A Comparison of Numerial Linear Algebra Algorithms in Various Programming Languages

Alex McNurlin

December 6, 2017

1 Abstract

A large variety of programming languages exist, each with their own purpose. That also means they each have their own strengths and weaknesses. Scientific computing languages, like MatLab and Fortran, are made to work with matrices and perform common mathematical operations. They are also made to be more user friendly, which can come at the cost of speed. C, on the other hand, is a popular language for software that runs closer to the hardware. Therefore it is made to be more fast and lightweight. This comes at the cost of usability.

This paper shows a side by side comparison of 4 programming languages, C, Fortran95, Python, and Octave. For each language, the speed and numerical accuracy is observed using implementations of 3 different numerical algorithms used in scientific computing. The algorithms are LU Factorization, QR Factorization using the Modified Gram-Schmidt method, and Cholesky Factorization to solve the Least Squares problem.

The tests showed that compiled languages like C and Fortran95 were generally faster than Python and Octave. However, the functions used by the Python library NumPy were sometimes faster than C or Fortran.

Additionally, increasing the accuracy of floating point arithmetic had a significant effect on the runtime.

2 Introduction

MatLab and Python are both popular choices for scientific computation. MatLab is power for matrix algegra, and is a popular choice for engineers. Python is free, open source, and becomes powerful for scientific computing when paired with other libraries. However, they are both interpreted languages, with are often slow, relative to compiled languages like C and Fortran.

The difference in performance is not usually obvious to the user, since each language is used for different purposes. This paper aims to give a side by side comparison of each language performing the same tasks, and comparing the resultant speed.

For this project I have chosen 4 different programming languages, and well as 3 additional variations on those languages The languages chosen and reasons why are discussed in the 'Languages' section. In each language, I implemented 3 different algorithms. The algorithms chosen are discussed in the 'Algorithms' section. Each language/algorithm pair was run against 2 matrices: a small 3x2 matrix with entries near $1 + \epsilon_{machine}$ to observe the numerical accuracy of each language, and a large, randomly generated, 375x350 matrix to observe the speed. The result for those are discussed under 'Results' in the 'Accuracy' and 'Performance' sections, repsectively.

Languages

The languages and variations on those languages used are listed below.

1. **Python (with NumPy)**: Python is a popular language for scientific computing, and very robust for software development. For the rest of this paper, 'Python with NumPy' will simply be referred to with 'Python'

2. **Python/NumPy builtin functions**: NumPy provides built in functions to perform common operations. These have usually been heavily optimized to be the

fastest and most efficient. Some functions use LAPACK, a matrix algegra library originally written in Fortran, as their underlying implmentation. To disambiguate NumPy builtins from algorithms implemented in Python, algorithms implemented using NumPy builtins will be simply referred to with "NumPy"

3. **Octave**: Octave is a free and open source clone of MatLab. However, MatLab is not free, so Octave is a good alternative. Octave is good for scientific computing as it has native support for matrix algebra. However, Octave is made to have the same syntax as MatLab, without as much attention to performance, so this will likely be the slowest language.

4. **Fortran95**: Fortran is a compiled language made for scientific computing. It was first released in 1957, (more than 10 years before C), so it is considered out of date. However, Fortran is fundamental in modern computing as it is the first compiled language. It is very well optimized, very fast for scientific programming, and supports native vector operations. All code described in this paper uses Fortran 95 (released in 1995) with the gfortran compiler. For the rest of this paper, implementations in this language will be referred to with 'Fortran95'.

5. **Fortran95 (quadruple precision)**: The 'double precision' type in Fortran95 defaults to a 64-bit floating point number. However, gfortran provides the ability to promote double precision values to 128-bit floating point numbers using the `-fdefault-real-8` flag at compile time. This allows for greater accuracy at the cost of speed. For the rest of this paper, 128 bit implementations will be referred to with 'Fortran95-d'.

6. **C**: C is a compiled language that is ubiquitous in software. C first appeared in 1972 as part of the UNIX operating system. It is often praised for it's speed, however it was not made for scientific computing. All C code is compiled using the GCC compiler.

7. **C (Optimized)**: C code can be made faster by using different options at compile time. For this paper, all optimized C code will be using the `-O3` flag at compile time, which provides the most optimization to reduce runtime speed. To disambiguate with the non-optimized C code, code compiled with the optimization flags will be referred to with 'C-optimized' or 'coptimized'.

