

- Discrete Laplacian Image Edge Detection: https://flothesof.github.io/optimizing-python-code-numpy-cython-pythran-numba.html
- Mandelbrot Set with PyCuda: https://wiki.tiker.net/PyCuda/Examples/Mandelbrot/
- Non-Uniform Fast Fourier Transform with NumPy and Numba: https://jakevdp.github.io/blog/2015/02/24/optimizing-python-with-numpy-and-numba/
- Inverse Fourier Transform with Numb and Cython: https://flothesof.github.io/cython-complex-numbers-parallel-optimization.html
- Numba and CUDA Mandelbrot visualization: https://www.kaggle.com/code/landlord/numba-cuda-mandelbrot/notebook
- Python HPC Tutorials: https://github.com/mmckerns/tuthpc
- Understanding Julia and Mandelbrot sets: https://www.karlsims.com/julia.html
- Mandelbrot Optimization Techniques: https://codereview.stackexchange.com/questions/216235/increase-performance-creating-mandelbrot-set-in-python
- Mandelbrot in CUDA/MPI C: https://github.com/e-bug/parallel-mandelbrot-set
- Mandelbrot multiprocessing: https://timothyawiseman.wordpress.com/2012/12/21/a-really-simple-multiprocessing-python-example/
- Simple Mandelbrot plot in Python: https://github.com/danyaal/mandelbrot
- Mandelbrot mpi4py: https://github.com/mpi4py/mpi4py/tree/master/demo/mandelbrot
- Mandelbrot mpi4py simpler: https://github.com/alshedivat/ACM-Python-Tutorials-KAUST-2014/blob/master/mpi4py-tutorial/examples/mandelbrot-mpi-block.py