Phillip Linson

Dr. Nick Flann

CS 4700

27 April 2020

# Does Julia Perform More Like C++ or Python?

**Abstract:**

The purpose of this project was to test the performance of Julia against that of C++ and Python, then determine whether or not the claim that Julia performs more like a statically typed language, like C++, was true. To do this, I wrote a variety of algorithms from several different disciplines in each language and timed their execution. The results were mixed; as expected Julia typically performed somewhere inbetween C++ and Python in terms of speed, but this varied widely by the algorithm used and the type of algorithm. The claim holds true in the sense that it performs closer to C++ than Python does, but false in that it performs more like Python than it does C++. The code and results can be found at my public repository at https://github.com/Shichimenchou/CS4700FinalProject.

**Motivation:**

Since this a programming languages course, I wanted to familiarize, or at least refamiliarize, myself with a breadth of languages as well as see for myself the various benefits and shortcomings of these languages. As Julia is a language I'd been hearing much about, I wanted to take the opportunity to learn it and compare it with languages I've learned in the past. My thought was if I got an interesting result during the project, I might have just stumbled upon quick a useful language.

My decision to test the languages against algorithms from multiple disciplines was as much for my personal benefit as it was to test the languages widely. This is my last semester at university and I don't know what kind of development work I'll end up doing, or even in what langauges I'll writing. By practicing broadly throughout this project I hoped to familiarze myself not only with the languages

themselves, but the algorithms I'm using. Furthermore, implementing the algorithms in each language would push me to use parts of them that I haven't used before.

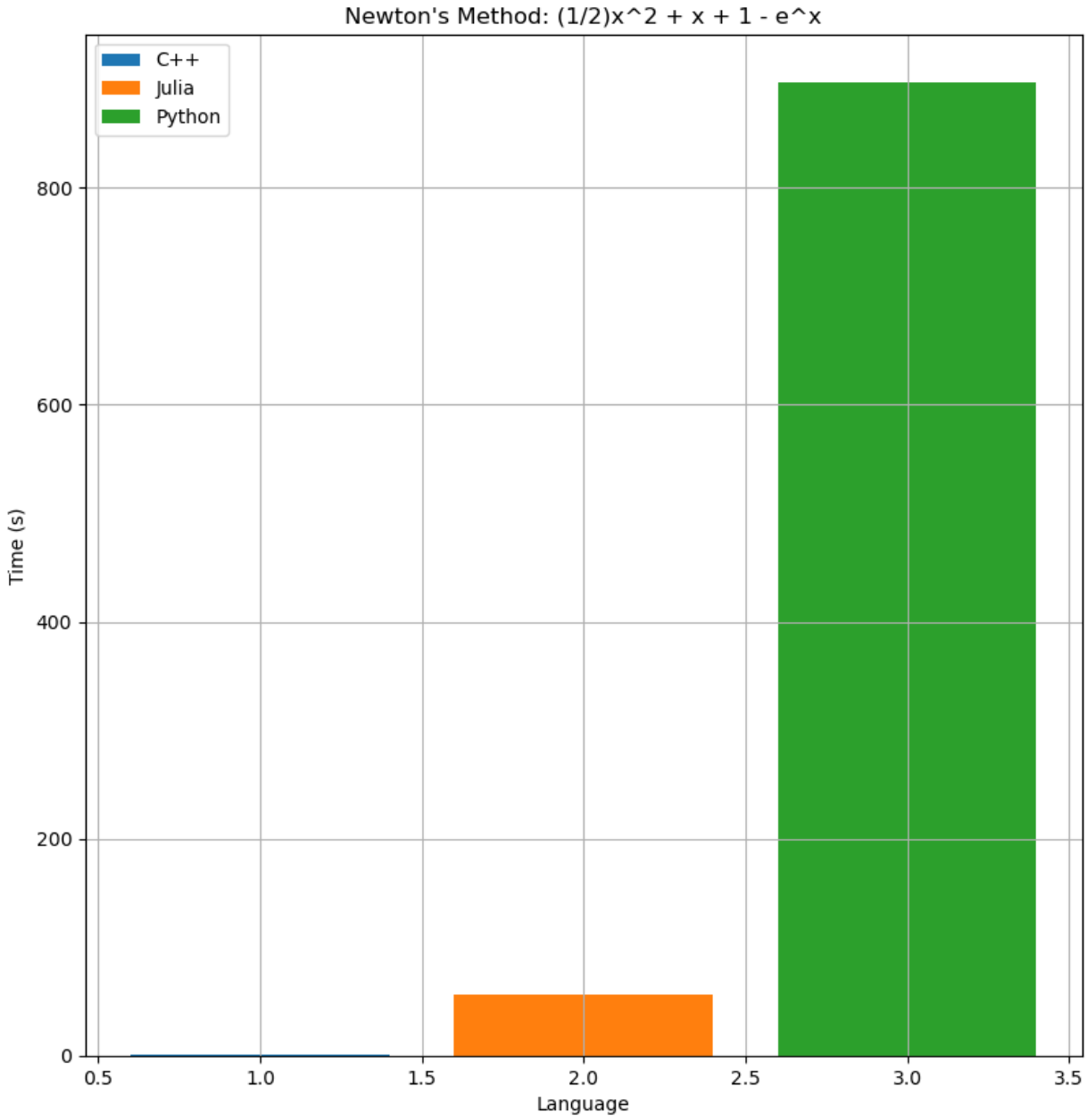
Finally, the central purpose of the project was the test a major claim that Julia makes; namely that it performs more like statically typed languages like C++ than it does other dynamically typed languages like Python. If this proved true, I'd have a handy replacement for Python in cases where the ease of dynamically typed languages is too much to resist without having to sacrifice much in the way of performance. Additionally, I hope this project and the resulting Git repository can be a resource for others to test this claim for themselves.

