CS840 Project 2:

Matrix Multiplication Runtime

For Different Languages

Samuel Gluss

Jozo Dujmovic

3-14-2016

Abstract – Selection of an appropriate language is a crucial task for any programming project. Some languages may have much better support, with more users. Other languages may have powerful libraries that enable developers to make fast progress. One tradeoff that is paid for higher level languages is reduced control over the low level operations of program execution. An example of this is C++ vs. Python. In C++ a matrix must be allocated in memory row by row, and values initialized individually. The programmer is entirely responsible for the fine points of this process. By contrast, Python allows matrices to be initialized in a single line using list comprehension. It is assumed that the interpreter is doing things in the best way possible, but the developer is given little say in the specifics. This paper will study the effects of different memory access patterns in different languages, and exhibit the advantages and disadvantages of choosing one language over another with access patterns in mind.

# I. Introduction:

The purpose of this paper is to demonstrate the level of control that different languages offer a programmer. What will be examined specifically is the effect of reconfiguring memory access patterns in the source code, and of secondary importance will be the total run times of the experiments. The reason for this is that the languages will be tested with just one compiler or interpreter each, so it is entirely possible that by varying the selection of interpreter/compiler/settings the relative performance of languages could be varied quite wildly.

For the purposes of this test, C++, Java, JavaScript, Ruby and Python will be examined. For each of these languages, the timing library has been evaluated for maximum error to determine nominal test duration. A basic matrix multiplication benchmark was developed for each language. This benchmark first initializes two input matrices, then calculates matrix products to populate a solution matrix. This process will be repeated for matrix sizes from 20x20 to 500x500 to see not only the performance improvement possible, but also the change in improvement with matrix size.

The key point of the test is to test all 48 possible configurations of the matrix product step. The three for loops can be swapped, as well as the positions of the indices in the inner op. A python script enables the source code to be modified, the resulting program to be run, and the results aggregated. These results are then processed by MatLab code to determine which configurations were most and least performant, what the performance ratios for different languages were, and other useful data.

Ultimately, the goal of this paper is to answer the question of how much potential for improvement of memory access performance is wielded by the programmer. With this info in mind, the developer can then make an educated decision on what language to choose for certain applications, and how much energy and time to expend testing different access patterns.

Table of Contents

[I. Introduction: 1](#_Toc446631214)

[II. Matrix Multiplication Time Analysis 4](#_Toc446631215)

[III. C++ Data 6](#_Toc446631216)

[III. Java Data 8](#_Toc446631217)

[IV. Python Data 10](#_Toc446631218)

[V. Ruby Data 12](#_Toc446631219)

[VI. JS Data 14](#_Toc446631220)

[VII. Q Factor Analysis 16](#_Toc446631221)

# II. Matrix Multiplication Time Analysis

All tests were executed on a Lenovo Thinkpad W541 equipped with an Intel i7-4710MQ 2.50 GHz quad-core processor and 16GB of Crucial DDR3 PC3-12800 RAM. This machine is running Microsoft Windows 7 Pro Service pack 1 x64. All tests were executed after a clean reboot of the OS, and with all network connections disabled.

The compiler and interpreter specifics are as follows:

C++: Microsoft (R) Build Engine version 14.0.24723.2 Default settings

Java: javac 1.8.0\_66 Default settings

JavaScript: phantomjs-2.1.1 Default settings

Ruby: ruby 2.2.4p230 (2015-12-16 revision 53155) [x64-mingw32] Default settings

Python: 2.7.11 (v2.7.11:6d1b6a68f775, Dec 5 2015, 20:32:19) [MSC v.1500 32 bit (Intel)] Default settings

For each programming language, a test program determines the resolution of the built in timing library. For most languages, the windows Query Performance Counter (QPC) API is leveraged in one way or another, providing approximately 500 nsec of resolution. In Ruby, there is no existing way to interface with QPC, so the built-in time library was utilized, providing a resolution of 1 ms. This means that any test that is measured must be able to reduce this timing error to under 0.1%, resulting in a minimum test time of 2 seconds. This ends up being a relatively trivial problem in Ruby, as most tests take much longer. The test program calls the function which gets current time as rapidly as possible, then measures the duration of the steps and determines the maximum error that can be expected from this function. This information is used to determine the number of iterations to measure in the matrix multiplication benchmark.

Once the timer resolution is determined, a quick test pass on the matrix multiplication benchmark is executed. This gives a rough estimate of the time taken for each matrix size in each language. A matrix size is selected which has an execution time with a safe margin above time required to minimize timer error to sub-0.1%, this matrix size is N. The number of matrix multiplications to execute for any size is given by the floor of the ratio of third power of N over the third power of the matrix size to measure. This results in execution times with a flat minimum time, as the calculation accounts for the O(x^3) complexity of matrix multiplication. At some point the ratio falls to under 1, at which point the floor becomes zero. This is the point at which a single matrix multiplication op is safely large enough for the timer error to be sub-0.1%. Even so, it is desirable to take multiple measurements to ensure a quality result, so a flat constant is added. This constant may be as small as 3-5 for interpreted languages, which have very long runtimes. Compiled languages can run the ops much faster, so repetition counts may be as large as 100.

Once each matrix multiplication cycle is done for each matrix size, the duration is divided by the number of execution times to get the mean time per single execution. This time is added to an array of recorded times for each matrix size. At the end of execution, these times are dumped to output as a comma separated string, with a single newline at the end. The driver file takes advantage of this to produce a matrix of results.

The driver file is written in Python, and takes advantage of metaprogramming to automate testing of each matrix multiplication program. This works well because while the driver executes each test file, it is effectively idle, and as a result does not adversely affect the results of each test any more than any other background service.

The driver files for each program are distinct because each program will require its own syntax for the for loops and inner matrix product addition step. Furthermore, the driver file must know where in each program to make the appropriate substitutions, so an anchor, in this case a line number, must be provided for each program. Finally, each driver contains distinct paths for an output file, a source file or script, a compiler or interpreter, and an executable, if one is to be produced.

The remainder of the driver file is much the same for each program. The script first reads in the source code of the program to modify. It then recursively produces all possible permutations of 0,1,2. Once each permutation is generated, it is used to create a combination of for loops, and for each permutation, 8 different matrix product op permutations are to be tested. For each of these, the for loops and op are written to the source file. At this point, the program will be compiled if it is C++ or Java. Next, the program is executed, with or without an interpreter. The output from this execution is appended to the designated outfile. By the time the driver completes execution, the output file will have 25 columns, one for each matrix size, and 48 rows, one for each possible configuration of for loops and inner op.

The driver file works by first reading in the specified source file, modifying the source in the script, then writing these modification back to the original source file. For python, and readability in other languages, the correct indentation of each line of code can be very important. For this reason, the script must also know the indentation of the first for loop. Then, the modified code which is inserted is offset by an appropriate amount.

