Math 477/577 Project

INVESTIGATING STABILITY OF MATRIX DECOMPOSITION AND LINEAR SYSTEM SOLVER USING QR, SVD, AND CHOLESKY

December 22, 2018

Joanna Czopek, Parker Joncus, Hailey Huong Nguyen, Kin Hang Wong

Contents

Abstract

This research project discusses the stability of matrix decomposition and linear system solver algorithms in Python. We choose to study three common matrix decompositions: QR Factorization (QR), Singular Value Decomposition (SVD), and the Cholesky decomposition (Cholesky). We will focus on the symmetric, positive-definite covariance matrices which frequently arise in machine learning problems. They are generated using Gaussian radial basis function with some additional small noise in the diagonal to mimic real life noisy data.

By conducting numerical studies of comparative backward and forward error analysis of the various decompositions and solvers algorithms for the linear system, we can observe how the size of such matrix affects the stability. We also study the scalability of the algorithms, which includes the run-time required from a computer.

At the end, we concluded the best algorithm is Cholesky decomposition which has minimum level of forward error and comparatively fast runtime when solving a linear system which associated with such covariance matrix.

Introduction

Matrix decomposition is to factorize a matrix into a product of matrices to perform matrix operations more efficiently. In theory, every existing algorithm could factorize a matrix completely without error with limited operations, or perform infinitely many iterations to converge. Unfortunately in real world, we have to use computers to perform the algorithms. Therefore, error is inevitable because every number are stored in a computer are based on floating point representation which is an approximation. For example, 1/3 cannot be stored exactly because it is a recurring decimal. In this project we will be investigating how the three matrix decompositions: QR Factorization, Singular Value Decomposition, and Cholesky Factorization; perform when one wants to solve the specific problem $A\vec{x} = \vec{b}$ where A is a covariance matrix.

The QR decomposition, $A = QR$, is for m x n matrices (not only limited to square matrices) where Q is unitary and R is upper triangular. There are many different methods that can be used for QR decomposition. Some of the popular methods are Classified Gram-Schmidt, Modified Gram-Schmidt, and Householder. In this project, we will use the Householder method.

Next, we have singular value decomposition, $A = U\Sigma V^*$, where U and V are unitary matrices and Σ is diagonal. Singular value decomposition offers extremely effective techniques for putting linear algebraic ideas into practice. The singular value decomposition was discovered independently by Beltrami in 1873 and Jordan in 1874 during their research on bi-linear forms. The existence of singular value decomposition means every linear mapping could be decomposed into preserving (V^*), stretch (Σ) and rotation (U)[2]. It is the generalization of the eigen-decomposition of a diagonalizable matrix to any m x n matrix which is the power of this decomposition.

Cholesky decomposition, $A = LL^*$, is applicable on symmetric and positive definite matrix A. The Cholesky decomposition is used for solving linear least squares for linear regression, as well as simulation and optimization methods. When decomposing symmetric matrices, the Cholesky decomposition is nearly twice as efficient as the LU decomposition and should be preferred in these cases. This is because Cholesky decomposition is similar to LU except since the matrix A is positive-definite and symmetric, we can have $U = L^T$. Cholesky decomposition is about a factor of two

faster than alternative methods for solving linear equations.

Runtime on computers mainly depends on two aspects: flop counts and memory. Flop count is the number of operations that an algorithm required the computer to perform. On the memory aspect, when considering the memory banks of computers, if the processor cycle is faster than the memory cycle, and memory consists of interleaved banks, consecutive elements will be in different banks. By contrast, taking elements separated a distance equal to the number of banks, all elements will come from the same bank. This is not an efficient memory management and it will reduce the performance of the algorithm. Our project will investigate how quick the computer solve the linear systems depends on the matrix size (scalability), and compare it to the theoretical runtime when one only considers flop count.

There is no doubt that accuracy is extremely important when solving a problem. Therefore, another goal of this project is to investigate the stability of the three matrix decompositions and linear solvers. By comparing the relative error on the three decompositions: QR, SVD, and Cholesky Factorization and their corresponding linear solvers, we could compare them to machine epsilon and find out which one is the most accurate. The accuracy on solving $A\vec{x} = \vec{b}$ will not be promising because covariance matrices are usually ill-conditioned, i.e. large condition number $\kappa(A)$, such that it leads an $O(\kappa(A)\epsilon_{machine})$ error.

This paper is comprised of a section to describe some background context of this project, a section to discuss mathematical ideas behind the experiments, and a section with experimental set up as well as results of computational experiments. We then conclude with an optimal algorithm for this special case together with future work to improve the outcomes.

Background Context

In many real world applications, there are linear systems of equations that need to be solved. This can be easy to solve directly if there are few variables and few equations, but when there are many variables and many equations, this becomes very difficult to solve directly. Therefore, there are many algorithms that are used to decompose a matrix into multiple matrices with special properties such that when

these matrices are multiplied together, they result in the original matrix. With the special properties of the decomposed matrices, we can solve the linear systems directly much easier and the solution will be the same for the original matrix. As mentioned before, the decomposition algorithms we are going to focus on are QR Factorization, Singular Value Decomposition, and Cholesky decomposition.

