**UNIVERSITY OF VICTORIA**

Department of Software Engineering
**SENG 440 – Embedded Systems**

# Matrix Diagonalization

August 10th, 2020

Michail Roesli - V00853253 - mroesli@uvic.ca
Robert Tulip - V00846133 - rtulip@uvic.ca

# Table of Contents

# 1 List of Figures

# 2 Introduction

In the following section we'll discuss the project requirements for designing a Matrix Diagonalization program, the contributions that will be made, and the general report organization.

## 2.1 Project Requirements

In this project we explore the design process of a matrix diagonalization program in C. To achieve matrix diagonalization, we will be using the Singular Value Decomposition (SVD) method. In particular, we'll explore Carl Gustav Jacob Jacobi's proposed method to calculate the SVD called the Jacobi method to optimize the calculation of the diagonal. To calculate the matrix diagonal we will assume a symmetric 4x4 matrix input with 14-bit word length values. Optimizations will be done while complying with the 32-bit ARM architecture

## 2.2 Contributions

In this report we will be exploring several possible optimizations and what we decided on doing. In particular we will explore:

- Why we chose to use a lookup table instead of a piecewise linear approximation for calculating the trigonometric functions
- How we implemented fixed point arithmetic, how we scaled our values, and the methods we defined to calculate the division and multiplication
- Matrix value manipulation and how we copied values across the iterations of the Jacobi method
- How we parallelized multiplication of our matrix through using NEON SIMD instructions
- Optimizations using a new defined instruction in assembly

For each of the tasks we examined the assembly inlining for cycle optimizations.

## 2.3 Organization

In the following sections, we'll explore the design process of developing and optimizing our matrix diagonalization program. In the background section, we'll cover the theory and technical knowledge that will be indispensable to a technical audience. Next in the design section, the engineering design decisions will be presented and analyzed with explanations. The discussion section then presents how these design decisions affected our final solution and what the cost and major limitations we encountered. Lastly, future work will be listed alongside the conclusions of our results.

# 3 Background

In this section, we'll cover the technical knowledge that will be necessary to understand matrix diagonalization.

## 3.1 Problem

Matrix diagonalization is used in applications including image processing and compression, molecular dynamics, small-angle scattering, and information retrieval [1]. It is used because diagonalized matrices have various desirable properties. One problem in obtaining a diagonalized matrix is that it "poses numerical stability issues" [2] and is "difficult to calculate it fast" [2]. Singular Value Decomposition (SVD) is one method in achieving diagonalization which helps solve this issue since it is "always numerically stable for any matrix [, however,] it is typically more expensive than other decompositions" [3]. This is due to several factors including the number of steps required at each iteration, the complexity of the floating point calculations and rotations, and the time it takes to converge. To improve the calculation speed, we can make use of fixed-point arithmetic, use special techniques in calculating the trigonometric results, and parallelize tasks using programming libraries. The chosen SVD method that will be examined is Jacobi's method.

## 3.2 Jacobi Method

In SVD, an n x n real or complex valued matrix M is used as input which is then multiplied with unitary orthogonal matrices on both sides so that all the diagonal elements become zero. The original matrix M is thus represented as seen in equation 1 below.

$$M = U\Sigma V^T \tag{1}$$

The Jacobi method is a method proposed by Carl Gustav Jacob Jacobi which moves the off-diagonal energy to the main diagonal by applying several iterations of plane rotations to M. This transforms M into Σ which is the diagonal matrix of singular values. In each iteration the matrix is partitioned into blocks of 2x2 matrices which results in n/2 x n/2 blocks. These 2x2 then have their off diagonal values set to zero using a two-sided rotation which is described by equation 2 below.

$$R(\theta_l)\begin{pmatrix} m_{ii} & m_{ij} \\ m_{ji} & m_{jj} \end{pmatrix} R(\theta_r)^T = \begin{pmatrix} \psi_1 & 0 \\ 0 & \psi_2 \end{pmatrix} \tag{2}$$

where $\theta_l$ and $\theta_r$ are the left and right rotation angles respectively.

At each iteration, the Jacobi method selects pairs in a cyclic order. For our implementation we chose to use cyclic by rows ordering which has the ordering 1-2, 1-3, … , 1-n, 2-3, 2-4, … , 2-n, 3-4, … , (n-1)-n.

With each of these iterations, the rotations of these 2x2 matrices cause the off-diagonal energy magnitudes to decrease while the magnitude of the diagonal elements increases. This requires the evaluation of arctan, cosine and sine to perform the rotations. To calculate the rotations, we use the rotation matrices $R(\theta_l)$ and $R(\theta_r)$ which are calculated as shown below in equation 3.

$$R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \tag{3}$$

Where $\theta$ is $\theta_l$ or $\theta_r$ and are calculated as shown in equation 4 and equation 5 respectively below.

$$\theta_{SUM} = \theta_r + \theta_l = \arctan \begin{pmatrix} m_{ji} + m_{ij} \\ m_{jj} - m_{ii} \end{pmatrix} \tag{4}$$

$$\theta_{SUM} = \theta_r + \theta_l = \arctan \begin{pmatrix} m_{ji} + m_{ij} \\ m_{jj} - m_{ii} \end{pmatrix} \tag{5}$$

A few notes on the convergence of towards the diagonalized matrix are that the off-diagonal elements may not completely vanish and that if the matrix input is symmetric, it results in finding the eigenvalues of the matrix. [2].

## 3.3 Fixed Point Arithmetic

In our project we make use of fixed point arithmetic to reduce the cost of calculations; floating-point unit calculations are expensive. To use fixed point arithmetic the numbers must be represented in fixed point notation which is another way of saying that it only stores a fixed number of values which come before a decimal point. To represent floating point values in fixed point notation, a scaling factor is required. These are usually selected by the user and remembered by the computer which don't need to have allocated hardware or software resources to be memorized [4].