**Results:**

Scientific Computing

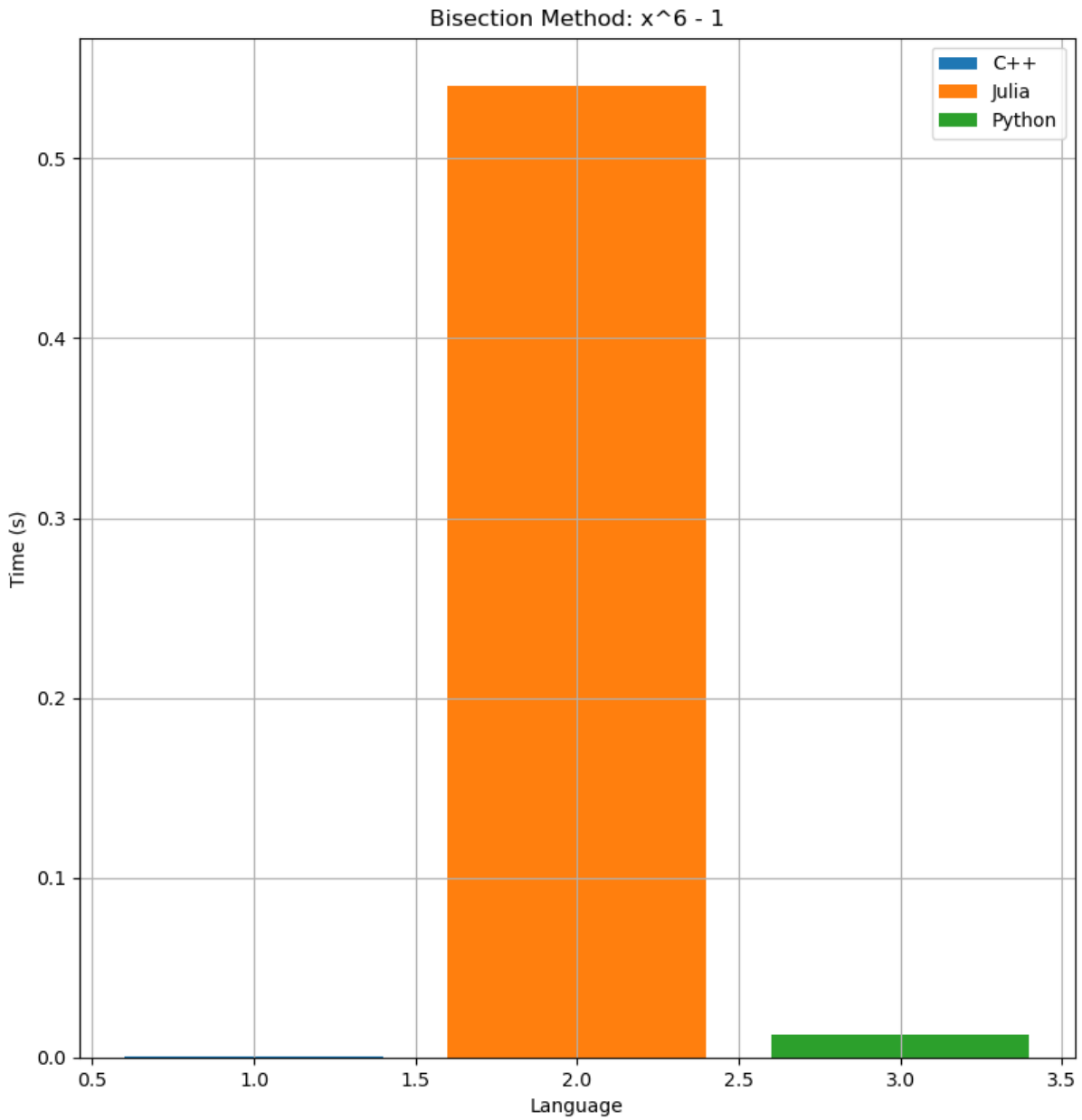
The results of this category are the most mixed. Some algorithms showed dramatic differences in time performance, while others resulting in each language giving the same result in very nearly the same time. Additionally, this category has to deal with the confounding factor of numerical accuracy. Python and Julia both support arbitrary precision, while C++ has no native support. In order to level the playing field I chose a level of tolerance within C++'s precision range, which had the effect of small time differences in some cases.