The problem that arises with these algorithms is that when they are run on a computer, a computer has limited memory. When there is division and multiplication throughout these algorithms, the numbers will have a long decimal values and will get rounded on a computer. This rounded value will result in a very small error; however, many small errors together can result in large errors. We want to make sure that our error overall does not get exponentially large since then our solution will be useless and can cause problems in the real world. In the first part of our project, we used the different decomposition algorithms on a symmetric matrix, and then multiplied the decomposition matrices to get the original matrix with some error. We then compared this error with machine epsilon and with each other to determine which algorithm has the smallest error.

The other important problem we are concerned with is how long an algorithm will take to complete. There may be an algorithm that is perfect in accuracy, but it will take years to run. Depending on what you are looking for, this is not really practical in the real world. Therefore, we also timed each algorithm to see how long it took and compared it with the theoretical flop count and well as with the other algorithms.

The matrix we were focusing on solving is a matrix that can be found within equation (2.20) in the paper *Gaussian Processes for Machine Learning* [2]. These equations can be complex overall, but in actuality, the main part of it is just a linear solver. This paper and the Github code provided was originally using Cholesky Factorization, and we wanted to see if there were any other decomposition methods that would work better and faster for this equation.

Mathematical Content

This research project focuses on the stability analysis on both matrix decomposition and solving linear systems. By seeing how each algorithm performs when the size

of the matrix changes, we can further delve into our investigation. We will also be searching for interesting things that may result from our analysis, such as error offsetting between decomposition and the linear solver. Our focus will be on the Gaussian Covariance Matrix (The matrix A throughout this paper) of the form ($K'(X,X) = \text{cov}(\vec{y}) = K(X,X) + \sigma^2 I$).[3] It has the following characteristics:

- Positive definite
- Symmetric
- All entries are in the interval (0,1],
  i.e. The Gaussian radial basis function: $k'_{pq} = exp\Big(-\frac{1}{2}\|\vec{x}_p - \vec{x}_q\|^2\Big)$ [4]
- Usually comes with a large condition number, i.e. ill-conditioned

We will analyze the linear system solvers in comparison to the different matrix decomposition, which we are comparing. This includes QR Factorization, Singular Value Decomposition, and Cholesky Factorization.

By comparing the error of each matrix decomposition and linear solving system to $O(\varepsilon_{machine})$, we get a sense of how good or bad the stability actually is. $O(\varepsilon_{machine})$ is precise in mathematics, which asserts that there exists some positive constant C such that, for all t sufficiently close to an understood limit. The limit in any particular machine arithmetic, the number $\varepsilon_{machine}$ is a fixed quantity. $O(\varepsilon_{machine})$ is the idealization of error an algorithm could possibly make.

The goal is to find the best algorithm that most accurately solves:

$$A\vec{x} = \vec{b} \tag{1}$$

We need to decompose A. However, we only have the true matrix A before we input it into a computer. During the input, rounding error is inevitable such that A will become $\tilde{A}$. So, we have this input $\tilde{A}$ that is being applied to $\vec{x}$ to get us the value of the $\vec{b}$. For convenience purpose, we will treat this as a known fact so **we will use A to denote $\tilde{A}$ throughout the project.**

1. QR

Full QR decomposition is applicable to any $m \times n$ matrix A. In our case, we only consider a symmetric matrix. The decomposition consists of $A = QR$, where Q is a $m \times m$ unitary matrix, and R is a $m \times m$ upper triangular matrix. In general, it is not unique, but when A is of full rank, then there exists a single R that has all positive diagonal elements. The QR decomposition provides an alternative way of solving the system of equations $A\vec{x} = \vec{b}$, without inverting the matrix A. Q is unitary, which means that $Q^T Q = I$. This in turn allows so that $A\vec{x} = \vec{b}$ is equivalent to $R\vec{x} = Q^T\vec{b}$, which thus is easier to solve using backward substitution since R is a upper triangular matrix.

To solve a $A\vec{x} = \vec{b}$ system using QR decomposition, you factorize A into QR, i.e. $QR = A$. Then construct $\vec{w} = Q^T\vec{b}$, and solve the triangular system $R\vec{x} = \vec{w}$ by back substitution. The decomposition algorithm is backward stable using Householder triangularization in theory[1], given the relative forward error:

$$\frac{\|\tilde{Q} - Q\|}{\|Q\|} = O(\varepsilon_{machine}) \tag{2}$$

$$\frac{\|\tilde{R} - R\|}{\|R\|} = O(\varepsilon_{machine}) \tag{3}$$

the QR factorization of A leads to the relative backward error:

$$\tilde{Q}\tilde{R} = A + \delta A \tag{4}$$

$$\frac{\|\delta A\|}{\|A\|} = O(\varepsilon_{machine}) \tag{5}$$

Once the decomposition is done, we have $\tilde{Q}\tilde{R}\vec{x} = \vec{b}$. Since $\tilde{Q}$ is unitary, the system could be written as $\tilde{R}\vec{x} = \tilde{Q}^*\vec{b}$ such that the linear system becomes upper triangular which could be solved by back substitution.