When performing fixed-point arithmetic the following rules are important

- Addition and subtraction increases the word length by 1
- Multiplication increases word length by 2 times (2x-1 if one value is unsigned)
- Division increases / decreases wordlength by the difference in bits between numerator and denominator

# 4 Design

In this section we'll discuss our engineering design decisions made, in particular we'll explore the trigonometric functions, fixed point arithmetic, matrix referencing, and the software and hardware instructions that could be designed.

## 4.1 Trigonometric Functions

Trigonometric functions such as arctangent, sine and cosine are expensive, and are repeated on every iteration of the Jacobi method. Arctangent, sine, and cosine are all transcendental functions [2] so they cannot satisfy a polynomial equation. To optimize trigonometric functions we examined two possible solutions including a lookup table and a piecewise linear approximator.

In piecewise linear approximation several slices of the domain are used to approximate a function, while in a lookup table the values are generated ahead of time and accessed using the index as the input value. When comparing both solutions we decided to use a lookup table because it produced more accurate results while having about the same cost in terms of clock cycles. By using a lookup table we were able to specify the range we'd like to use and the accuracy based on the number of samples we generate between the range.

When implementing our lookup table for arctan we selected the range 0 to 10 for our input and generated 10000 fixed-point values, so that they would not need to be converted when retrieving them. If the input was negative, the lookup function would take the absolute value of the input and return the fixed point value multiplied by sign and if it was outside of the bounds we would approximate the value to π/2. Although this introduces some loss in accuracy if our values are ever outside those bounds, we are able to regenerate our table using a larger range and precision and take a larger toll on our memory for the benefit of having more accurate values. For our sine and cosine lookup tables, we chose the input domain to be 0 to π because we are able to represent all the results for sine below 0 by multiplying our results by the sign of our input, and ignoring the sign for cosine. Furthermore we do not need to take the modulo of our input because we found that we were guaranteed to have a value between 0 and π as shown in figure 1 below.

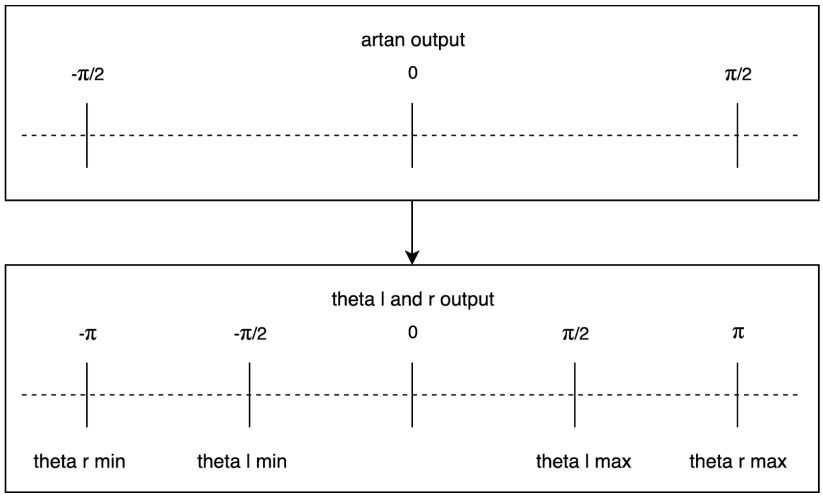


Figure 1. possible input ranges for sin and cos

## 4.2 Fixed Point Arithmetic

For the purposes of this project, we were instructed to convert an input matrix, M, of floating point numbers into 14 bit fixed point. We were also instructed to use 14 bit integers for all of our fixed point arithmetic, for sake of simplicity. Additionally, we chose our valid input range, for M, to be in the range $[-2^7, 2^7]$. These values were chosen, mostly out of convenience, because it covers all the values used in the example from the slides; this way the slides provided served as a tool for expected output.

$$\begin{pmatrix} 33 & 77 & -11 & 26 \\ -42 & 14 & 79 & -53 \\ -68 & -10 & 45 & 90 \\ 34 & 16 & 38 & -19 \end{pmatrix}$$

Figure 2. Input matrix from slides used for testing

Given our input range for M, our scale factor for converting elements of matrix M to and from fixed point is $2^6$.

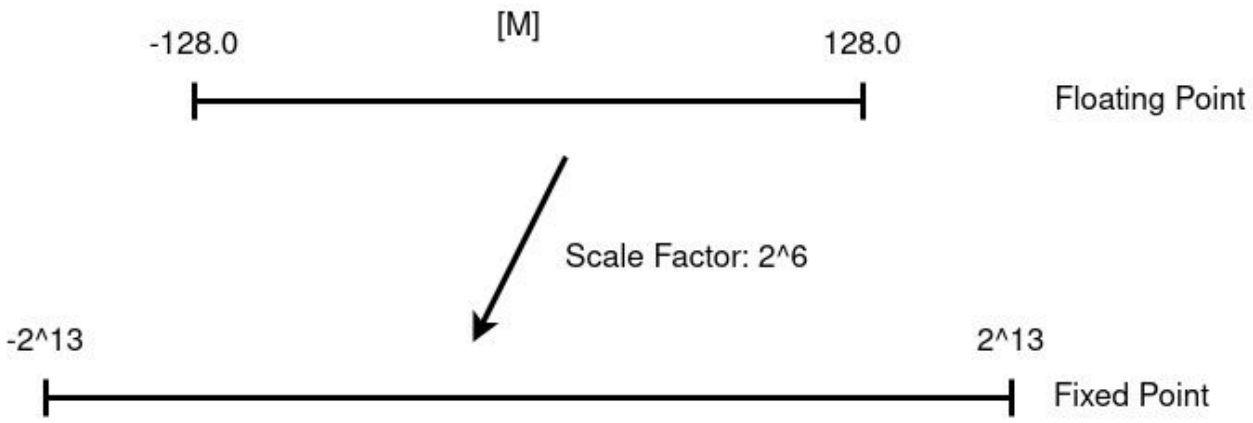


Figure 3. Calculating the Scale Factor for M with 14 bit integer

Matrices U and V are stable and their elements range only from -1.0 to 1.0 in floating point. Again, we chose to use a 14 bit signed integer for representing these values. As seen below, the scale factor for U and V is $2^{13}$.