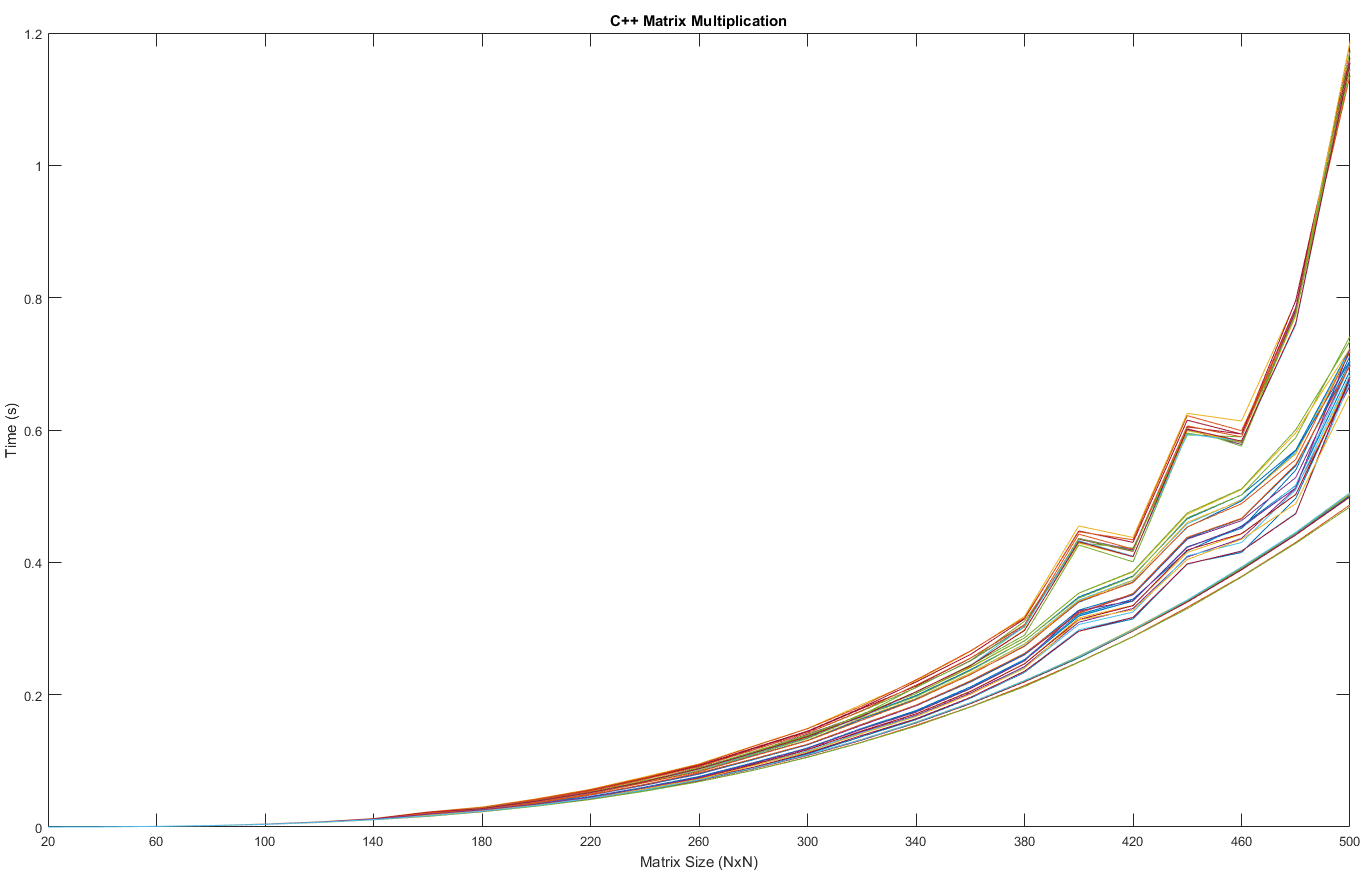
These data matrix output files can be parsed by MatLab into data objects, where the fastest and slowest times recorded for each matrix size are saved, as well as the index of the access pattern that produced the best and worst results for the 500x500 matrix size. The fastest and slowest data points are used to compute the Q factor for each language, at each matrix size. It is assumed that the access pattern will have more effect the larger the matrices, so the best and worst access patterns are chosen at the largest matrix size (500x500).

The Q factor curve changes with matrix size, and is computed as the ratio of

100 \* (Tmax – Tmin) / (Tmax + Tmin).

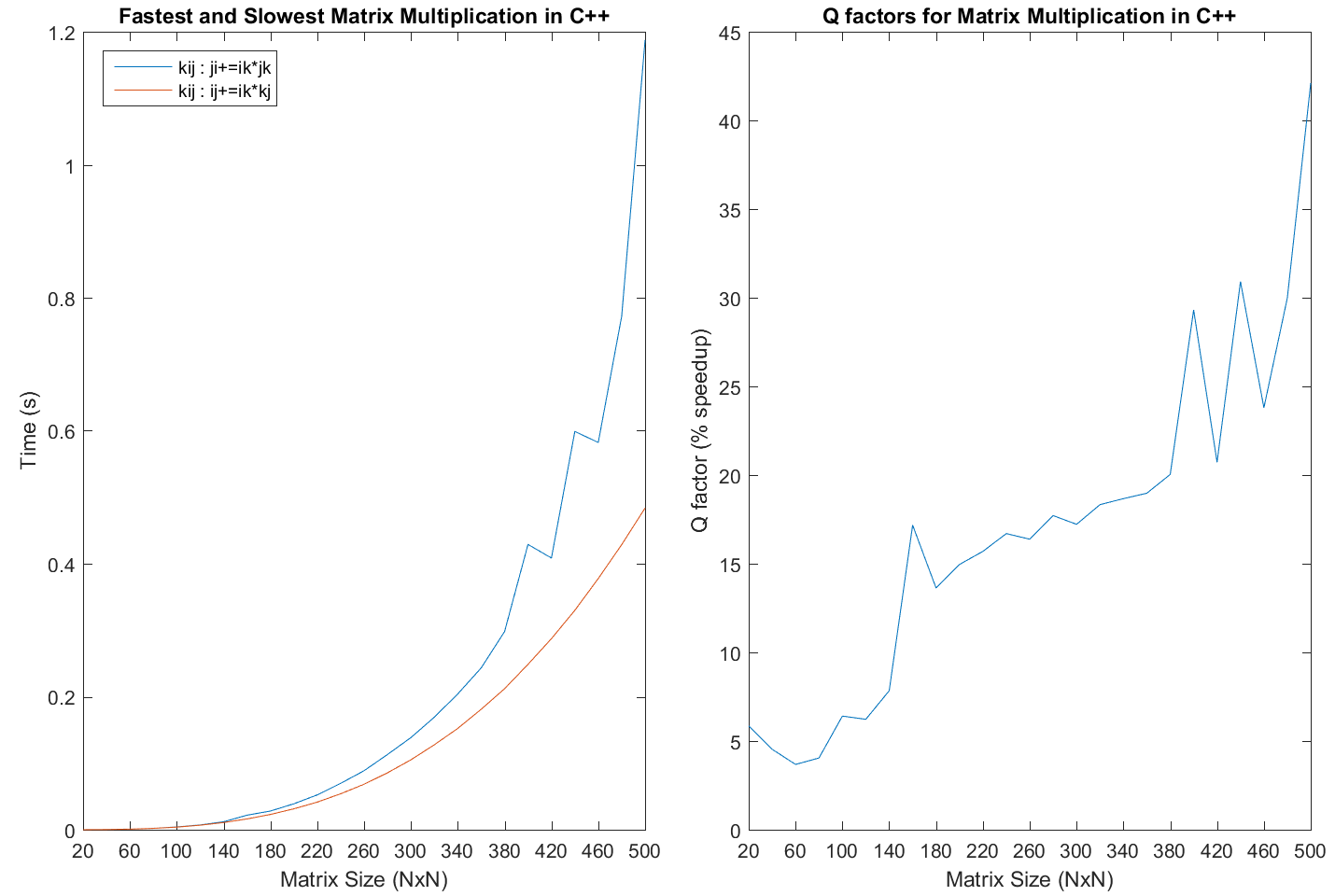
The 80-40-20 rule suggests that compiled languages such as C++ should have a Qfactor of around 80%, virtual machine languages such as Java should be around 40%, and interpreted languages such as python and ruby to be in the 20% range. One of the objects of investigation of this paper will be whether these languages behave as expected, and if not, what the reasons for this could be.