First I tested Newton's method for finding a root of a function. Fortunately, I was able to find a function that converges slowly over many iterations with this method, providing the most interesting results of this category. The function in question is $f(x) = ½ x^2 + x + 1 – e^x$. The roots to this function are clearly 0 and 1, but because of the similarity between the the function and its derivative, each iteration of Newton's method changes only slightly. With an intial guess of x=100, I tasked each language to find the root 1 to a precision of 0.01 with Newton's algorithm. The results can be seen in the graph on the next page.




Needless to say, Python underperformed on this test. I thought at first that this was due to my using Python's arbitrary precision package, but I also used Julia's equivalent package, which actually gave root estimates to a greater precision than Python's despite its much faster time. As Python is currently a large player in the scientific computing community this result may well indicate Julia is on its way to replacing Python for that purpose. Also note that while C++ finished much quicker, the admittedly low precision of 0.001 for this test is the lowest C++ can comfortably compute.

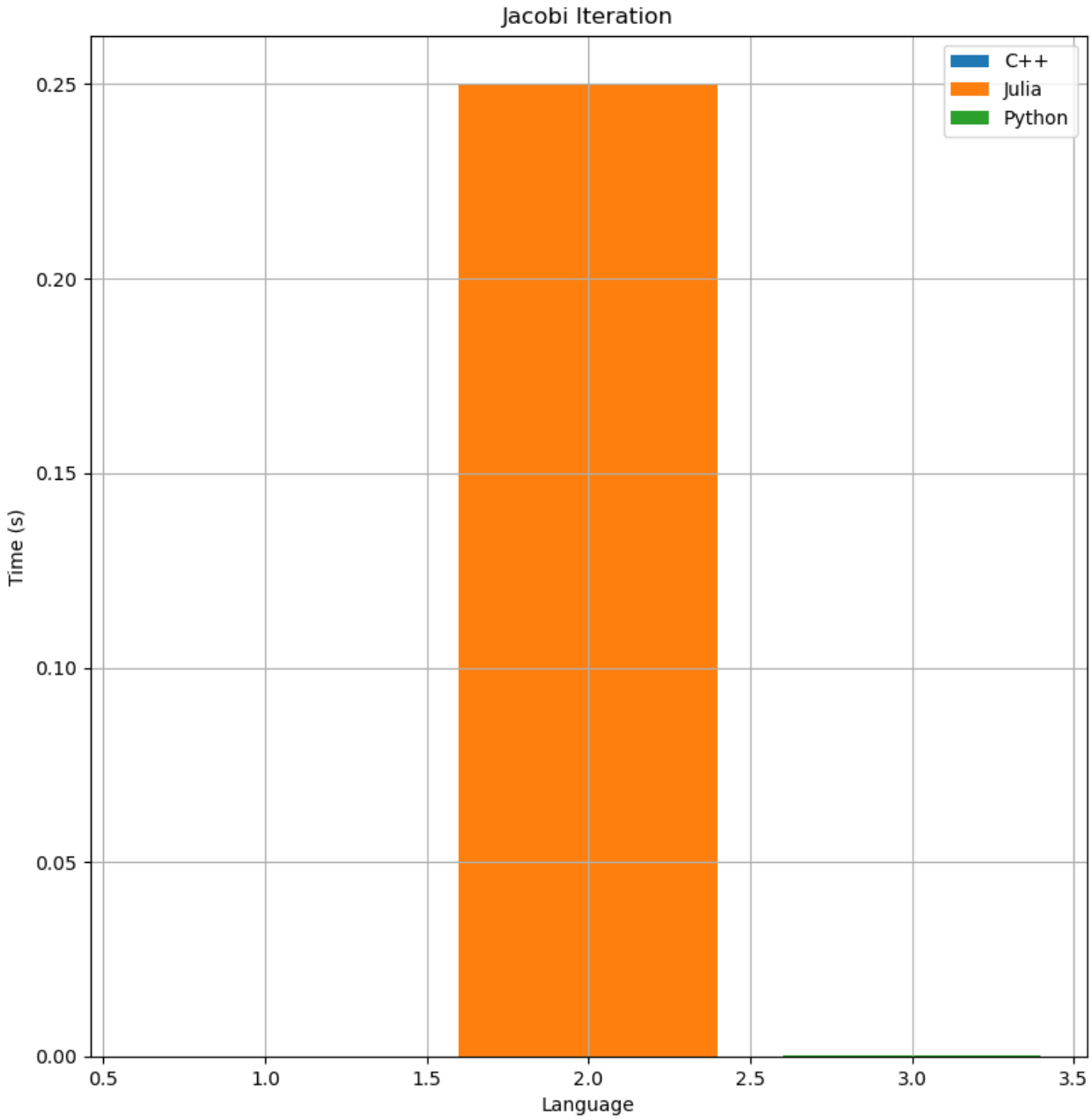
The next algorithm I tested was the Bisection method. Since this algorithm reduces its search range by half each iteration by design, it has a relatively high floor for simple algorithms in terms of number of iterations, but a very low ceiling for complicated algorithms by the same measure. After testing many functions and finding the number of iterations varied little, I chose the simple function $f(x) = x^6 - 1$. In the end, each language performed very well, as you can see in the graph below.