$A = \tilde{Q}\tilde{R}$, where $\tilde{Q}$ is orthogonal and $\tilde{R}$ is upper triangular. Here A is the input, the x in the definition, and $\tilde{Q}$ and $\tilde{R}$ are the outputs of the function. In this case, since the algorithm is for back substitution, the forward and backward error are:
Relative forward error:

$$\frac{\|\tilde{\vec{x}} - \vec{x}\|}{\|\vec{x}\|} = O(\kappa(A)\varepsilon_{machine}) \tag{6}$$

where $\kappa(A)$ is the condition number of A. The backward error is the norm between the actual system A that we are solving and another slightly different system that is corresponding to the solution .
Relative backward error:

$$\frac{\|\tilde{A} - A\|}{\|A\|} \tag{7}$$

Since the $\tilde{A}$ is not observable and not unique, we will not cover the backward error of any linear solvers in this project.

2. SVD

Singular Value decomposition is applicable to all $m \times n$ matrix, A. The decomposition is of $A = U\Sigma V^*$, Σ is a non-negative diagonal matrix that is composed of the singular values of the matrix. U and V are unitary matrices, and V^* is the conjugate transpose of V.

Once we have the SVD, $A = U\Sigma V^*$, we can easily solve $A\vec{x} = \vec{b}$. We end up getting $(U\Sigma V^*)\vec{x} = \vec{b}$. Since U is unitary, its inverse equals to its transpose, so we can easily calculate $\Sigma(V^*\vec{x}) = U^*\vec{b}$. This is a system associated with a diagonal matrix Σ which is trivial to solve.

For the SVD decomposition, we are worried about both the forward and backward error. Relative forward error:

$$\frac{\|\tilde{U} - U\|}{\|U\|} = O(\varepsilon_{machine}) \tag{8}$$

$$\frac{\|\tilde{\Sigma} - \Sigma\|}{\|\Sigma\|} = O(\varepsilon_{machine}) \tag{9}$$

$$\frac{\|\tilde{V} - V\|}{\|V\|} = O(\varepsilon_{machine}) \tag{10}$$

As mentioned before, we cannot explicitly calculate the forward error of the decomposition since the real U, Σ and V are unknown.

But we could investigate the relative backward error:

$$\frac{\|\tilde{U}\tilde{\Sigma}\tilde{V}^* - \tilde{A}\|}{\|\tilde{A}\|} = O(\varepsilon_{machine}) \tag{11}$$

As mentioned before, the backward error of the linear solver is not available but we could work on the forward error:

$$\frac{\|\tilde{\vec{x}} - \vec{x}\|}{\|\vec{x}\|} = O(\kappa(A)\varepsilon_{machine}) \tag{12}$$

where $\kappa(A)$ is the condition number of A.

3. Cholesky

Cholesky decomposition is applicable to a square, Hermitian, positive definite matrix A. The decomposition consists of splitting A into $A = LL^*$, where L is a lower triangular matrix. Similar to QR and SVD:
Relative forward error:

$$\frac{\|\tilde{L} - L\|}{\|L\|} = O(\varepsilon_{machine}) \tag{13}$$

Relative backward error:

$$\frac{\|\tilde{L}\tilde{L}^* - \tilde{A}\|}{\|\tilde{A}\|} = O(\varepsilon_{machine}) \tag{14}$$

After we use the decomposition algorithm to solve for $\tilde{L}$, we need to solve $\tilde{L}\tilde{L}^*\vec{x} = \vec{b}$. Thus, we need to compute $\vec{y} = \tilde{L}^*\vec{x}$ first, and then solve:

$$\tilde{L}\vec{y} = \vec{b} \tag{15}$$

Similar to QR and SVD, the backward error of the linear solver is not available but forward error is.
Relative forward error:

$$\frac{\|\tilde{\vec{x}} - \vec{x}\|}{\|\vec{x}\|} = O(\kappa(A)\varepsilon_{machine}) \tag{16}$$

where $\kappa(A)$ is the condition number of A.

4. Condition Number

It is also important to mention the what is the condition number of A to understanding the forward error of the linear solvers. It can be defined as follows:

$$\kappa(A) = \|A\|\|A^{-1}\| \tag{17}$$

where the norm is not restricted here. In other words, if we use the 2-norm, it becomes:

$$\kappa(A) = \frac{\rho_1}{\rho_m} \tag{18}$$

where ρ_1 and ρ_m are the largest and smallest singular values of A. Geometrically, this number $\kappa(A)$ represents the eccentricity of the hyper-ellipse when one mapping a unit sphere onto the column space of A. If $\kappa(A)$ is large, we say the matrix A is ill-conditioned and then the linear system $A\vec{x} = \vec{b}$ cannot be solved accurately under the linear system solvers we covered in this project because the forward error is of order $\kappa(A)\varepsilon_{machine}$.