# III. C++ Data



The lower level languages such as C++ and Java required a much higher number of average runs to eliminate artifacts from the experiment. This is because the runtimes are so short, and so interference from other processes can cause aberrations in the data. With a minimum execution count of 100 for each matrix multiplication, random events are attenuated, and the resulting data is a bit unusual. The most prominent feature is the jagged results visible from matrix size 380x380 to 480x480. These features are more pronounced for the slowest-performing access patterns, and are consistent with behavior known to be caused by CPU clock rate adjustments. This effect is surprisingly consistent across many runs on multiple access patterns though, so further testing on additional machines and fixing the clock rate would help determine the true cause of this anomaly. Furthermore, additional testing to larger matrix sizes would help determine if this is just a brief glitch, or if this unusual pattern would persist to larger matrix sizes.

A curious feature of the data is that the different access patterns do not begin to really diverge until around 300x300. This data was generated with the default mode for the Visual Studio 2015 compiler, so it is very possible that debugging features were left enabled in this test, which would create a larger overhead. This overhead will be almost constant for all the datasets, and would drown out small differences in performance. Once the matrices become large enough, the performance differences between different access patterns do become more obvious.

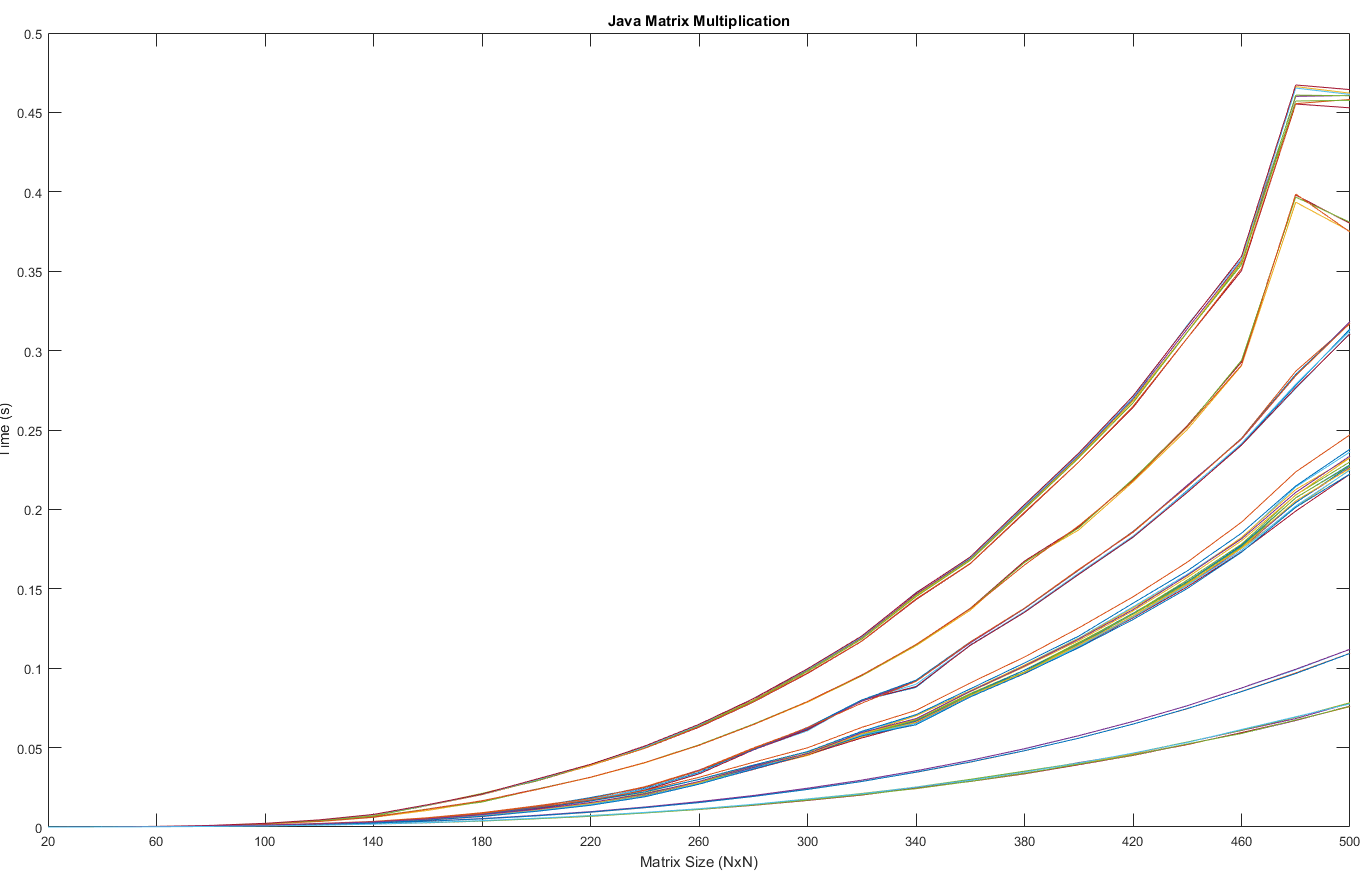


In this graphic the separation in performance between best and worst performance is more apparent. The other surprising piece of information we learn here is that the fastest and slowest access pattern for C++ both had the same ‘for’ loop configuration, so performance benefits are completely down to the inner op indexing order.

The jagged results visible in the previous graphs do show up again here, and of interest is the difference in consistency between the fastest and slowest access patterns. The fastest stays smooth all the way through the test, whereas the slowest pattern behaves in a jagged manner from 380x380 onwards. These jagged features of course then turn up in the Q factor plot as well, although it is still possible to discern the average shape of the plot.

The Q factor for C++ is not at all in line with what would be expected from a compiled language. The 80-40-20 rule suggests that C++ would be in the 60-80% range, but we instead see that C++ is really more in the 15-45% range. In this case, this is due to the Microsoft CL compiler being in debug mode with no compiler options enabled.