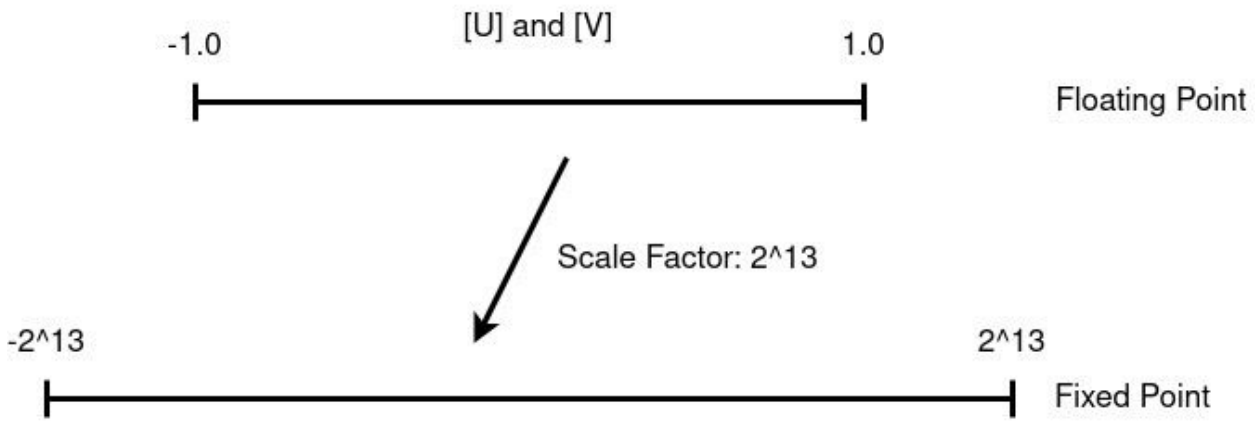


Figure 4. Calculating the Scale Factor for U and V with 14 bit integers

The Jacobi Method requires calculating an arctangent, sine and cosine. An arctangent returns a value between $-\pi/2$ and $\pi/2$. For the sake of keeping the same scale factor, our arctangent values are 15bit values. This is because the underlying C type is an int16_t, which requires no additional work.

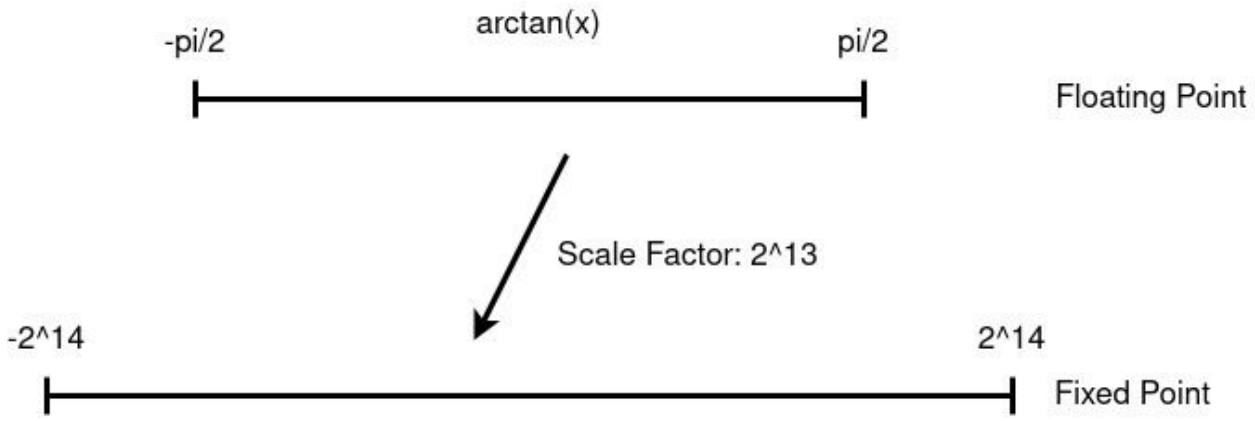


Figure 5. Calculating the Scale Factor for arctan(x) with 15 bit integer

Sine and cosine both range between -1.0 and 1.0. Again a scale factor of $2^{13}$ can be used to convert the result into a 14 bit integer.