Computational Experiments and Outcomes

For the purpose of testing, we generated some random data for our experiments. We first created a column vector with size n. Then we prepared a kernel matrix using Gaussian radial basis function with that column vector as an input. Some small noise was added to the diagonal of the kernel matrix to imitate the real-life noisy data. As we needed an exact solution $\vec{x}$ to compute the forward error of the linear solver later, an n-vector of $\vec{1}$ was generated. To solve a linear system $A\vec{x} = \vec{b}$, $\vec{b}$ was computed using $A \cdot \vec{1}$. After setting up those criteria and defining functions (see Appendix) to compute backward error, forward error, backward substitution, 3 solvers and runtime, we experimented with matrices in various sizes. We set the range for matrix size from 100 to 1800 with an interval of 50. During each iteration, decomposition time, linear solver time, decomposition backward error, linear solver forward error and condition number of the matrix were recorded to compare. For the runtime, we ran the experiment for each type of decomposition and linear solving twice and averaged the results. Our experimental outcomes are illustrated below.

1. Comparing relative errors (actual vs theoretical)

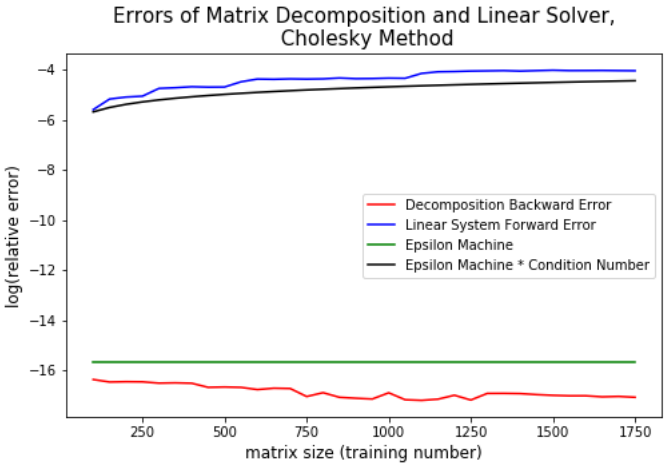

Figure 1: The backward error of the Cholesky decomposition is **well under** machine epsilon; although the actual forward error of the linear system solver is significant, it closely matches with the theoretical error $O(\kappa(A)\epsilon_{machine})$ where $\kappa(A)$ is the condition number of the matrix A.

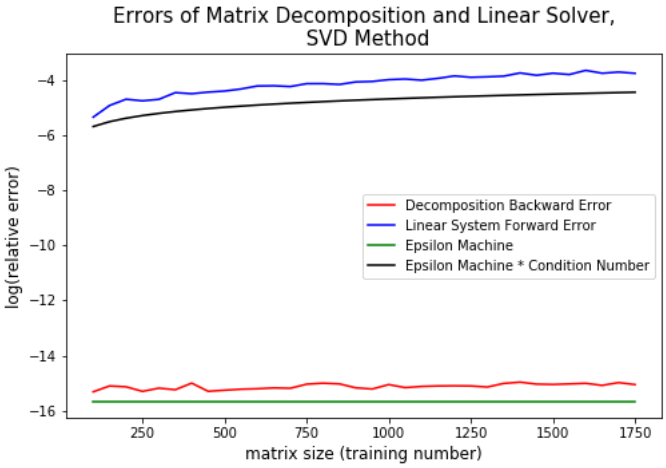

Figure 2: The backward error of the SVD decomposition is **very close** to machine epsilon; although the actual forward error of the linear system solver is significant, it closely matches with the theoretical error $O(\kappa(A)\epsilon_{machine})$ where $\kappa(A)$ is the condition number of the matrix A.

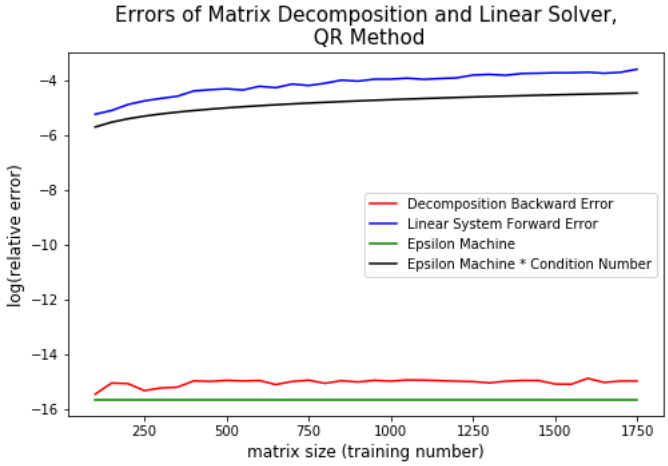

Figure 3: The backward error of the QR decomposition is **very close** to machine epsilon; although the actual forward error of the linear system solver is significant, it closely matches with the theoretical error $O(\kappa(A)\epsilon_{machine})$ where $\kappa(A)$ is the condition number of the matrix A.