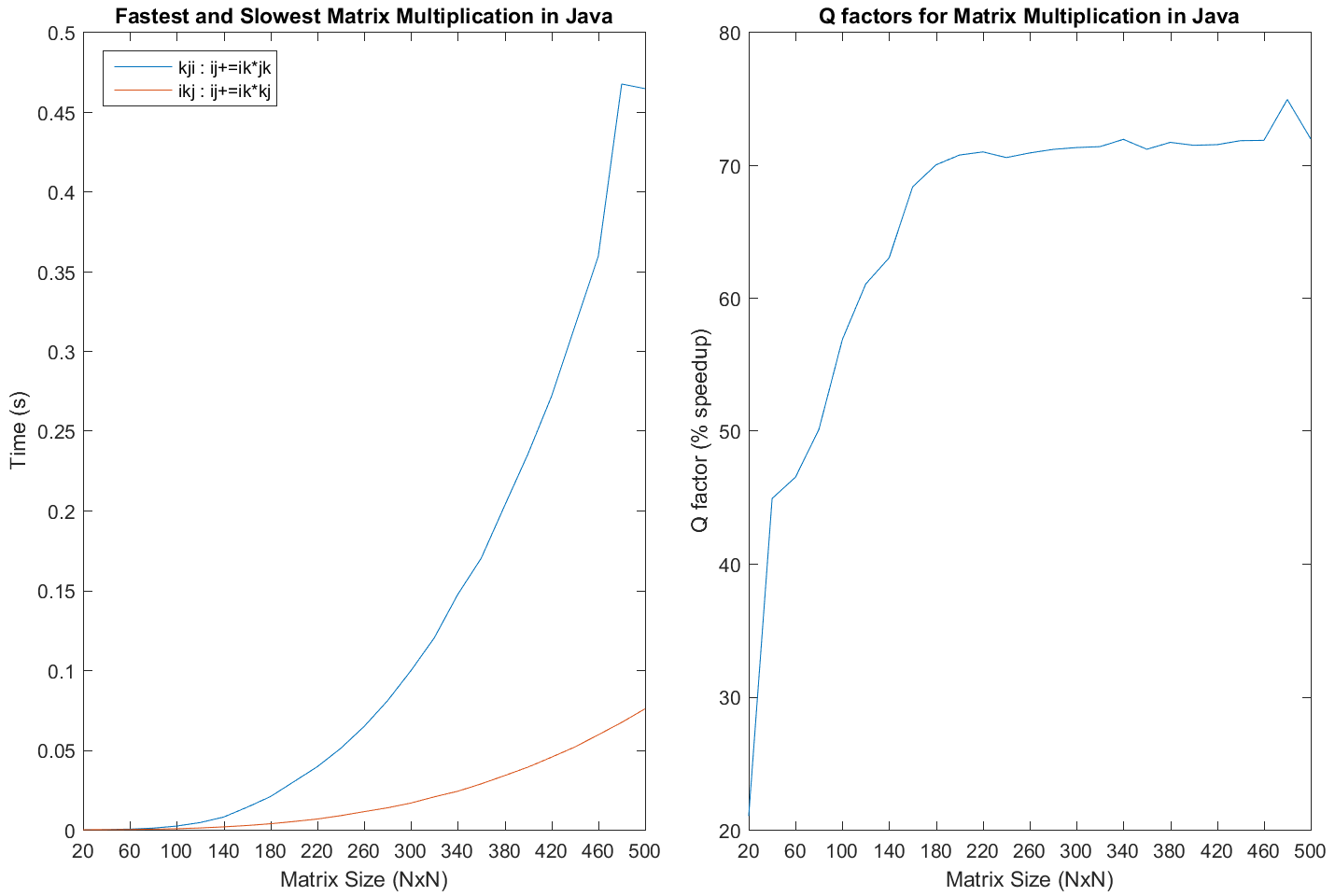
# III. Java Data



The Java data is characterized by having good separation between each access pattern, which means that these access patterns do have a significant effect on program performance. Of interest is the unusual behavior present in the few slowest patterns, where from 480X480 to 500x500, performance even seems to speed up or stay the same. This could be due to compiler settings or peculiar behavior of the processor clock.

In order to attenuate random behavior, the tests were run a minimum of 100 times. While this caused testing to require an extended period of time to complete, the smooth results clearly show the expected 6 clusters of data. It appears that most of the access patterns fall into the middle cluster, or the slowest. Most of the other groups, including the fastest, only contain two or three access patterns. The extremely small spread within clusters demonstrates that different access patterns either are very similar, or dramatically different in performance.

Aside from the aberrant results for slow patterns at the largest matrix size, all data seems to be smooth and consistent. Earlier data gathering tests with lower minimum test counts often had a fair amount of random noise in the data. In a future test, it would be of value to run these tests on a more completely controlled operating system, such as a clean install of Linux. Here, outside influences can be more easily controlled. Another variable not under control in these tests is the processors clock speed, which may change in response to the load of the test.

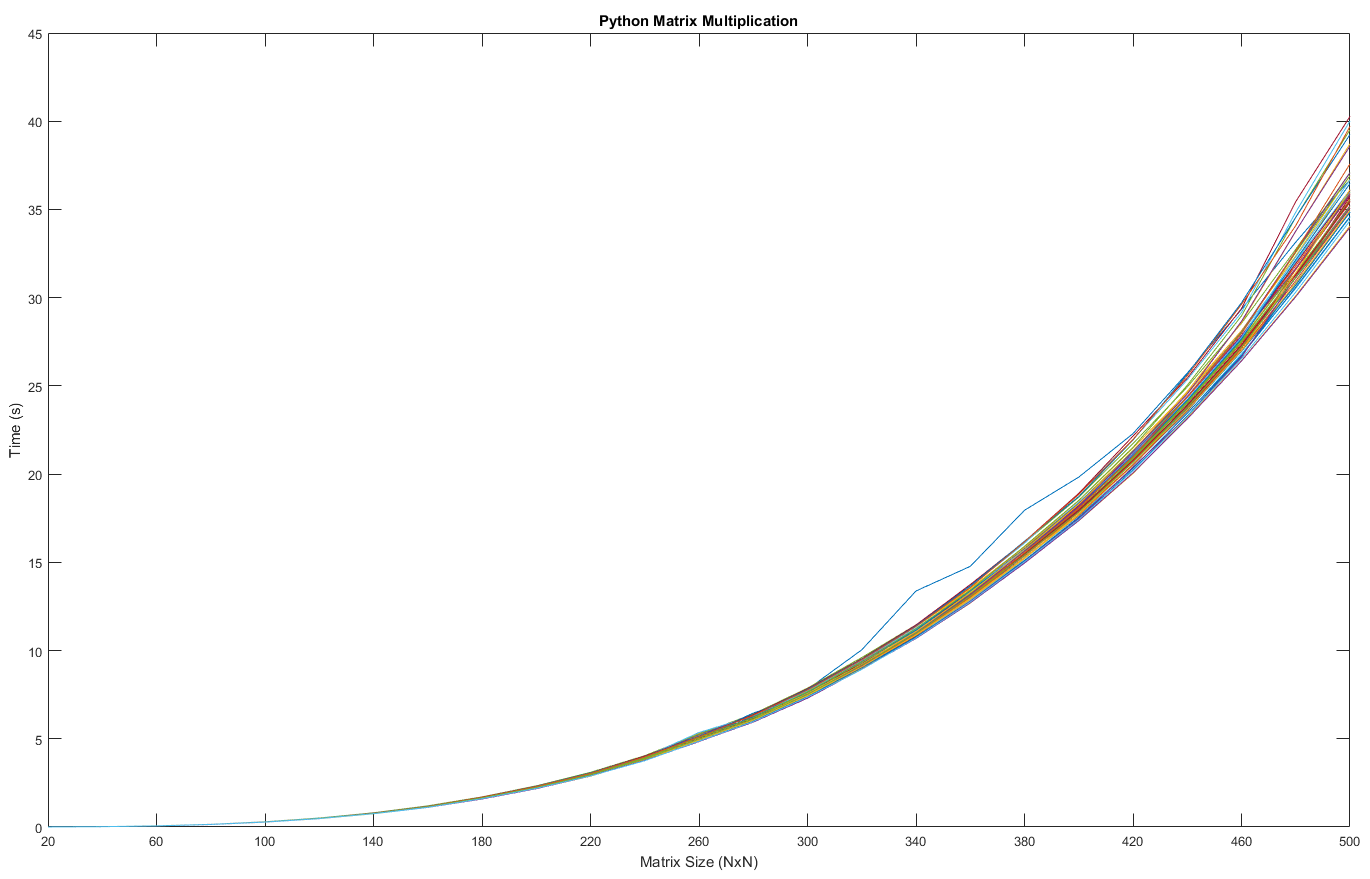


Java demonstrates extremely consistent performance. The best and worst performing access patterns have very different ‘for’ loop configurations, which indicates that access patterns did have a significant effect on the execution time. The Qfactor quickly climbs to 70%, which means that performance benefits of the superior access patterns rapidly outweigh the overhead of the rest of the code, and the Java virtual machine.

The Qfactor converges to about about 72%, which actually puts Java into the range of languages which compile straight to machine code. There are many possible reasons for this. One is that the JVM has been highly optimized for many years, so it’s very possible that most of the low level operations in this simple matrix multiplication program go straight to machine code. This is reasonable because the program does not utilize anything higher level than for loops, math operators, and arrays.