Algorithms

Three different algorithms were chose to compare the languages. The algorithms were chosen to be representative of common numerical operations.
First is Compact LU Factorization without pivoting: This acts as a simple algorithm that becomes very slow for large matrices.

```
# LU Factorization in Python
for i in range(MAT_SIZE):
    for j in range(i+1, M_DIM):
        m = A[j][i]/A[i][i]
        for k in range(i, MAT_SIZE):
            temp = A[j][k] - m*A[i][k]
```

```
            A[j][k] = temp
        A[j][i] = m
```

Second is Cholesky Factorization to solve the Least Squares problem Ax = b for $A \in \mathbb{R}^{mxn}, m > n$. This is done by solving the system $A^TAx = A^Tb$ using Cholesky Factorization. This gives us the equation $HH^Tx = b'$ where $b' = A^Tb$. The vector x can be found by solving $Hy = b'$ for y, and $H^Tx = y$ for x.

```
# Cholesky Factorization in Python
if sys.argv[1] == "1":
    for k in range(MAT_SIZE):
        h_temp = sum(H[k, i]**2 for i in range(k))
        H[k, k] = sqrt(A[k, k] - h_temp)
        for i in range(k+1, MAT_SIZE):
            sum_h = sum(H[i, j]*H[k, j] for j in range(k))
            H[i, k] = (A[i, k] - sum_h)/(H[k, k])
```

Lastly is Reduced QR factorization using the modified Gram Schmidt method.

```
# QR Factorization in Python
for k in range(MAT_SIZE):
    r[k][k] = norm(A[:, k])
    q[:, k] = -A[:, k]/(r[k][k])

    for j in range(k+1, N_DIM):
        r[k][j] = q[:, k].dot(A[:, j])
        A[:, j] = A[:, j] - r[k][j]*q[:, k]
```

Measurement Procedure

To measure the performance of each program, the `time` utility was used. Each program was compiled individually (if necessary). The matrix used was serialized and passed with the matrix dimensions as a command line argument using `xargs`. For example, the Python implementation of LU Factorization was run with `xargs --arg-file my_matrix_file time python/lu_factorization.py`, where `my_matrix_file` is a file containing the matrix in question and it's dimensions. The output of each program was stored, and the runtime reproted by `time` was stored separately. Additionally, a baseline run was recorded for each program, where only the time to read in command line arguments and setup the inital matrix was recorded. This was to understand how much time went to program overhead, and how much went to computation. Each program was run 5 times and the average runtime was recorded. The results can be seen in figures 1-3.

3 Results

Accuracy

To test the accuracy of each language, the following 3x2 matrix was used.

$$A_{small} = \begin{bmatrix} 1 & 1 \\ \epsilon_1 & 0 \\ 0 & \epsilon_2 \end{bmatrix}$$

With $\epsilon_1 = 1+2^{52} \approx 1.0000000000000002220446$ and $\epsilon_2 = 1+2^{53} \approx 1.0000000000000001110223$.

These numbers were chosen to be near machine epsilon (2^{52} for 64-bit numbers) to easily show which languages use 64 bit numbers and which use 128 bit. Octave and Python each use 64-bit floats by default, while C and Fortran95 allow you to specify. For the C and Fotran95 code, 64-bit floats were used, while Fortran95-d used 128-bit.

Since Fortran95-d has the highest precision, it will be used as the example for the following section.

LU Factorization

Fotran95-d gave the following result

$$A' = \begin{bmatrix} 1.00000000000000000000 & 1.00000000000000000000 \\ 1.00000000000000022204 & -1.00000000000000022204 \\ 0.00000000000000000000 & -0.99999999999999988898 \end{bmatrix}$$

NumPy gave a different answer, becuase it uses partial pivoting. All of the other languages had a slightly different element in the 2nd element of the 3rd row.

Least Squares Solution using Cholesky Factorization

Fortran95-d gives the solution

$$x' = \begin{bmatrix} 0.66666666666666662966 \\ 0.66666666666666666667 \end{bmatrix}$$

Each other language gave something similar but with less precision. The most surprising part is that each language gives an answer that varies after a different number of decimal places, rather than all being the same. C and C-optimized had the lowest precision, and Fortran95-d had the highest.

QR Factorization using Modified Gram-Schmidt

This one had less variation in the floating point arithmetic. Fortran95-d gave the answer

$$Q = \begin{bmatrix} -0.7071067811 & -0.408248290 \\ -0.7071067811 & 0.408248290 \\ 0 & -0.816496580 \end{bmatrix}, \text{ and } R = \begin{bmatrix} 1.4142135623 & -0.7071067811 \\ 0 & 1.2247448713 \end{bmatrix}$$

Each language yielded something very close to that, with varying levels of precision.