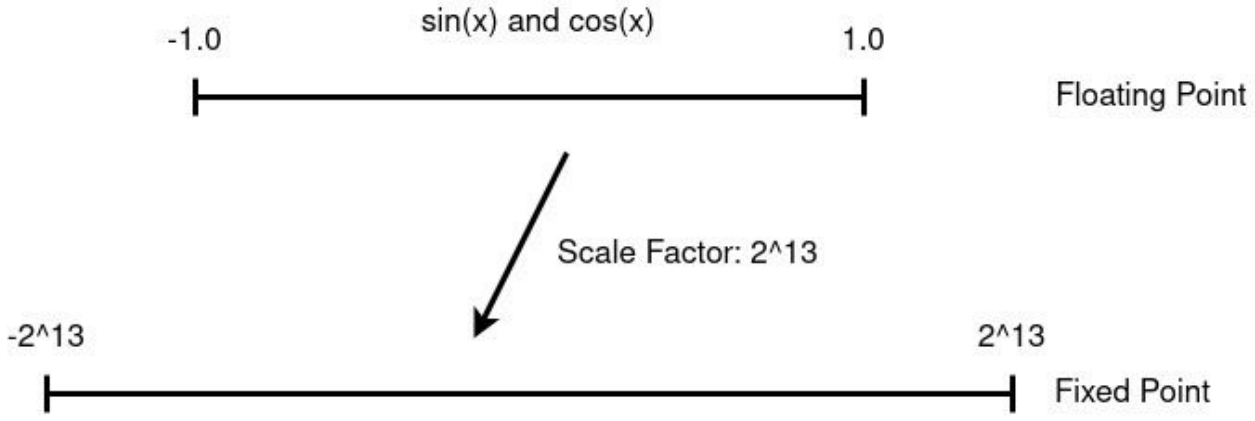


Figure 6. Calculating the Scale Factor for sin(x) and cos(x) with 14 bit integer

As mentioned above, the Jacobi Method of matrix diagonalization involves many matrix multiplications, which, in turn, will require a fixed point multiplication. Because we're using 14 bit fixed point integers, the result of a multiplication is 27 bit. Since the result from a matrix multiplication is frequently used as the input for the next iteration of i and j in the Jacobi method, we chose to truncate the resulting 27 bit integer back to 14 bits. This keeps the scale factor consistent from iteration to iteration, and sweep to sweep.

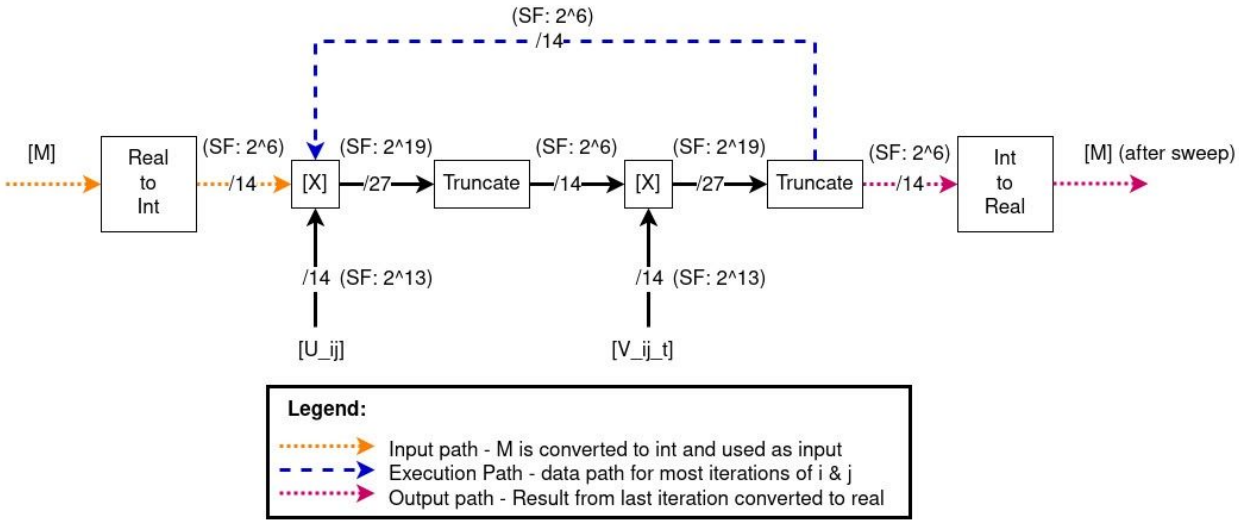


Figure 7. Summarized Jacobi Sweep process.

Unfortunately, there are a number of fixed point divisions which are required in our implementation of the Jacobi method. In total, for every iteration of i and j, 8 divisions are required for calculating trigonometric functions.

## 4.3 Reducing Copies

Our initial implementation would copy values from M`, U`, and V` back into M, U, and V at the end of every iteration of i and j. We were able to avoid performing all of these copies by using two matrices for M`, U`, and V`. Every iteration, the matrix which was used as output in the previous iteration is used for input, and vice versa.

## 4.4 Single Instruction Multiple Data (SIMD)

Two operations were evaluated when designing our project. One of them being single Instruction Multiple Data (SIMD) instructions. For this we considered the use of Neon intrinsics [5-8] to optimize our matrix multiplications. Our original matrix multiplication algorithm is as seen in figure 8 below.

```
matrix_multiply(matrix LHS, matrix RHS, matrix RESULT){
    for (int i = 0; i < MATRIX_SIZE; i++){
        for (int j = 0; j < MATRIX_SIZE; j++){
            RESULT[i][j] = 0;
            for (int k = 0; k < MATRIX_SIZE; k++){
                RESULT[i][j] += LHS[i][k] * RHS[k][j];
            }
        }
    }
}
```

Figure 8. Matrix multiply code without parallelization

And the updated SIMD matrix multiplication algorithm is as seen in figure 9 below [9].

```
matrix_multiply(matrix LHS, matrix RHS, matrix RESULT){
    int32x4_t RHS_row_0 = load_int32x4(RHS[0][0]);
    int32x4_t RHS_row_1 = load_int32x4(RHS[1][0]);
    int32x4_t RHS_row_2 = load_int32x4(RHS[2][0]);
    int32x4_t RHS_row_3 = load_int32x4(RHS[3][0]);
    int32x4_t RESULT_row;

    for (int i = 0; i < MATRIX_SIZE; i++){
        RESULT_row = multiply_int32x4_by_scalar(RHS_row_0, LHS[i][0]);
        RESULT_row += multiply_int32x4_by_scalar(RHS_row_1, LHS[i][1]);
        RESULT_row += multiply_int32x4_by_scalar(RHS_row_2, LHS[i][2]);
        RESULT_row += multiply_int32x4_by_scalar(RHS_row_3, LHS[i][3]);
        RESULT[i][0] = store_int32x4(RESULT_row);
    }
}
```

Figure 9. Matrix multiply code with Neon Intrinsics parallelization

Although we were able to speed up the matrix multiplication using SIMD operations, we are not able to increase the calculation speed of trigonometric functions using SIMD operations [2]. This is where Application-Specific Instruction Set Processors (ASIP) operations may prove helpful.