2. Comparing runtime between Matrix Decomposition and Linear Solver

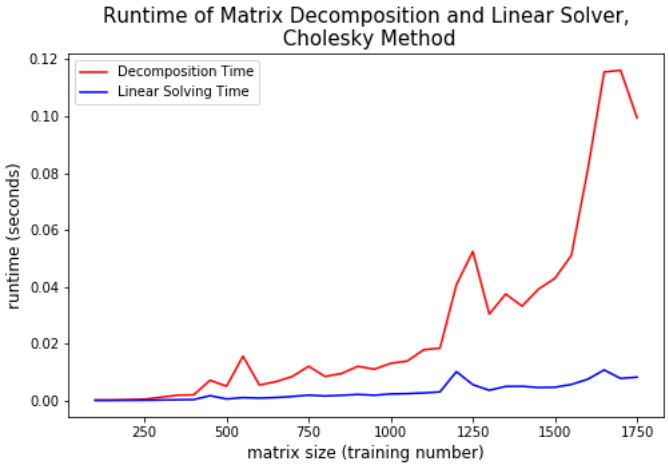

Figure 4: This result aligns with expectation because theoretically Cholesky decomposition requires $\sim \frac{1}{3}m^3$ flops or $O(m^3)$ flops while linear solver only requires $\sim 2m^2$ flops or $O(m^2)$ flops.

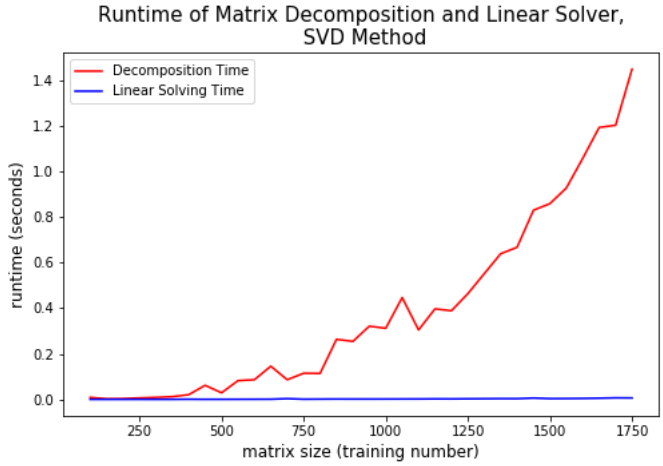

Figure 5: This result aligns with expectation because theoretically SVD decomposition requires $\sim \frac{8}{3}m^3$ flops or $O(m^3)$ flops while linear solver only requires $\sim 4m^2$ flops or $O(m^2)$ flops.

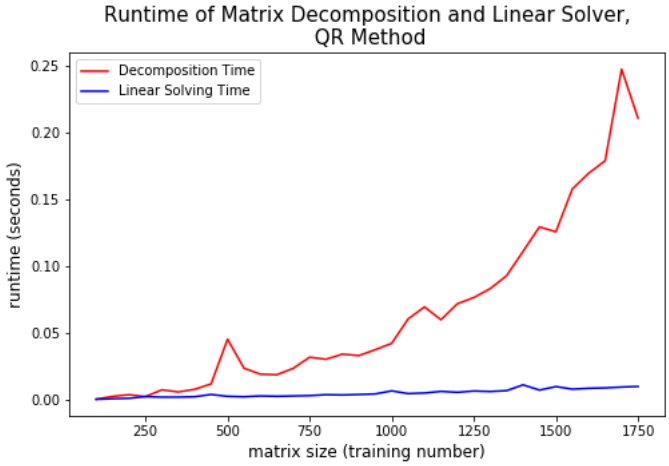

Figure 6: This result aligns with expectation because theoretically QR decomposition requires $\sim \frac{4}{3}m^3$ flops or $O(m^3)$ flops while linear solver only requires $\sim 3m^2$ flops or $O(m^2)$ flops.

3. Comparing runtime of Decomposition (Actual vs Theoretical)

In this subsection, we derived the theoretical runtime vs matrix size solely by using flop count and the speed of the computer used.