Performance

Figures 1-3 show the runtime of each of the programs plotted on a logarithmic scale. The total runtime represents the total time the program was being run (obviously). Computation runtime represents the amount of time spent actually performing computations, rather than reading in command line arguments and setting up matrices.

In most of the tests, C and C-optimized were the most efficient, with Fortran95 close behind. Surprisingly, Fortran95-d was nearly an order of magnitude slower than Fortran95. It was expected to only be half the speed, not 1/10th. That's likely because this was run on a 64-bit computer, which doesn't support 128-bit arithmetic. Therefore, any 128-bit operations must be represented as a series of 64-bit operations.

C was the fastest in most cases. Despite not natively supporting matrix math, it was comparable to Fortran in most cases. Surprisingly, however, C-optimized was not any faster or slower than C. The most likely reason for this is a mistake was made when compiling the code, so no optimizations were actually performed. Fortran95 performed on par with C and C optimized. This is unsurprising since they are both compiled languages that boast being fast.

NumPy was the next slowest, with total runtime comparable to Fortran95-d, but computation time was between Fortan95 and Fortran95-d. It was especially fast in the QR Factorization. The low computation time was likely because the NumPy QR factorization function uses LAPACK, which is highly optimized. This is the only NumPy algorithm in this paper that utilizes LAPACK.

Python was slower than NumPy in most tests. This is expected, as it lacks the optimizations of NumPy. In each of the tests, it took a decent amount of time to set up the matrices. This extra time to setup is the key disadvantage of interpreted programming languages. The extra setup time was nearly a second if every case. For a program that isn't run often, that is trivial. However, for a program that is run often, that can be very significant.

Octave was the slowest in every test. This was expected, as Octave is likely not very optimized for speed, especially when compared to MatLab. The most interesting part was found in the Cholesky Factorization test, Octave had a similar computation speed to Python, but the total runtime was much larger. Cholesky Factorization requires doing the multiplication $A^T A$ before the computation. For a 357x350 matrix, that requires multiplying 350 375-vectors. This shows that Octave performed the worst in matrix multiplication.