## 4.5 Application-Specific Instruction Set Processor (ASIP)

We also considered the performance improvements of a theoretical application specific instruction set process (ASIP), which would help optimize the required trigonometric functions. Our hypothetical ASIP

instruction would do the following: take in four 14 bit integers, m_ij, m_ji, m_jj, m_ii, as input, and produces cos(theta_l), sin(theta_l), cos(theta_r), and sin(theta_r) as output.

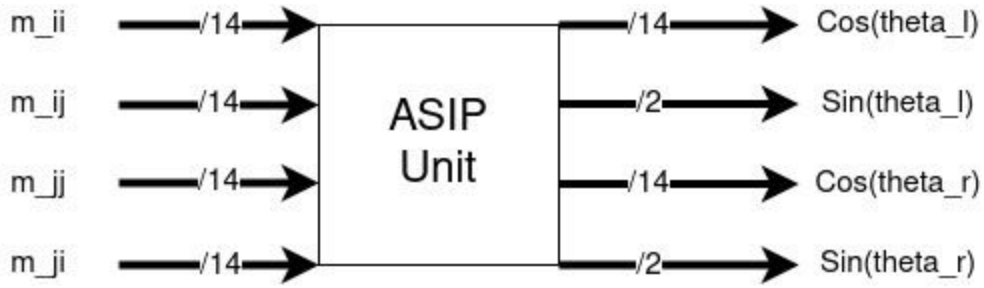


Figure 10. ASIP unit description

Because ARM assembly instructions only accept two 32 bit inputs, we have to pack m_ij and m_ji, and m_ii and m_jj together into two 32 bit integers. Additionally, ARM assembly only produces one 32 bit output. As such, we have to pack cos(theta_l), sin(theta_l), cos(theta_r), and sin(theta_r) together. Since most of the energy is in the diagonal elements, we can assign 14 bits to each of cos(theta_l) and cos(theta_r) and 2 bits to each of sin(theta_l) and sin(theta_r).

# 5 Discussions

In this section, our initial implementation is discussed, alongside the cost, resulting limitations and some analysis on the development of the final implementation.

Our initial implementation didn't try to make any optimizations. Floating point arithmetic was used throughout the algorithm. The arctangent, sine, and cosine functions from the math.h header were used. The algorithm had many redundant copies between iterations. Additionally, a naive matrix multiplication algorithm was used. The pseudocode of our initial implementation is seen in figure 10 below.

There were many areas where we were able to improve upon this initial implementation. As mentioned in section 3, Design, we were able to use fixed point arithmetic rather than floating point, lookup tables for trigonometric functions, minimize the number of copies, and perform an optimized matrix multiplication. The following sections will discuss the implementations and performance benefits of these improvements.

```
jacobi_sweep(matrix M, matrix U, matrix V){
   for (int i = 0; i < MATRIX_SIZE - 1; i++){
       for (int j = i; j < MATRIX_SIZE; j++){
           // Calculate sin(theta_1), cos(theta_1).
           // Init U_ij, U_ij_trans, and V_ij_trans
           // U * U_ij_trans -> U`
           // U_ij * M -> M`tmp
           // M`tmp * V_ij_trans -> M`
           // V_ij_trans * V_trans -> V_trans`
           // Copy M`, U`, and V_trans` into M, U, and V_trans
       }
   }
}
```

Figure 11. Jacobi method base implementation pseudocode

## 5.1 Trigonometric Functions

Upon implementing the lookup table our end result became less accurate by around 1 decimal point. This was expected for the following reasons:

- The input we receive may not be exact as is due to lack of precision in previous iterations
- Getting results which are outside of the bounds we selected for lookup table
- Some information is discarded when shifting our fixed point value to be provided as input to our lookup table
- Results from the lookup table may not correspond to the input, and do not round to the correct value as this would increase the time complexity

When using a lookup table we found that there was a loss of accuracy when our results were outside our specified range for arctangent and were approximated to be +-π/2. One way we could rectify this is to provide a larger lookup table range that we know the values will not exceed and increase the samples generated in between to keep our accuracy consistent over the range. This would remove the large result uncertainty range if it looks up a value outside of our lookup table range. One possible way to increase the accuracy of the arctangent lookup table, in particular, would be to have a non-linear division of the range. Using a logarithmic scale would allow for greater accuracy where there is the most change in the arctangent function. This optimization may be less performant, because more complex operations would be required to map a fixed point value to the index.

In performing a lookup into one of the tables, we have to scale the absolute value of the input by the number of values in the table, divide by the range of the table, and then scale the resulting fixed point integer back by the fixed point scale factor for use as an index into the table. The result from indexing into the table, may need to be scaled by the sign of the input, depending on the table being used. This scaling

results in a loss of information and in effect floors the value. This means that the result we get from the table is not as accurate as it could be because at times it should have retrieved the value that was in the following index; it should have rounded up. In our results, we've found that these lower accuracy rotations result in slowing the convergence of the SVD algorithm. In the future we will have to consider if providing better accuracy is worth the cost in performance.