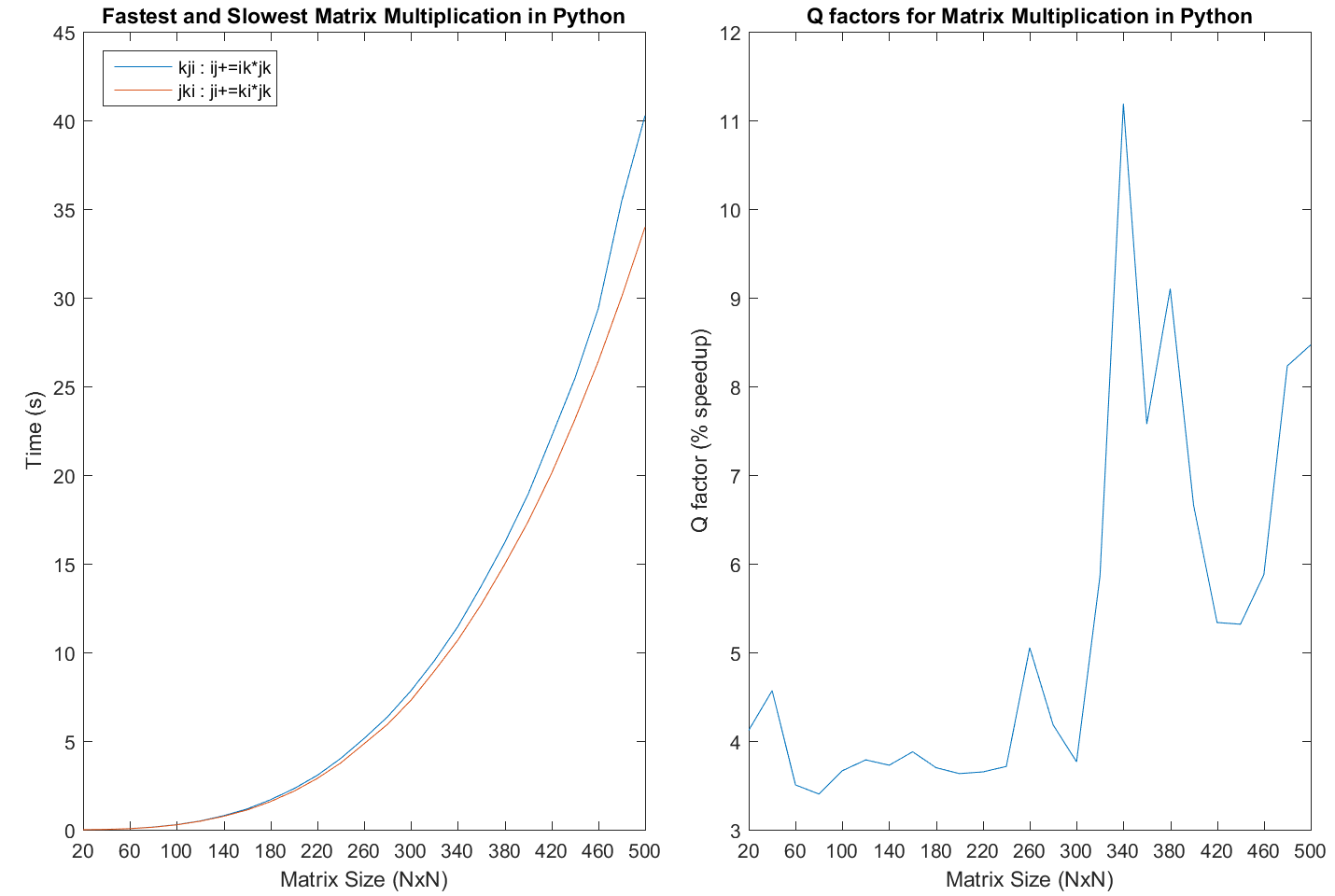
Given more time, testing out to matrix sizes of 750x750 or 1000x1000 would allow help determine exactly what the Qfactor converges to, and what the nature of the unusual behaviour at 500x500 is.

# IV. Python Data



The python results seem to be tightly clustered until the matrix size becomes quite large. At the largest matrix sizes, the data begins to separate into a few discernible groups, but the separation is minimal. There is one curve that appears to have unusual behavior for a few data points, but since the minimum test count for python was only 1 test. This is due to the extraordinarily long runtimes of the python code, taking often between 35 and 40 seconds to complete a single multiplication on large matrices.

While the data is clustered quite tightly, there is an observable spread towards the end, with the data separating into a few groups. This suggests that the access patterns do have some effect as the matrix sizes grow large in size. One hypothesis was that Python was so overhead-heavy, these improvements may not even have been noticeable, but this separation shows that this is not the case. Indeed, testing to even larger matrices, while prohibitively time intensive for this cumbersome language, would help to demonstrate the benefits of modifying access patterns.

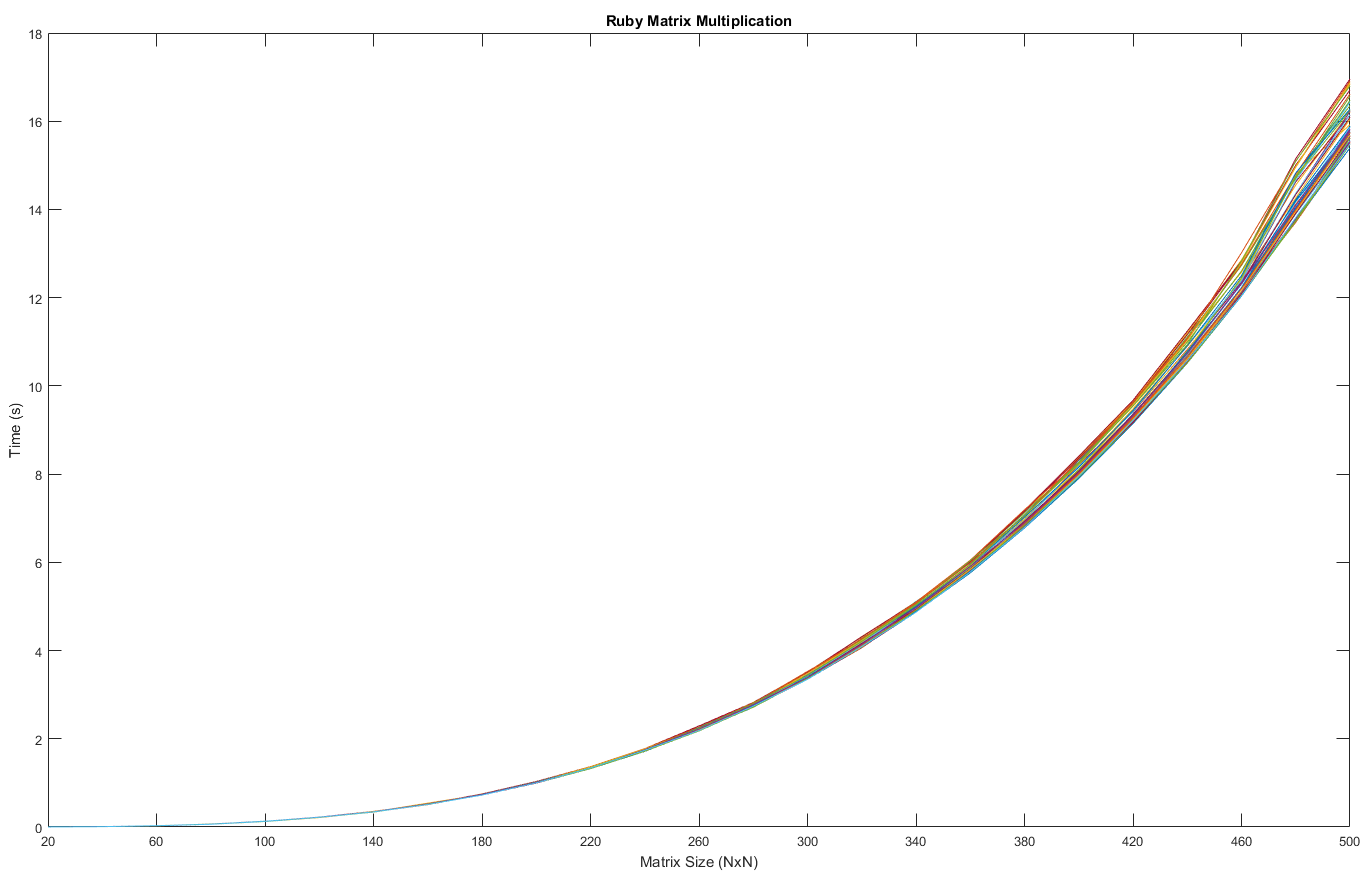


Python performance is elucidated in these charts. The first things to notice is that the fastest and worst access patterns are indeed different ‘for’ loop configurations, in addition to being different inner op permutations. This indicates that these adjustments do make a difference, if small.