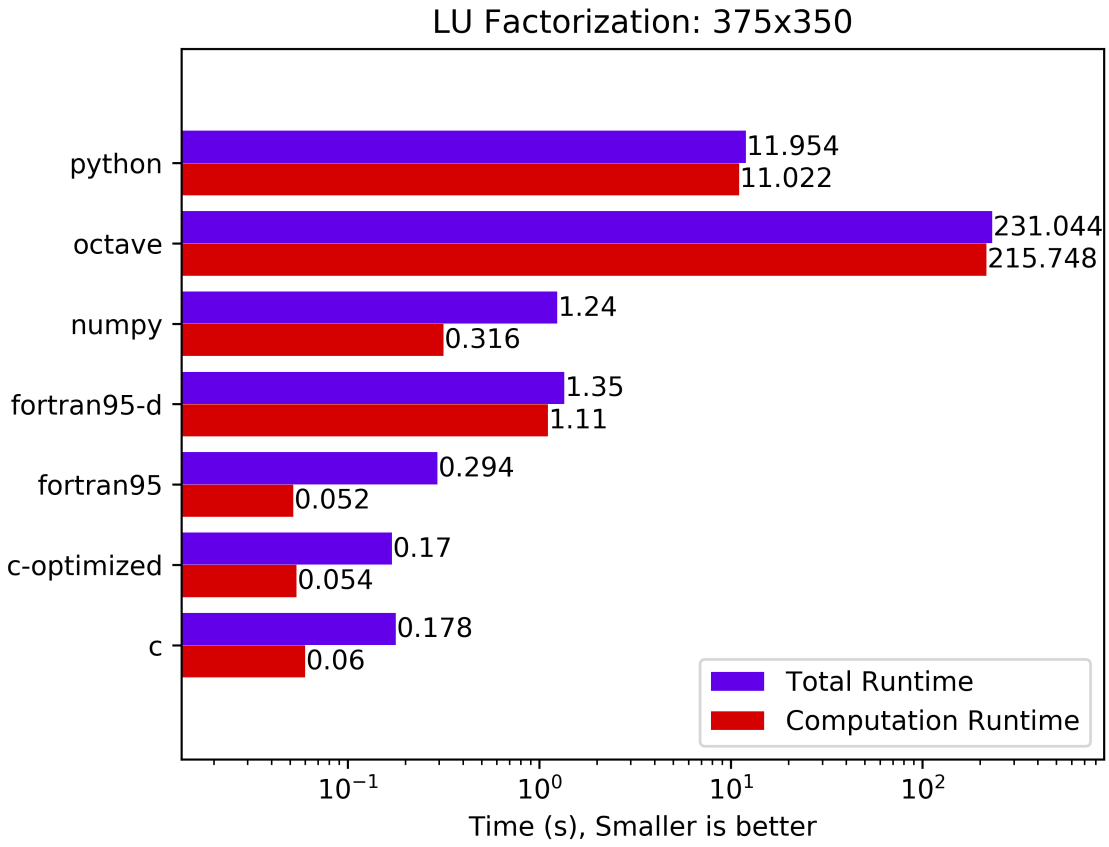

Figure 1: Time taken to solve the system $A = LU$. The total runtime is the total time taken by the program, and the computation runtime is the total runtime minus setup time, such as reading command line arguments and allocating the matrix

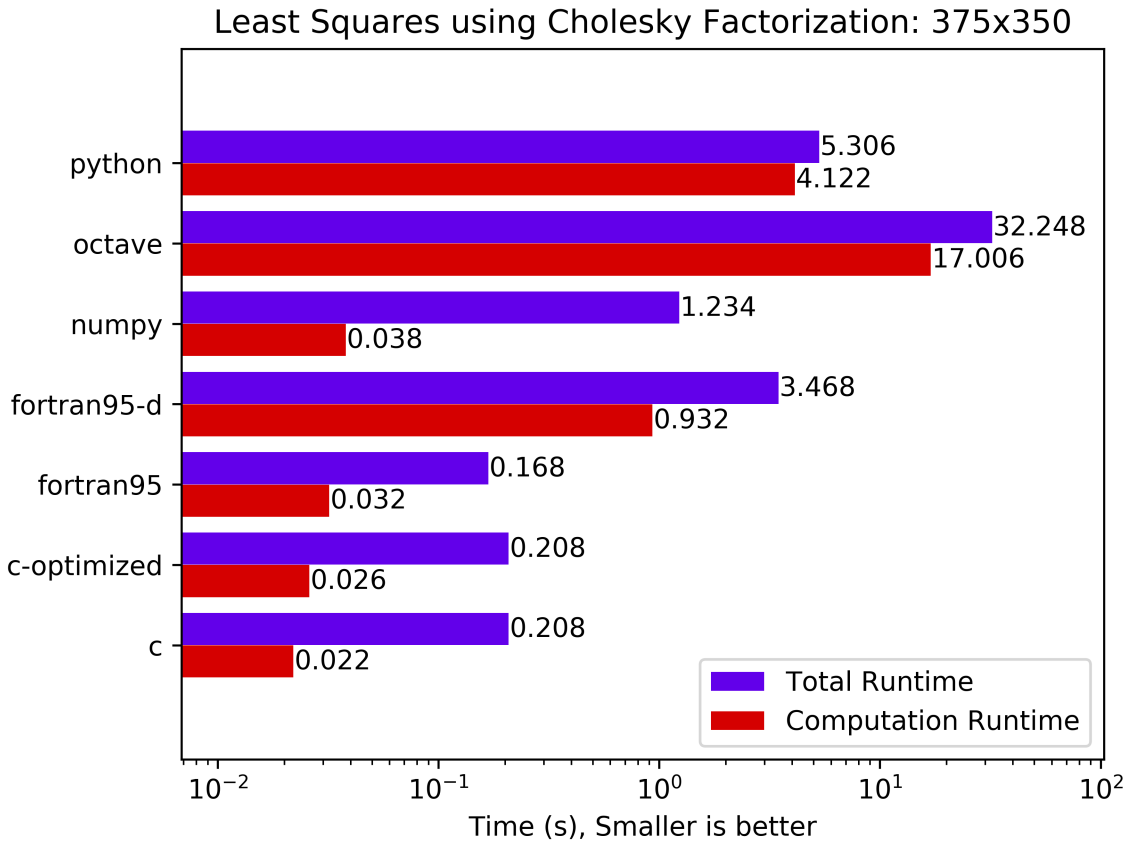

Figure 2: Solving the system Ax=b where A is a tall matrix. The total runtime is the total time taken by the program, and the computation runtime is the total runtime minus setup time, such as reading command line arguments and allocating the matrix