Julia technically performed the worst, but since all languages returned the correct result in well under a second, I'm inclined to ignore those differences; time differences this small could easily be accounted for by startup costs the the timing mechanism themselves. In general, the Bisection method proved to be a poor algorithm to test performance in terms of time. Because of that low ceiling in iteration numbers I mentioned before it was nearly impossible to push the languages to their limits with this algorithm.

Finally, the last numerical method I tested was Jacobi iteration. When the input matrices get large, this algorithm can quickly grow complex. Unfortunately, I ran into the same problem with C++ where my testing was hindered by its relatively low precision. The results can be seen in the graph on the next page.

While again Julia is technically the worst performer here, it still returned correct results in our precision range of 0.00001 in under a quarter of a second while the others returned in very near to 0 seconds. Again, I'd say that puts these languages in a three-way tie for this test. If I were to do this project again, or indeed if time permitted extra rounds of testing on this current project, I'd like to extensively test this algorithm on large matrices with only Python and Julia. Based on the results in the first section and the close performance on this small test, I believe Julia might outperform Python in both accuracy and speed for this algorithm. In the end, I leave it inconslusive.

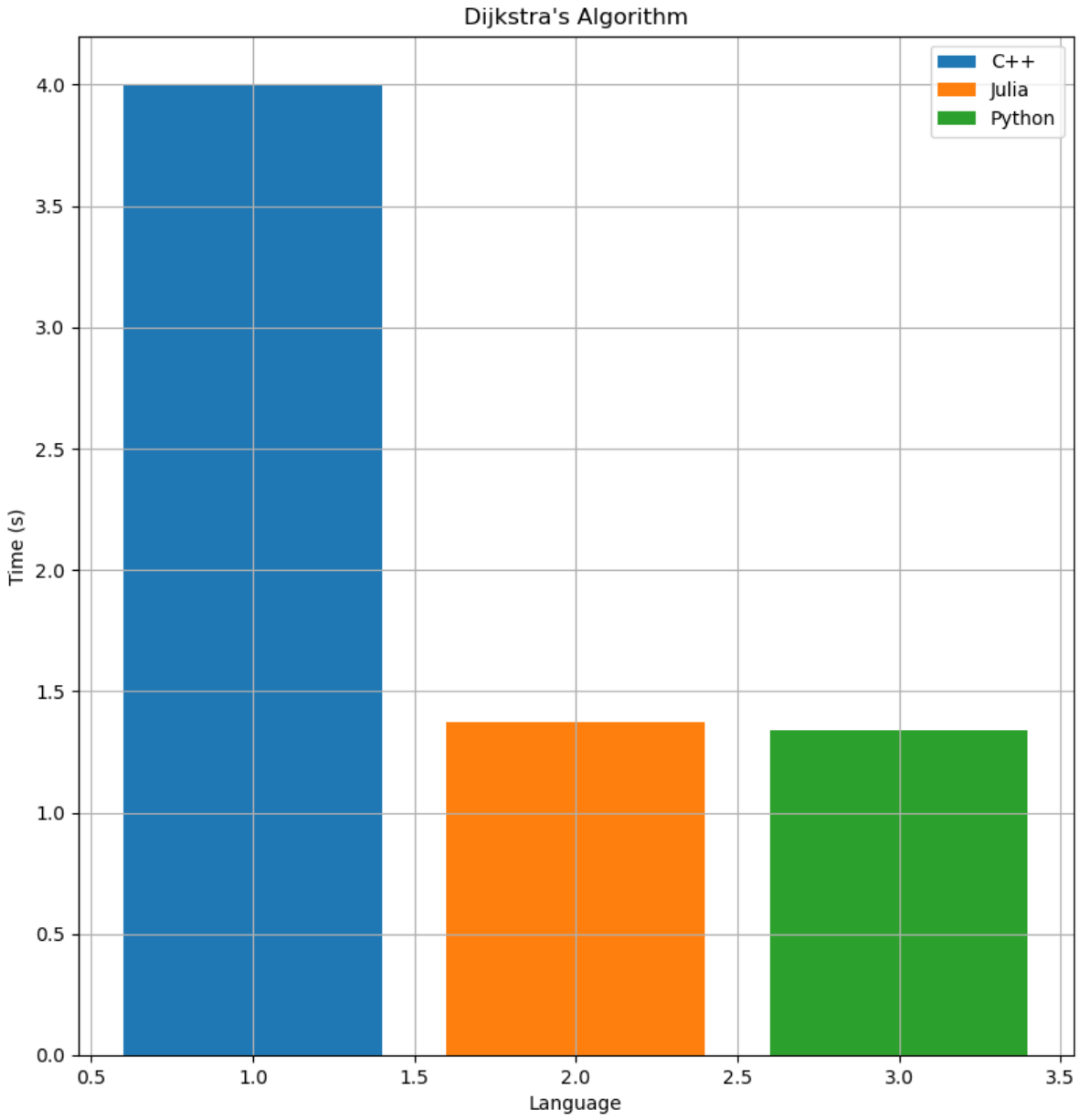



Graph Algorithms

The next category was that of graph algorithms, which also yielded suprising results. The motivation to include these algorithms is that graph theory is the mathematics behind database connectiveness, and a language which can handle these well will be exceptionally versatile.

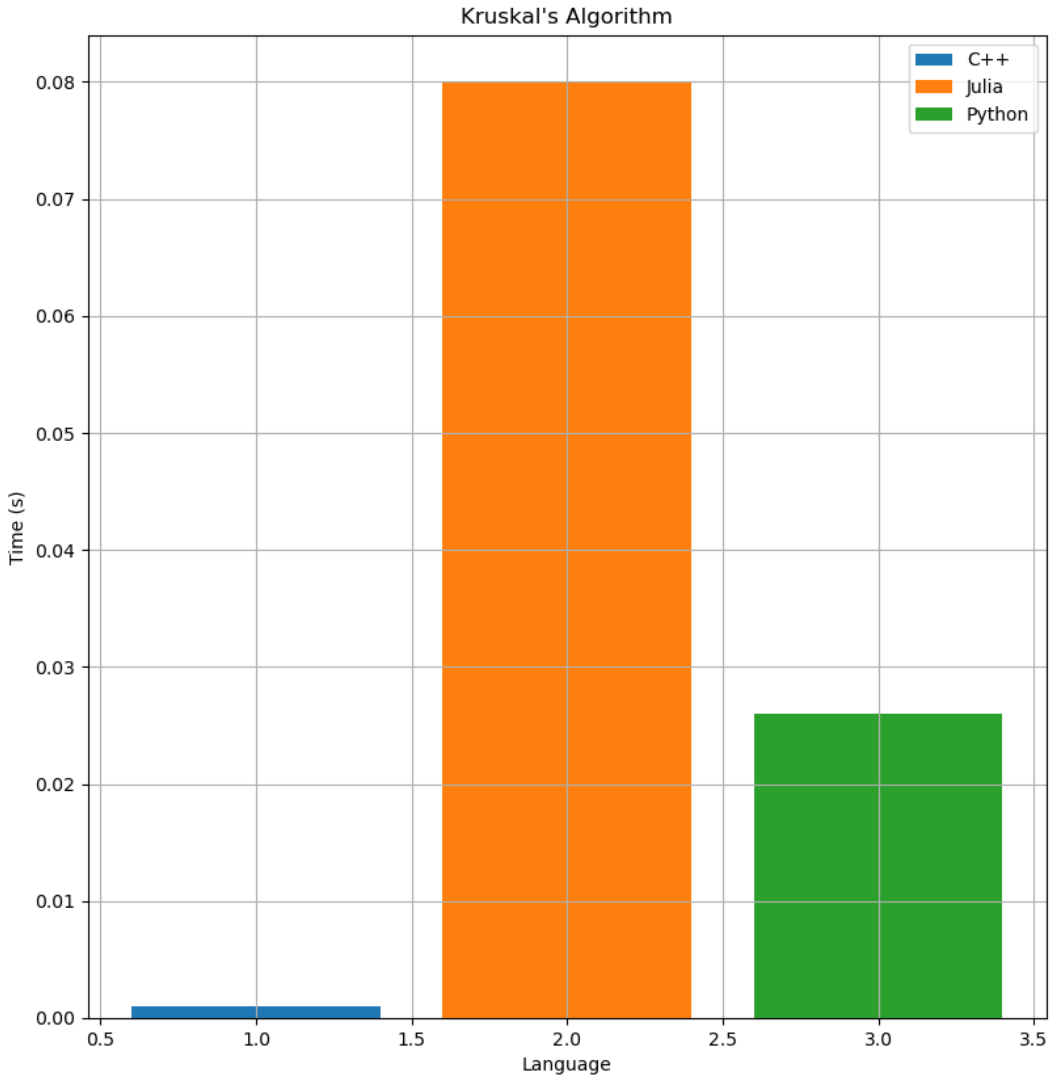
The first algorithm I used was Dijkstra's famous algorithm for finding a spanning tree on a graph. In particular, since I was only testing the performance of the various languages I made the graph