- Computer used: Macbook Pro, 2.7 GHz Intel Core i5, Dual Core
- Assuming 2 flops per cycle

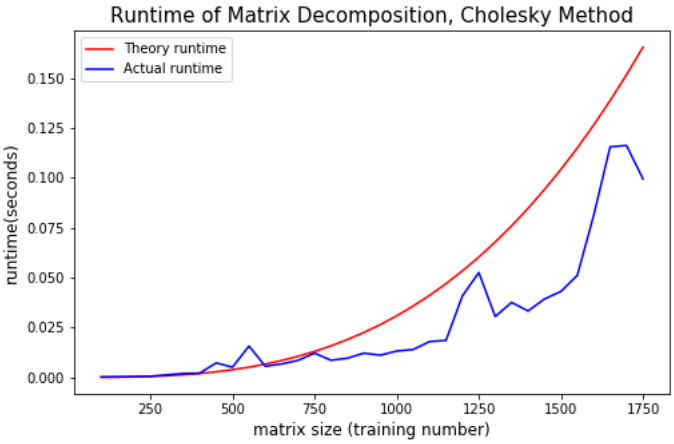

Figure 7: This result shows the actual performance is slightly better than the theoretical flop count from well written algorithm *scipy.linalg.cho_solve*, the significance grows with the matrix size.

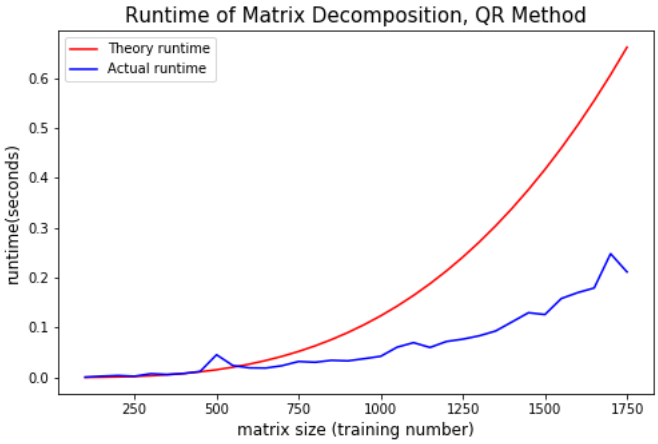

Figure 8: This result shows the actual performance is faster than the theoretical runtime, though the trend is closely aligns with the theoretical trend when the size of the matrix is increasing. The QR decomposition function from Numpy, which is called from LAPACK, might get optimized in some ways that make it run very fast on this computer.

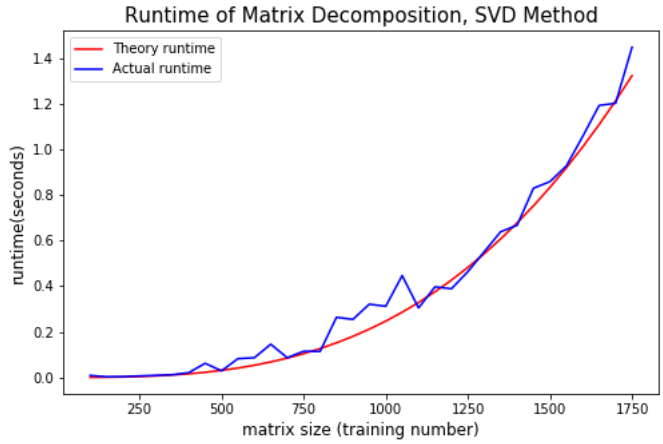

Figure 9: This result shows that the actual performance closely aligns with the theoretical performance. There is some wiggles with the blue curve due to the memory usage of the computer.

4. Comparing runtime of Linear Solver (Actual vs Theoretical)

In this section, algorithm of back substitution has been developed to solve linear systems in the shape of upper triangular.

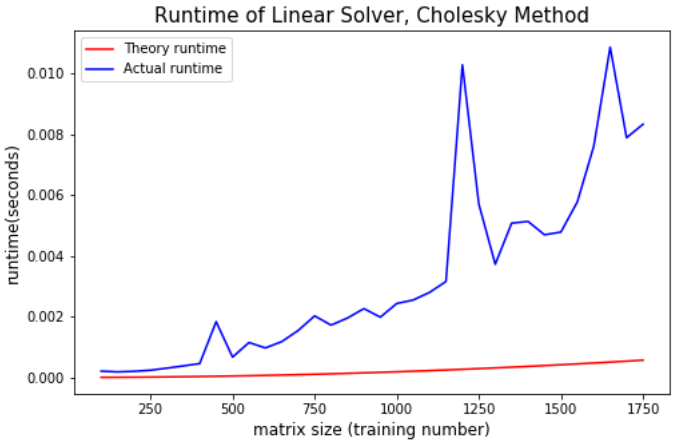

Figure 10: Theoretically the flop count of this linear solver is $O(m^2)$. While the memory causes a longer actual runtime, the trend of the actual runtime is mildly increasing and aligns with the theory.