With lookup tables as large as we are using, we assume that every index into the table will cause a cache miss, as the lookup table itself hasn't been optimised. As such, we can expect one load operation, a fixed point division, a right shift, some multiplication, and some branching per table lookup. Given that the native trigonometric functions are complex floating point operations, we expect the lookup operation to be more performant.

## 5.2 Result Uncertainty Range

An observation we have made while testing our algorithm, is that when the number of sweeps is increased significantly, the energy of the entire matrix is slowly decreased. We've noticed this behaviour after introducing the trigonometric lookup tables, so we're assuming that the total energy decrease can be attributed to the error introduced by the lookup tables.

We aren't too concerned about this behaviour, because our average % error for the diagonal elements of our resulting matrix after four sweeps is 0.35%, thus we are satisfied with the performance of our algorithm, though there is certainly room for improvement, as to not reduce the energies of the diagonal elements.

## 5.3 SIMD Matrix Multiplication

The produced assembly for both matrix multiplication algorithms can be seen in Appendix C 10.3 and 10.4 respectively. The standard algorithm performs 16 multiply and accumulate operations before branching and repeating 4 times, totaling 64 multiply and accumulate operations. The parallel algorithm performs a total of 16 Neon vector multiply operations.

If we assume that the Neon vector multiply operation takes twice as long as the standard multiply and accumulate operation, then the parallel would be at least twice as fast. This doesn't take into consideration all of the additional overhead which is required in the normal implementation, which is also greater than what is needed for the SIMD implementation.

## 5.4 Hypothetical Hardware Solution

We mocked up the ASIP functionality described in section 3.5. This was done by assuming that an assembly instruction 'ASIP' would take in 4 14 bit numbers (compressed into two 32 bit inputs), perform the sine and cosine operations needed, and return four values packed into a 32 bit output. The resulting assembly, of the relevant parts, can be seen in Appendix C 10.3. In comparison to the original assembly,

seen in Appendix C - 10.4,  the use of the ASIP function would be much more efficient. In particular, the ASIP implementation prevents 8 function calls, 8 fixed point divisions, all table lookups, and all the associated overhead.

Using the table in reference [10], we were able to approximate the number of clock cycles required to perform a table lookup. This approximation is based on the three most computationally expensive operations required: reading from RAM (assuming cache miss for table lookup), performing a function call, and fixed point division. In this manner, we approximate all of the trigonometric functions would require roughly 1500 clock cycles. As such, so long as the ASIP instruction took fewer than 1500 clock cycles, we would expect to see a performance improvement. If we estimate that the ASIP would take 1000 clock cycles, we would expect to see a 33% increase in speed.

# 6 Future Work

In our final implementation we found several areas of improvement:

- What input we provide
- Truncation vs rounding
- Lookup table optimization
- And problems with scaling and branching

The scope of this algorithm can be expanded to work for complex values. This would allow the system to be used in more applications. In the future, the performance costs of the complex algorithm should be compared to the real valued algorithm, to determine if the increased flexibility is worth the performance cost.

Currently, whenever we convert a floating point real value to a fixed point integer, perform a fixed point division, or perform a matrix multiplication, the result is truncated rather than rounded. Truncating in this way results in an increased loss of information in comparison to rounding. We would recommend investigating the performance cost and accuracy gained by using rounding instead of truncation.

In addition, we would recommend optimizing the trigonometric lookup tables. Experimentation could be done to find the most optimal ranges, step sizes, and table sizes. Additionally, the memory layout of the tables could be optimized to increase locality.

When we continue through iterations after converging we also find that the energy of our results decrease both along and off the diagonal. We hypothesized that this is due to the less accurate rotations, and one way to mitigate this is to stop our iterations after the change in the values falls below a specific value change epsilon.

More work could be done to optimally select the (i,j) pair. Ideally (i,j) would be chosen such that |mij| is the maximum non-diagonal element as suggested in the slides [2] which will provide more optimal 2x2

matrix selection. We hypothesize that this could increase the computation time on each iteration but it has the potential to reduce the number of iterations that are required.

It would be possible to further improve upon our parallel matrix multiplication by eliminating branching. We were unable to get the Neon intrinsics to work using pointers to the volatile matrices. As such, the parallel matrix multiplication has to branch to determine which of the two matrices to use (see Section 3.3 on reducing copies). Branching in this way is avoidable by swapping pointers to the matrices used for input and output every iteration.

# 7 Conclusion

In this project we explored our design process for creating a matrix diagonalization program in C using the Jacobi method on a 32-bit ARM machine. Once we implemented our base implementation we made optimizations in the arithmetic calculations, created a more efficient way to calculate the trigonometric functions by using lookup tables, parallelized tasks and removed unnecessary copies.

We also explored how a theoretical ASIP assembly instruction could optimize our solution.

In our design decisions we compared the use of a piecewise approximation and a lookup table in calculating our angle rotations and examined how we introduced fixed point notation in our algorithm. We then discussed some of the limitations of our design decisions and followed it up with some future work that can be done to further optimize or provide more flexibility in our solution.

What we found was that we can further explore flexibility in terms of the input values and size, how we can provide greater accuracy through our conversions, and identify why we are having problems with our result energies decreasing after iterations past our convergence. Overall we found that in this project we were able to provide an increase in speed, at the cost of some accuracy.

# 8 References


[1] L. Michael E. Wall, "Singular value decomposition and principal component analysis", *Public.lanl.gov*, 2003. [Online]. Available: https://public.lanl.gov/mewall/kluwer2002.html. [Accessed: 09- Aug- 2020].

[2] M. Sima, "SENG440 Embedded Systems - Lesson 112: Matrix Diagonalization", Online (Zoom), 2020.