In the Qfactor plot, there is an unusual feature from 320 to 380, which is caused by the aberrant dataset in the previous plot. It is apparent from the behavior of this chart that, while poor, the performance improvement due to changing access patterns is steadily improving all the way to the largest matrices. This means that the overhead from the python interpreter is quite prodigious, for the Qfactor to not show any real asymptotic behavior even for large matrix sizes.

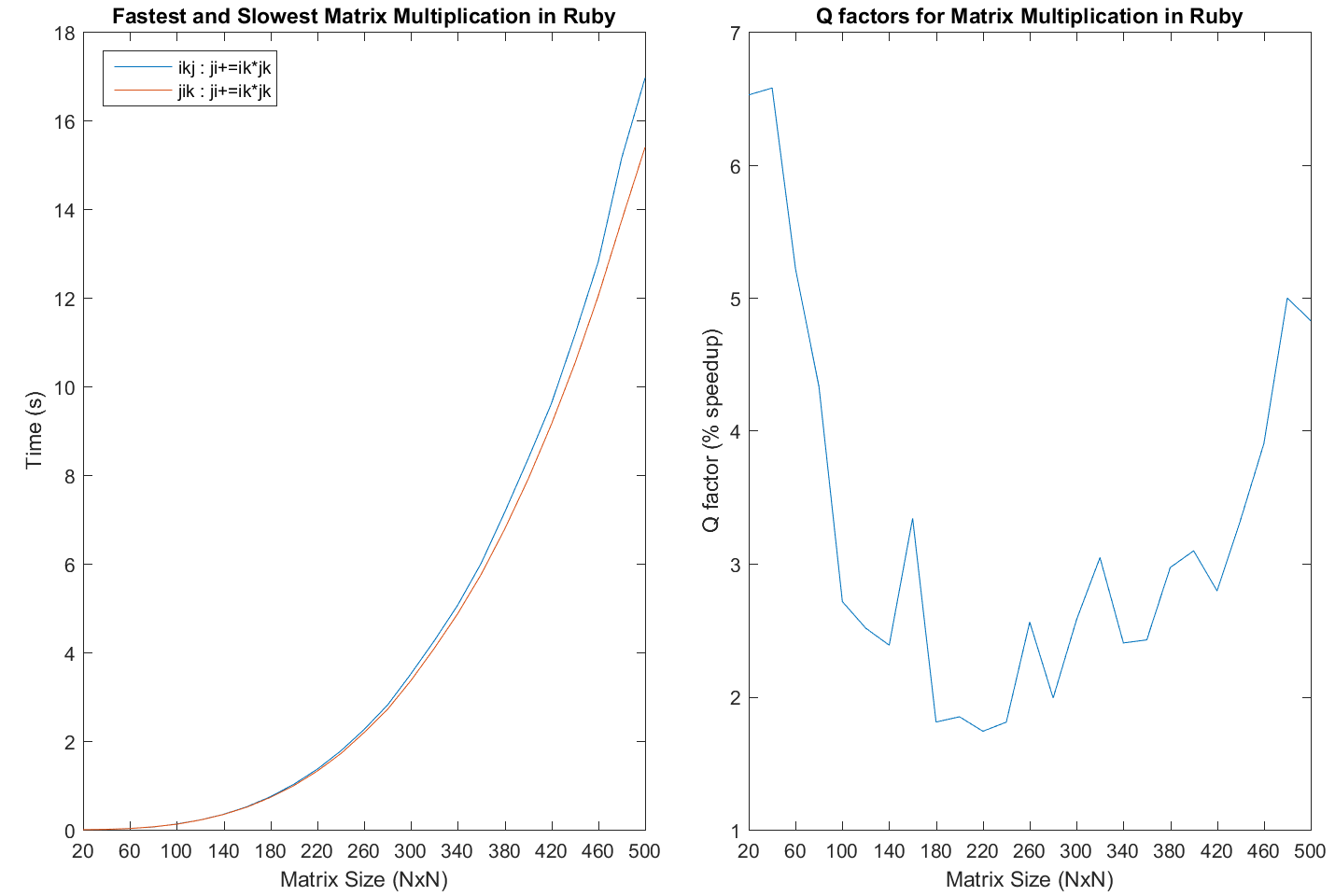
In terms of the 80-40-20 rule, a test to larger matrix sizes would help reveal the precise convergence value for the Qfactor, but this plot suggests that 10-15% would be possible. This would be quite reasonable for an interpreted language such as python. The main takeaway is definitely that access patterns should not be of much concern to the Python programmer, except for large datasets. Any improvement on a small set would be marginal at best.

# V. Ruby Data



The Ruby data is uniquely interesting because it shows almost no spread at all, even to the largest matrix sizes in this test. At that point, three groups become barely discernible, with tiny gaps between. This immediately indicates almost no improvement from memory access patterns.

This data was collected with a minimum of three executions per matrix size, so the resulting data looks appropriately smooth with no unusual values. One point of interest in the code itself is that the built-in timing for Ruby within the Microsoft windows environment is only really accurate to within 0.1ms, which is in stark contrast to the ~450ns precision offered by the other languages, which all leverage Microsoft’s Query Performance Counter API. As a result, the factor for determining minimum execution time is set higher than the other interpreted languages, to ensure that any given test length will keep timer error under 0.1%. For these tests, that factor is set to 250, so any matrix multiplication tests under this should take the same amount of time to complete. Above this size, and the minimum test count of 3 is utilized to acquire a reliable average.