a complete graph on 10 vertices for simplicity. The weights for each edge were randomly generated (but put in by hand) and shared exactly between each language to cut down on confounding factors. In each case the language standard for a “list” object was used; I didn’t take advantage of C++’s pointer functions to speed up graph traversal for fairness.  The results can be seen in the graph below.



Here we see a first among these tests with C++ performing by far the worst. Since the test was relatively small the overall time is still quick, but the difference between C++ and the others is around 2.5 seconds—too much to ignore. For this test the difference in performance between Julia and Python, however, is negligible. Overall, this specific example could be seen as evidence that the claim "Julia performs as well as C++" is actually an *understatement*.

Finally, the next algorithm of the graph category was Kruskal's algorithm. Here the performance difference between the languages were small, which surprised me because of the variety of results with the similar Dijkstra's algorithm. The results from Kruskal's can be seen below.



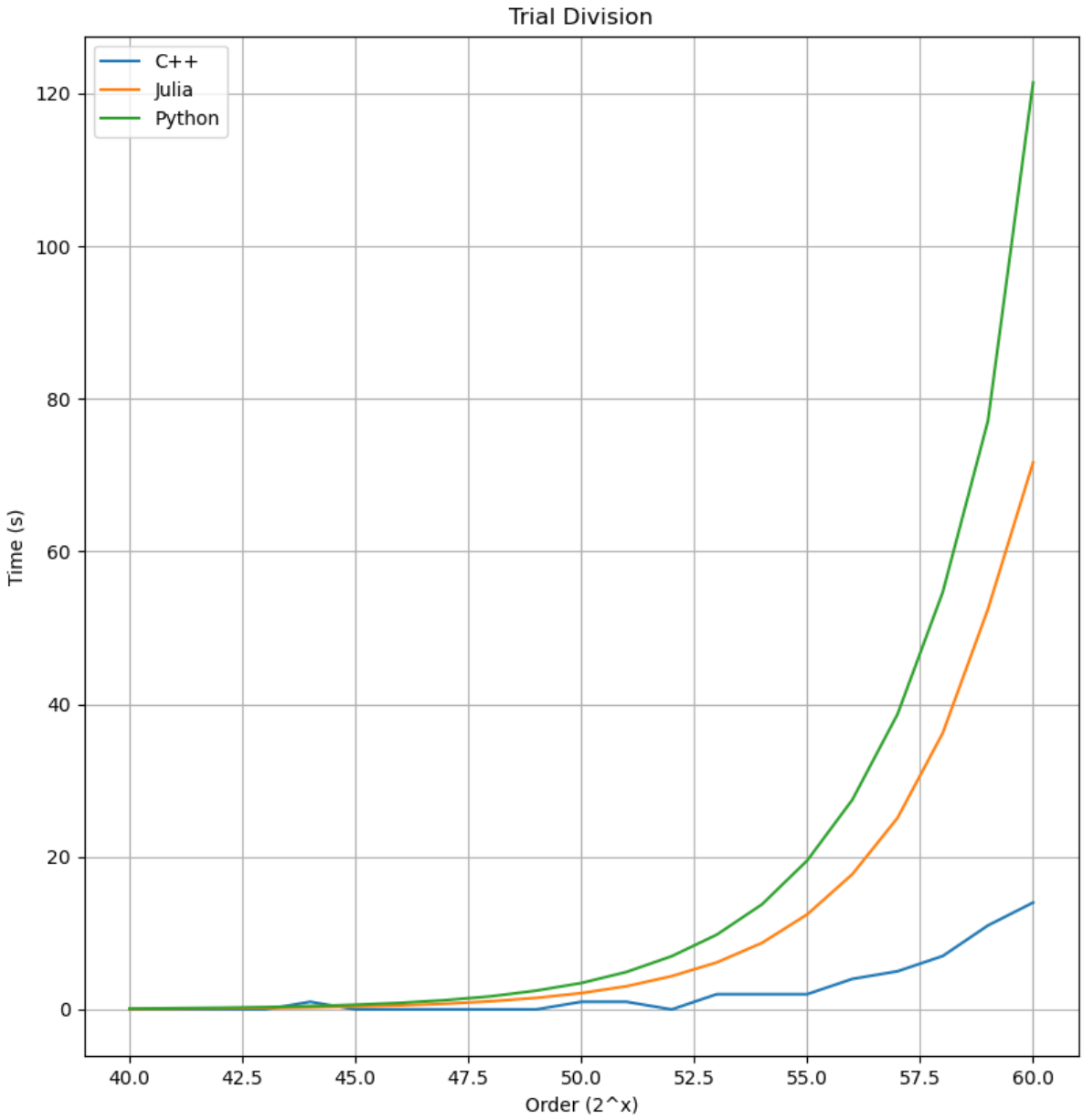
Each implementation returned in under a tenth of a second. If time permitted I'd like to test Kruskal's algorithm over larger graphs, but as it stands we have to work with the single test. As far as it goes Julia seemed to perform the worst, with C++ performing by far the best (I had to inflate C++'s bar so it would be visible). Still, it seems from other tests that Julia has a small startup cost for timing that the other languages do not. As such, I expect Julia would scale fairly well with Kruskal's if I were to perform more tests.

## Factoring Algorithms

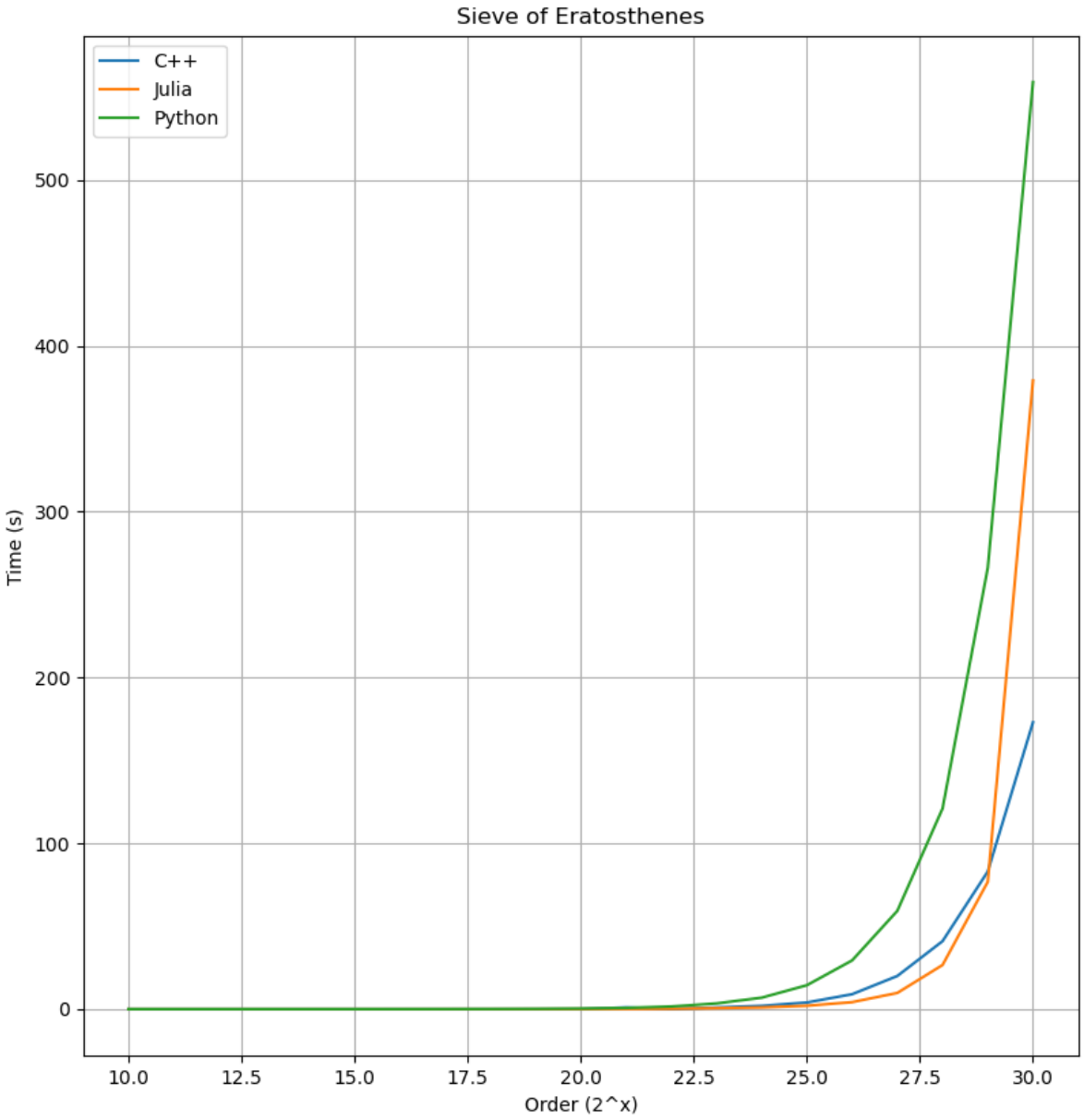
Of all the categories, this one most directly answered my questions of Julia's performance. I'm also quite pleased with the relatively consistency of the results I found in this category, even though there were some hiccups.