[3] A. Donev, *Numerical Methods I - Singular Value Decomposition*. 2010, p. 19. [Online]. Available: https://cims.nyu.edu/~donev/Teaching/NMI-Fall2010/Lecture5.handout.pdf

[4] M. Sima, "SENG440 Embedded Systems - Lesson 6: Fixed-Point Arithmetic I", Online (Zoom), 2020.

[5] A. Ltd., "SIMD ISAs | Neon Intrinsics Reference – Arm Developer", *Arm Developer*, 2020. [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics?search=vld1q_s32. [Accessed: 06- Aug- 2020].

[6] A. Ltd., "SIMD ISAs | Neon Intrinsics Reference – Arm Developer", *Arm Developer*, 2020. [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics?search=vmlaq_n_s32. [Accessed: 06- Aug- 2020].

[7] A. Ltd., "SIMD ISAs | Neon Intrinsics Reference – Arm Developer", *Arm Developer*, 2020. [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics?search=vshrq_n_s32. [Accessed: 06- Aug- 2020].

[8] A. Ltd., "SIMD ISAs | Neon Intrinsics Reference – Arm Developer", *Arm Developer*, 2020. [Online]. Available: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics?search=vst1q_s32. [Accessed: 06- Aug- 2020].

[9] *An Optimized Matrix Multiplication on ARMv7 Architecture*. pp. 44-45. [Online]. Available: https://pdfs.semanticscholar.org/d1ac/a83d9a414336818962e449395a5bf7c57c3b.pdf

[10] ". Hare, "Infographics: Operation Costs in CPU Clock Cycles - IT Hare on Soft.ware", IT Hare on Soft.ware, 2016. [Online]. Available: http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/. [Accessed: 09- Aug- 2020].

# 9 Appendix A: Matrix Results

Input:
[31.0, 77.0, -11.0, 26.0]
[-42.0, 14.0, 79.0, -53.0]
[-68.0, -10.0, 45.0, 90.0]
[34.0, 16.0, 38.0, -19.0]

After 1 sweep:
[ 100.468750, -14.437500, 8.156250, 30.625000 ]
[ -25.296875, 104.578125, 1.171875, 4.250000 ]
[ -10.656250, -2.093750, -110.718750, -2.125000 ]
[ 3.140625, -0.578125, -2.093750, 37.500000 ]

After 2 sweeps:
[ 85.375000, 1.046875, 0.078125, 0.203125 ]
[ -6.000000, 126.109375, 0.140625, -0.109375 ]
[ -0.984375, 0.203125, -110.828125, -0.062500 ]
[ -0.390625, -0.046875, -0.109375, 33.921875 ]

After 3 sweeps:
[ 85.203125, -0.015625, -0.078125, 0.031250 ]
[ -0.093750, 126.234375, 0.031250, -0.140625 ]
[ -0.125000, 0.031250, -110.843750, -0.062500 ]
[ -0.125000, -0.046875, -0.140625, 33.843750 ]

After 4 sweeps:
[ 85.109375, 0.000000, -0.093750, -0.031250 ]
[ -0.078125, 126.171875, 0.015625, -0.093750 ]
[ -0.156250, 0.015625, -110.843750, -0.062500 ]
[ -0.078125, 0.000000, -0.078125, 33.765625 ]

Expected output:
[85.570, 0, 0 , 0]
[0, 126.429, 0, 0]
[0, 0, -110.905, 0]
[0, 0, 0, 34.008]

# 10 Appendix B: Code

All of our code has been uploaded to github, and can be found here: https://github.com/rtulip/seng440project. This may be useful for insights in how the implementation changed over time.

For convenience, the code has been provided alongside this document in a zip file.