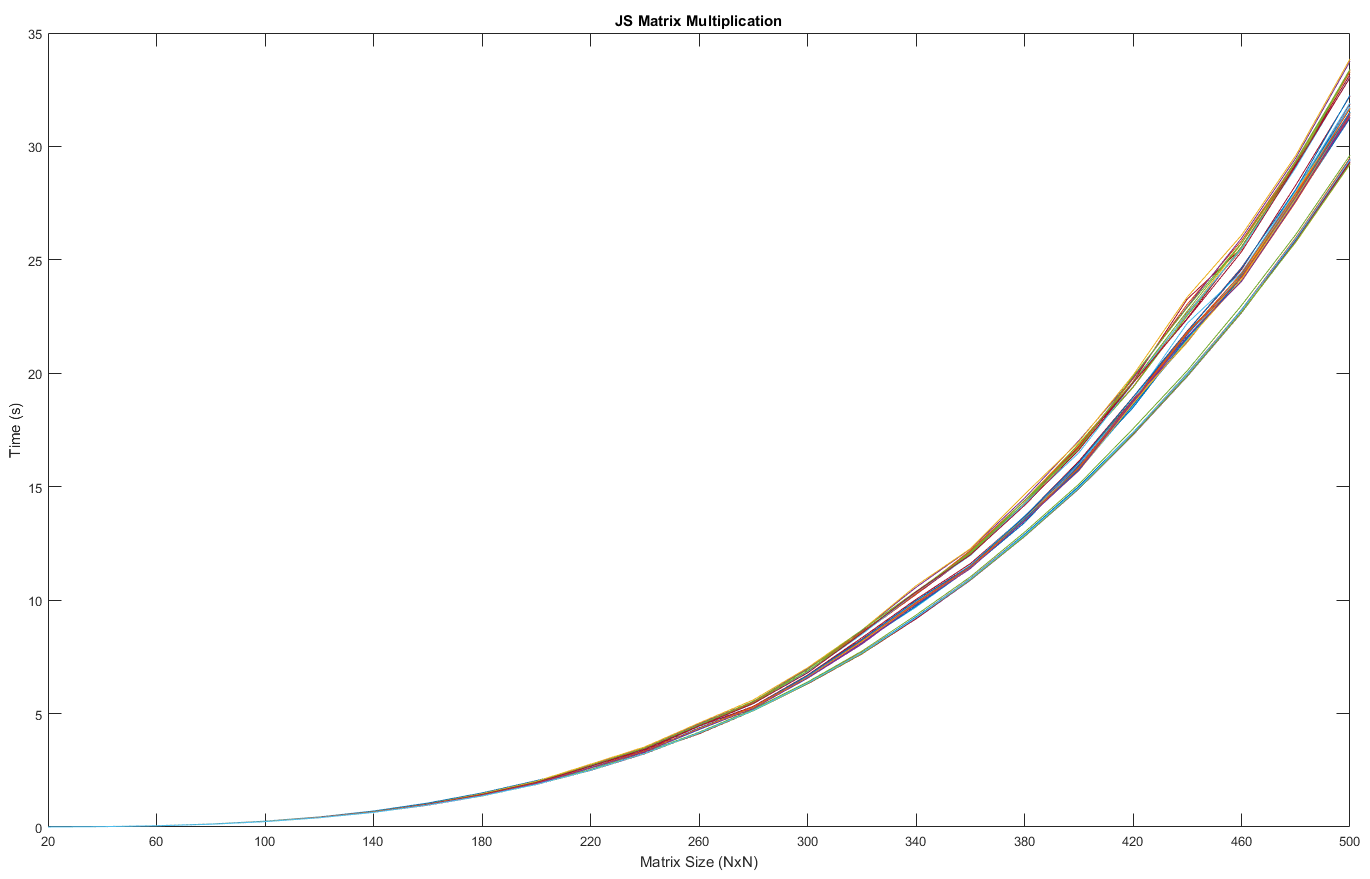


Comparing the best and worst performing patterns in Ruby, it is evident that the difference is very slight. While the access patterns are indeed different, the performance benefit stays between 2-5% for most matrix sizes. Of interest is that while the ‘for’ loops do change between the fastest and slowest patterns, the inner op configuration is the same. This suggests that the Ruby interpreter may be doing something underneath the code that causes there to be no tangible benefit to different configurations of the inner op. A further investigation would be to determine which inner ops tend to do better than others, if that is the case at all.

Another peculiar characteristic of the Ruby dataset is the Qfactor plot versus matrix size. From very small matrix sizes until about 220x220, the Qfactor actually seems to go down, which is very unusual. Python exhibited a bit of this behavior as well, but to a far lower degree. Once Ruby hits this low point, it begins to go back up, but never quite hits the Qfactor that it achieved for 20x20 to 40x40 matrix sizes. What causes this is most likely the interpreter switching the manner in which it handles memory addressing. For small matrices, the interpreter may choose an addressing schema that favors small array sizes. As the matrices grow in size, this schema becomes less and less efficient. Ultimately, the interpreter switches to a schema that handles large array accesses more gracefully, and the resulting improvements are visible from 240x240 matrices onwards.

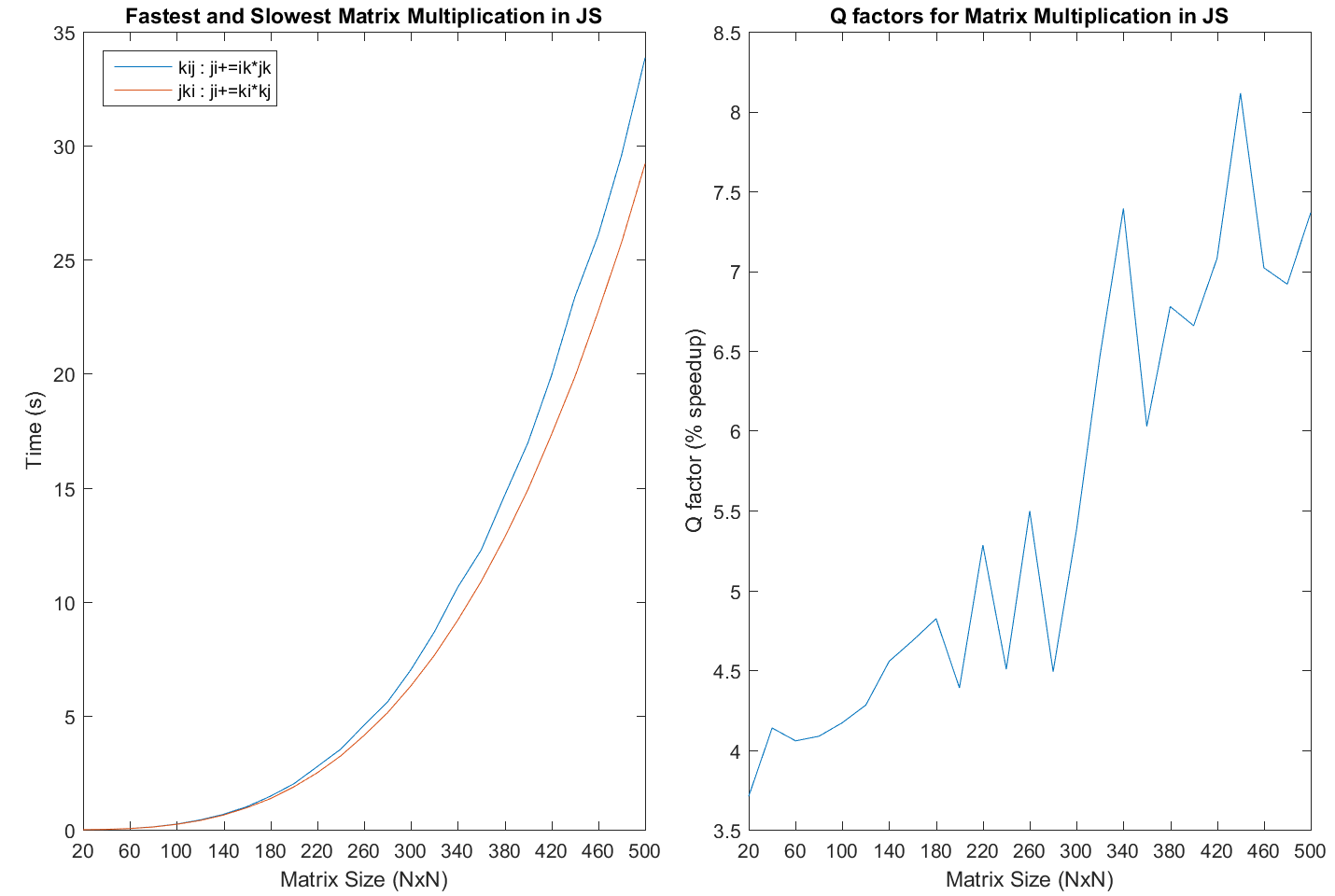
Ultimately, a 5-6% performance improvement at best means the average programmer would be better off looking elsewhere for performance gains in Ruby.