Perhaps the simplest algorithm one can think of in computer science is factoring by trial division. As a brute force algorithm, this proved very useful in demonstrating performances differences in these languages. For these test I ran the algorithm in each language multiple times, from $2^{40}$ to $2^{60}$, factoring each number by trial division up to the square root of the number. The results of each test can be seen the graph on the following page.

As you can see, each language performed very similarly in the lower orders, consistently returning all factors in at most a few seconds. As the orders grew, we see an explosion in time for each language, as well as clear differences in performance. In such extreme conditions C++ was by far the winner, factoring the largest number in around 15 seconds compared to Julia's 75 and Python's 120. This does seem to indicate that Julia performs more like C++ than Python does, but is more akin to Python in overall performance. Even still, Julia performed about 30% better than Python at the extremes, which is no small matter.




Given that I used such a brute force algorithm first, I wanted to explore an elegant, quick, but still simple algorith: the Sieve of Eratosthenes. I included this in the factoring section as it best fits here even though I used the algorithm to find all primes under a given value without using those primes to further factor anything. In particular, I tasked each language to find all prime numbers from $2^{10}$ to $2^{30}$ using the Sieve. The results can be seen in the graph on the next page.




As you can see the results for low order are again very clustered, with each language returning with only negligible time differences. However during the first of the larger orders Julia actually outperforms C++ by a respectable margin! It wasn't until the final test that Julia hit a wall. At this point Julia was consuming so much memory my RAM was filled and Julia began using virtual memory as well, dramatically reducing its performance. To verify that this was a problem with Julia I checked that

for leaking memory and reran the results of the lower orders with very similar times, again bottlenecking at $2^{30}$. Since all these tests were ran under the same conditions with very little else running on my machine, I am forced to conclude that this is most likely caused by a less-than-perfect memory manage system of Julia. However, this may be something that more veteran Julia programmers can ameliorate or avoid altogether. Still, Julia outperformed expectations by not only returning faster than Python in every case but also by outperforming C++ at lower orders.

**Summary:**

In total, I spent the first part of the project learning Julia and practicing with C++. When I was comfortable with these, I set out to broadly test their time-wise performances against eachother and, in particular, test Julia's claim that it performs more like C++ than Python.

To this end, I wrote in each of these languages several algorithms varying from factoring and scientific computing to grpah algorithms, then timed their results.

The conclusions are striking. While not faster than Python in every case, Julia does tend to perform faster than Python in most cases. In particular, Julia performs better than Python in scientific computing, where Python is king! Additionally, in one algorithm Julia even outperformed C++ for a significant stretch. Further testing would be useful to determine exactly which cases Julia is best used for over C++, but it stands true that for a simple, approachable language Julia performs exeedingly well.