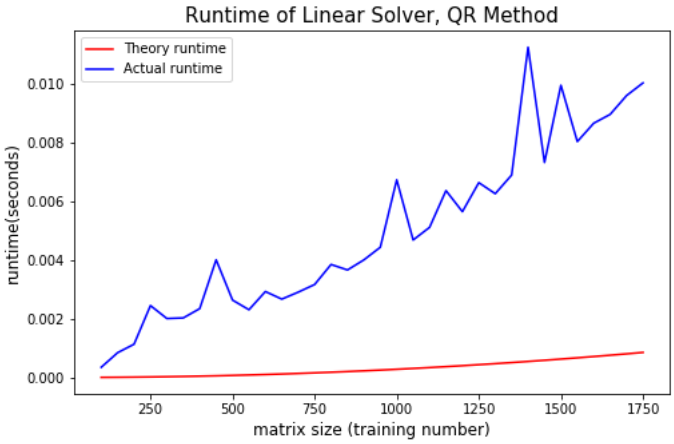

Figure 11: Theoretically the flop count of this linear solver is $O(m^2)$, the actual runtime seems to be much higher than theory due to memory consumption of the computer. The heavy-lifting part of the algorithm's speed might be from our simple application of the function $back\_substitute(A,b)$ that is not fully optimized for memory usage (please see Appendix for details).

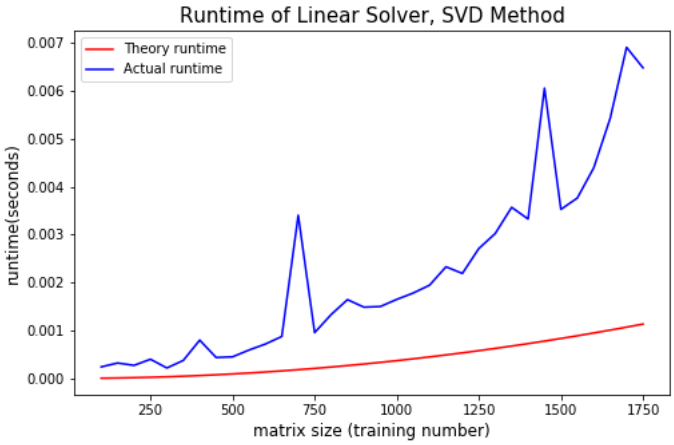

Figure 12: Theoretically the flop count of this linear solver is $O(m^2)$. While the memory causes a longer runtime, on average, the trend of the actual runtime is slightly increasing and aligns with the theory.

Assimilation and Next Steps

If we were to do this project again, there are many different things that we would change. First, we would study more new decomposition algorithms to see how some of the less popular algorithms compare. We would also try different kinds of matri-

ces. For example, in this project we focused on symmetric, positive-definite matrices because Cholesky only works for this special kind of matrix and we wanted to keep it consistent for the other algorithms. We could instead look at tri-diagonal matrices or non-special matrices to see if the decomposition algorithms' accuracy or speed changes (excluding Cholesky).

Lastly, if we had a lot more time, we would see if there were a way to increase the speed of any algorithm and set up a computing environment to prioritize running the experiments to avoid background noise when comparing runtime among different algorithms. Improving the speed of the algorithm does not mean increasing the speed by its $O(-)$ order, but maybe by some constant factor speed increase. This, however, is widely studied and would be very hard to do. As far as advice for any team that plans to do something similar, be sure to check the results of the computer's forward and backward error and the time with the theoretical values to make sure that there are no errors in the coded algorithms. They should also prepare separate environments to test the algorithms under different cases of memory usage in the computer to achieve more robust results.

Conclusion

From our experiments with the decomposition methods, we found that all three methods had a very significant forward error for linear system solver that depends on the condition number of the matrix, but they all have decomposition backward errors close to machine epsilon. Cholesky factorization however, was the only algorithm such that the decomposition backward error was less than machine epsilon, while the other two had a decomposition backward error slightly greater than machine epsilon. From this, we can conclude that if we only cared about accuracy of our solution, we would choose Cholesky factorization over SVD and QR.

We also care about the performance of each of the algorithms, and while each algorithm has a $O(m^3)$ decomposition and $O(m^2)$ linear solver, they still have slight differences in run time on the computer. It can be seen that both Cholesky and QR have decomposition performances slightly faster than the theoretical performance, but the SVD method actually performs closely to its theoretical expected performance. This means that if we care about time, we may not want to use the SVD decomposition

method because it may not run as fast as we expect it to.

Lastly, the linear solvers all have a performance worse than their theoretical expected performance, but this is because of a memory constraint. Cholesky's linear solver seems to have similar runtime pattern to that of SVD solver which was computed manually by matrix multiplication and backward substitution. However, on average, Cholesky's method performs slightly better than do other methods. Overall, we can conclude that Cholesky factorization and solver are the best algorithms to use for both accuracy and speed on symmetric, positive-definite matrices.