# VI. JS Data



Unlike Python and Ruby, JavaScript behaves in a much more predictable manner. The data separates into 3 distinct groupings by 300x300 matrix size. This mean JavaScript produces meaningful performance benefits, relative to interpreter overhead, relatively quickly. These 3 clusters stay close to each other to the end of the test. One of big variables to consider with JavaScript is the engine used to run the code. PhantomJS was used in these tests, which uses JavaScriptCore. Just one engine gives only a small snapshot. In future tests, adding additional engines such as Microsoft Chakra, Google V8, Netscape SpiderMonkey, and Mozilla Rhino, which are all commonly used engines in various browsers and webservers. In the context of this paper, the performance of PhantomJS will be satisfactory for basic comparative analysis to other languages.

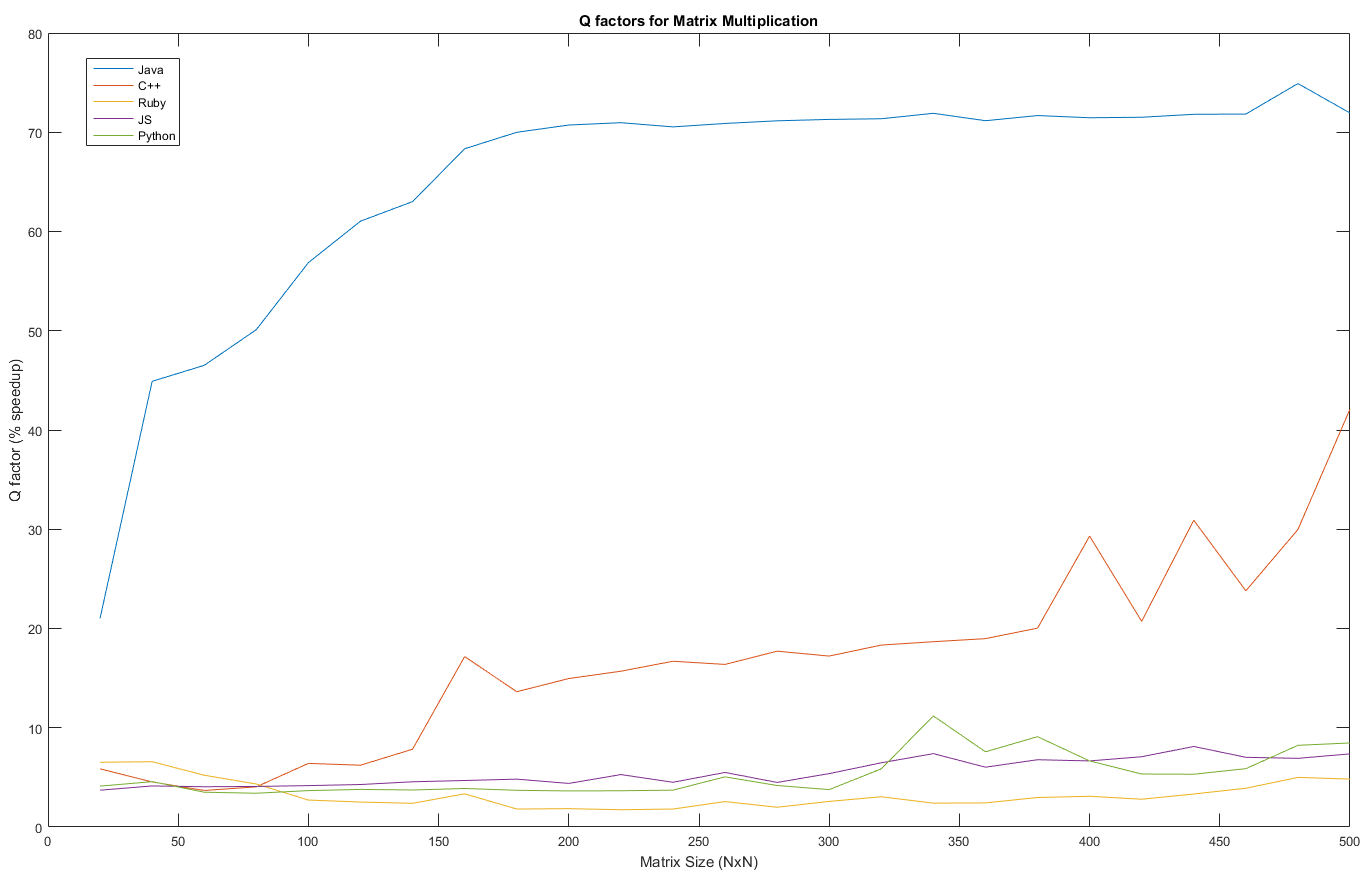
While JavaScript seems to spread out in performance faster than its interpreted cousins, it is still quite far off from the characteristics of the compiled languages. With regards to matrix multiplication, the bulk of the interpreter drowns out almost all performance benefits from varying access patterns until the matrices become quite large. The long test times do result in relatively clean data, with no obvious aberrations.



JavaScript came in between Ruby and Python in terms of overall performance, although this not really a fair comparison without testing additional engines and configurations for each language. Again, the fastest access patterns deliver somewhat smoother looking data, while the slowest configuration seems to have somewhat irregular data.

The Qfactor data is a bit jagged, but there is a clear correlation between increasing matrix size, and improved speedup from access pattern configuration. These jagged features are most likely a result of some access patterns being slower than others at one matrix size, then switching places at another. JavaScript shows a strong climb from 4% to 7-8% at the largest sizes, and no strong sign of converging to an asymptotic value. This suggests that Javascript does not do much memory access optimization in the interpreter, which results in the variance in results between access patterns.

# VII. Q Factor Analysis

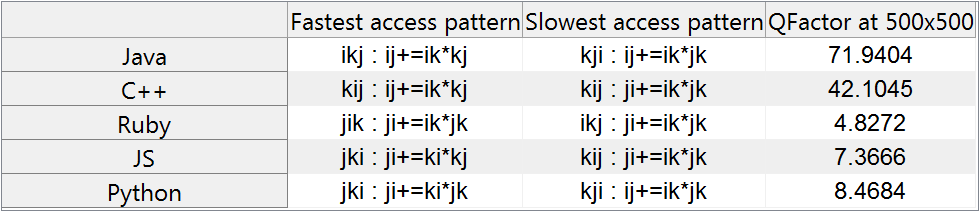


Once all the Qfactors are together for reference, it becomes clear that the sometimes unusual behaviours of the interpreted languages is relatively insignificant gulf that separates the interpreted from the compiled. Java builds up to a massive lead, converging asymptotically to just above 70%, while the interpreted languages hover in the 4-9% range.

Another interesting feature is the behavior of C++ in debug mode, which slowly increases from interpreted territory, and steadily climbs towards 40%. This is clearly a symptom of the latent debugger addons in the machine code. With these included, C++ slowly gets greater returns from speedup, but never quite reaches the level of release-version compiled languages.

It is difficult to identify any significant difference between the interpreted languages, but speedup in Python and JavaScript appear very close, with Ruby just a bit lower. The unusual ‘U’ shape of the Ruby Qfactor is also evident, and not expressed by any other languages Qfactor. This is most likely due to memory optimizations in Ruby, which may also cause it to have the best overall performance in terms of execution time, out of the interpreted languages.

**Access Pattern Results:**



One constant

One other observation at this point in the analysis is that as the matrix sizes increase, the interpreted languages tend to split into 2-3 tight clusters, versus 6 distinct groups for the compiled languages. This indicates that the compiled languages can take more advantage