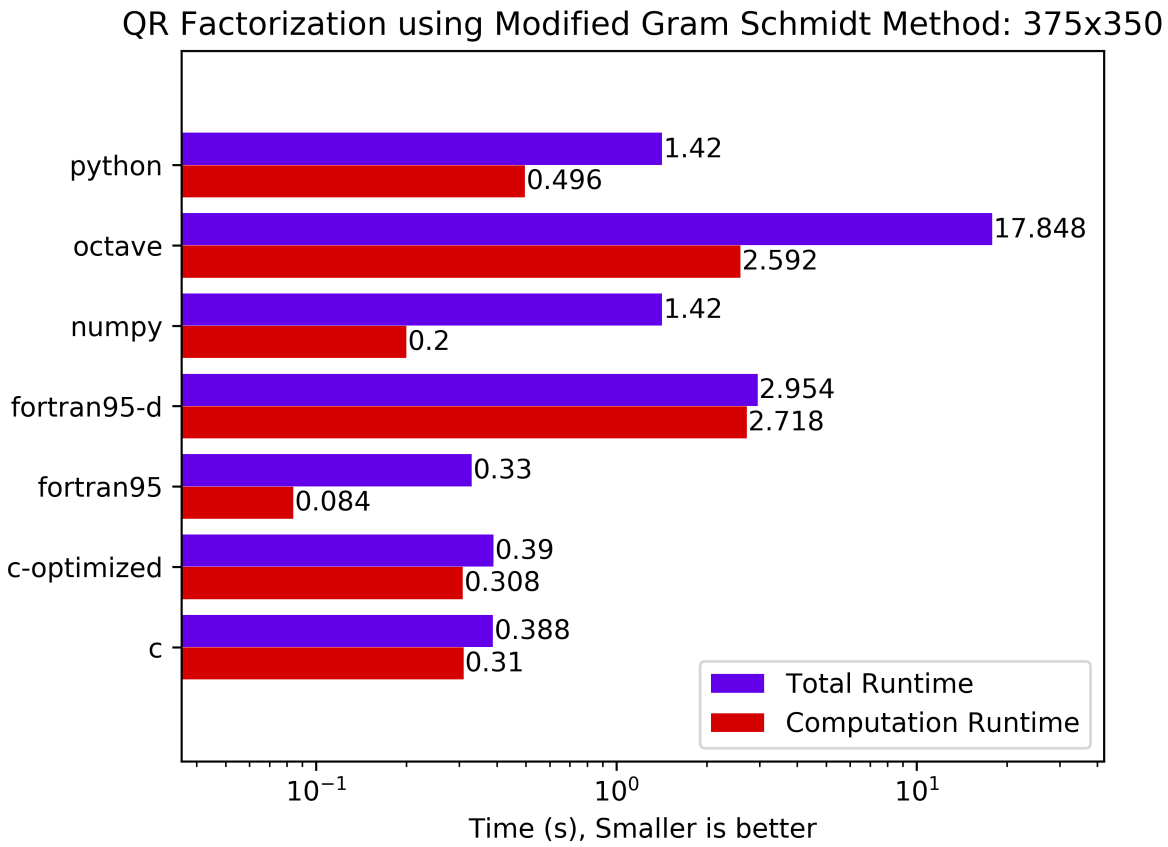

Figure 3: Solving the system A = QR using the Modified Grahm Schmidt method. The total runtime is the total time taken by the program, and the computation runtime is the total runtime minus setup time, such as reading command line arguments and allocating the matrix

4 Discussion

This section will be used to say a few things in a less formal tone, or to say things that I couldn't fit well into other sections. Also, I would like to reflect on a few of the things that I didn't do in the project.

1. I'm most surprised that C and Fortran were similar in speed. I expected Fortran to be quite a bit faster, since it has vector and multiplication built in.

2. Octave was slow. Painfully slow. When I was testing my environment for running the scripts, I usually disabled the Octave scripts in the Makefile, since I ran each program 5 times, and octave took up to 230 seconds per run, including the baseline.

3. I would have liked to try this with even larger matrices (1000x1000 or more). I chose the 375x350 matrix because xargs would only allow about 2,000,000 characters in command line arguments. I also would have like to use multiple large matrices, to see how the runtime increased with the size of the matrix (For example, I believe LU factorization has an efficiency of O(n$\hat{3}$), which can become a huge problem)

4. I would have liked to use MatLab instead of Octave. I didn't use MatLab simply because I couldn't run it natively on my computer. Octave is useful when MatLab isn't available, but for large calculations it's painfully slow!

5. I also would have liked to lear Julia and compare it to the other languages. It's syntax borrows the best things from MatLab and Python, and claims to be comparable to C and Fortran. It also can be run as an Interpreted language, or precompiled for more speed.

6. Lastly, I would have also liked to compare the memory usage of each program. `time` allegedly records memory usage, but it always reported 0 Kilobytes, even though a 375x350 matrix of 8-byte floats should use 1,500,000 bytes