Appendix

The main functions that we used for the experiments are below. For more details on the testing, please visit **Kernel Linear Solver Repository on GitHub**

```
## Function to calculate forward error of linear system solver
def cal_linsystem_forward_error(sol, x, p=2):
    return np.linalg.norm(sol-x,p)/np.linalg.norm(x,p)

## Function to calculate back-substitution for diagonal matrices
def back_substitute_diag(A, b):
    n = b.size
    b_list = b.ravel()
    x = b_list/A

    return x.reshape(n,1)

def back_substitute(A, b):
    n = b.size
    x = np.zeros(b.shape)

    x[n-1] = b[n-1]/A[n-1,n-1]
    for i in range(n-2,-1,-1):
        x[i] = b[i]
        u = A[i,i+1:n]
        v = x[i+1:n]
        dotproduct = np.dot(u,v)
        x[i] -= dotproduct
        x[i] = x[i]/A[i,i]

    return x

## Function to compute linear system solution using Cholesky
def cholesky_solver(K,x, y, p=2):
    #Matrix Decomposition
    tic = time.time()
```

```
    L = sp.linalg.cholesky(K, lower = True)
    toc = time.time()
    decomp_time = toc-tic
    #Linear solver
    tic1 = time.time()
    lower = True
    sol = sp.linalg.cho_solve((L, lower), y)
    toc1 = time.time()
    lin_solve_time = toc1-tic1
    #Compute error
    decomp_error = np.linalg.norm(L@L.T-K,p)/np.linalg.norm(K,p)
    lin_forward_error = np.linalg.norm(sol-x,p)/np.linalg.norm(x,p)

    return decomp_time, lin_solve_time, decomp_error, lin_forward_error

## Function to compute linear system solution using SVD
def svd_solver(K,x,y, p=2):
    #Matrix Decomposition
    tic = time.time()
    U,S,Vh = np.linalg.svd(K,full_matrices=False)
    toc = time.time()
    decomp_time = toc-tic
    decomp_error = np.linalg.norm(U@np.diag(S)@Vh-K,p)/
                                        np.linalg.norm(K,p)

    #Linear solver
    tic1 = time.time()
    c = U.T@y
    w = back_substitute_diag(S,c)
    sol = Vh.T@w
    toc1 = time.time()
    lin_solve_time = toc1-tic1
    lin_forward_error = np.linalg.norm(sol-x,p)/np.linalg.norm(x,p)

    return decomp_time, lin_solve_time, decomp_error, lin_forward_error
```

```
## Function to compute linear system solution using QR
def qr_solver(K, x, y,p=2):
    #Matrix Decomposition
    tic = time.time()
    Q, R = np.linalg.qr(K)
    toc = time.time()
    decomp_time = toc-tic
    decomp_error = np.linalg.norm(Q@R - K, p)/np.linalg.norm(K,p)

    #Linear solver
    tic1 = time.time()
    p = Q.T@y
    sol = back_substitute(R, p)
    toc1 = time.time()
    lin_solve_time = toc1-tic1

    return decomp_time, lin_solve_time, decomp_error, sol



## Function to experiment kernel matrix decomposition
## and linear system solver
def kernel_linear_testing(training_number, decomp_type, p = 2):
    #Generate K_y matrix
    x_train = np.array(np.linspace(0.01,1.0, training_number),
    dtype='float32').reshape(training_number, 1)
    rbf= gaussian_process.kernels.RBF()
    Kernel= rbf(x_train, x_train)
    K_y = Kernel + np.eye(training_number) * 1e-8
    cond_K_y = np.linalg.cond(K_y, p)

    #Generate x vector (vector of 1)
    x = np.ones((training_number,1), dtype = 'int')
```

```
    #Generate y vector
    y = K_y@x

    #Set up experiment for matrix decomposition and
    # linear system solver
    if decomp_type == 'cho':
        decomp_time, lin_solve_time, decomp_error, lin_forward_error =
        cholesky_solver(K_y,x, y, p)
    elif decomp_type == 'svd':
        decomp_time, lin_solve_time, decomp_error, lin_forward_error =
        svd_solver(K_y,x, y, p)
    elif decomp_type == 'qr':
        decomp_time, lin_solve_time, decomp_error, sol_x =
        qr_solver(K_y,x, y, p)
        lin_forward_error = cal_linsystem_forward_error(sol_x, x, p)
    else:
        print ('Please select a decomposition method (svd, qr, cho)!')
        return 0

    return decomp_time, lin_solve_time,
            decomp_error, lin_forward_error, cond_K_y
```

Bibliography

[1] Lloyd N. Trefethen, David Bau. *Numerical Linear Algebra*, Page 116 - 117. SIAM, ISBN 0-89871-361-7, 1997.

[2] Lloyd N. Trefethen, David Bau. *Numerical Linear Algebra*, Page 29. SIAM, ISBN 0-89871-361-7, 1997.

[3] C. E. Rasmussen & C. K. I. Williams. *Gaussian Processes for Machine Learning*, Equation (2.20). MIT Press, ISBN 026218253X, 2006.

[4] C. E. Rasmussen & C. K. I. Williams. *Gaussian Processes for Machine Learning*, Equation (2.16). MIT Press, ISBN 026218253X, 2006. Retrieved from http://www.gaussianprocess.org/gpml/chapters/RW.pdf