# 11 Appendix C: Assembly

## 11.1 Matrix Multiply NEON

```
matrix_multiply_NEON:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 0, uses_anonymous_args = 0
    @ link register save eliminated.
    movw    r3, #:lower16:.LANCHOR0
    movt    r3, #:upper16:.LANCHOR0
    mov    r2, r3
    vmov.i32    d5, #0  @ v2si
    vld1.32    {d24-d25}, [r2:64]!
    vmov    d7, d5  @ v2si
    add    r0, r3, #32
    add    r1, r3, #48
    vld1.32    {d16-d17}, [r2:64]
    vld1.32    {d22-d23}, [r0:64]
    vld1.32    {d20-d21}, [r1:64]
    ldr    r1, [r3, #64]
    ldr    r2, [r3, #68]
    vmov.32    d7[0], r1
    vmov.32    d5[0], r2
    vmov.i32    d6, #0  @ v2si
    ldr    r2, [r3, #72]
    vmul.i32    q13, q8, d5[0]
    vmul.i32    q9, q12, d7[0]
    vmov.32    d6[0], r2
    vmov.i32    d7, #0  @ v2si
    ldr    r2, [r3, #76]
    vadd.i32    q9, q9, q13
    vmov.32    d7[0], r2
```

```
vmul.i32    q13, q11, d6[0]
movw    r2, #:lower16:.LANCHOR1
vadd.i32    q9, q9, q13
vmul.i32    q13, q10, d7[0]
movt    r2, #:upper16:.LANCHOR1
mov    r1, r2
vadd.i32    q9, q9, q13
vmov.i32    d5, #0  @ v2si
vst1.32    {d18-d19}, [r1:64]!
vmov    d7, d5  @ v2si
ldr    ip, [r3, #80]
ldr    r0, [r3, #84]
vmov.32    d7[0], ip
vmov.32    d5[0], r0
vmov.i32    d6, #0  @ v2si
ldr    r0, [r3, #88]
vmul.i32    q13, q8, d5[0]
vmul.i32    q9, q12, d7[0]
vmov.32    d6[0], r0
vmov.i32    d7, #0  @ v2si
ldr    r0, [r3, #92]
vadd.i32    q9, q9, q13
vmov.32    d7[0], r0
vmul.i32    q13, q11, d6[0]
vadd.i32    q9, q9, q13
vmul.i32    q13, q10, d7[0]
vadd.i32    q9, q9, q13
vmov.i32    d7, #0  @ v2si
vst1.32    {d18-d19}, [r1:64]
vmov    d5, d7  @ v2si
ldr    r0, [r3, #96]
ldr    r1, [r3, #100]
vmov.32    d7[0], r0
vmov.32    d5[0], r1
vmov.i32    d6, #0  @ v2si
ldr    r1, [r3, #104]
vmul.i32    q13, q8, d5[0]
vmul.i32    q9, q12, d7[0]
vmov.32    d6[0], r1
vmov.i32    d7, #0  @ v2si
ldr    r1, [r3, #108]
vadd.i32    q9, q9, q13
vmov.32    d7[0], r1
```

```
    vmul.i32    q13, q11, d6[0]
    vadd.i32    q9, q9, q13
    vmul.i32    q13, q10, d7[0]
    vadd.i32    q9, q9, q13
    vmov.i32    d5, #0  @ v2si
    add    r1, r2, #32
    vst1.32    {d18-d19}, [r1:64]
    vmov    d7, d5  @ v2si
    ldr    r0, [r3, #112]
    ldr    r1, [r3, #116]
    vmov.32    d7[0], r0
    vmov.32    d5[0], r1
    vmov.i32    d6, #0  @ v2si
    ldr    r1, [r3, #120]
    vmul.i32    q12, q12, d7[0]
    vmul.i32    q8, q8, d5[0]
    vmov.32    d6[0], r1
    vmov.i32    d7, #0  @ v2si
    ldr    r3, [r3, #124]
    vadd.i32    q8, q8, q12
    vmul.i32    q11, q11, d6[0]
    vmov.32    d7[0], r3
    vadd.i32    q8, q8, q11
    vmul.i32    q10, q10, d7[0]
    vadd.i32    q8, q8, q10
    add    r2, r2, #48
    vst1.32    {d16-d17}, [r2:64]
    bx    lr
    .size    matrix_multiply_NEON, .-matrix_multiply_NEON
    .section    .text.startup,"ax",%progbits
    .align    2
    .global    main
    .syntax unified
    .arm
    .fpu neon
    .type    main, %function
```

## 11.2 Naive Matrix Multiply

```
matrix_multiply:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 0, uses_anonymous_args = 0
```

```
    mov    ip, #0
    push    {r4, r5, r6, r7, lr}
    mov    r4, ip
.L9:
    str    r4, [r2, ip]
    ldr    r3, [r0, ip]
    ldr    r6, [r1]
    ldr    r5, [r2, ip]
    add    lr, r0, ip
    mla    r3, r6, r3, r5
    str    r3, [r2, ip]
    ldr    r5, [lr, #4]
    ldr    r7, [r1, #16]
    ldr    r6, [r2, ip]
    add    r3, r2, ip
    mla    r5, r7, r5, r6
    str    r5, [r2, ip]
    ldr    r5, [lr, #8]
    ldr    r7, [r1, #32]
    ldr    r6, [r2, ip]
    mla    r5, r7, r5, r6
    str    r5, [r2, ip]
    ldr    r5, [lr, #12]
    ldr    r7, [r1, #48]
    ldr    r6, [r2, ip]
    mla    r5, r7, r5, r6
    str    r5, [r2, ip]
    str    r4, [r3, #4]
    ldr    r5, [r0, ip]
    ldr    r7, [r1, #4]
    ldr    r6, [r3, #4]
    mla    r5, r7, r5, r6
    str    r5, [r3, #4]
    ldr    r5, [lr, #4]
    ldr    r7, [r1, #20]
    ldr    r6, [r3, #4]
    mla    r5, r7, r5, r6
    str    r5, [r3, #4]
    ldr    r5, [lr, #8]
    ldr    r7, [r1, #36]
    ldr    r6, [r3, #4]
    mla    r5, r7, r5, r6
    str    r5, [r3, #4]
```

```
ldr    r5, [lr, #12]
ldr    r7, [r1, #52]
ldr    r6, [r3, #4]
mla    r5, r7, r5, r6
str    r5, [r3, #4]
str    r4, [r3, #8]
ldr    r5, [r0, ip]
ldr    r7, [r1, #8]
ldr    r6, [r3, #8]
mla    r5, r7, r5, r6
str    r5, [r3, #8]
ldr    r5, [lr, #4]
ldr    r7, [r1, #24]
ldr    r6, [r3, #8]
mla    r5, r7, r5, r6
str    r5, [r3, #8]
ldr    r5, [lr, #8]
ldr    r7, [r1, #40]
ldr    r6, [r3, #8]
mla    r5, r7, r5, r6
str    r5, [r3, #8]
ldr    r5, [lr, #12]
ldr    r7, [r1, #56]
ldr    r6, [r3, #8]
mla    r5, r7, r5, r6
str    r5, [r3, #8]
str    r4, [r3, #12]
ldr    r5, [r0, ip]
ldr    r7, [r1, #12]
ldr    r6, [r3, #12]
add    ip, ip, #16
mla    r5, r7, r5, r6
str    r5, [r3, #12]
ldr    r5, [lr, #4]
ldr    r7, [r1, #28]
ldr    r6, [r3, #12]
cmp    ip, #64
mla    r5, r7, r5, r6
str    r5, [r3, #12]
ldr    r5, [lr, #8]
ldr    r7, [r1, #44]
ldr    r6, [r3, #12]
mla    r5, r7, r5, r6
```

```
    str    r5, [r3, #12]
    ldr    lr, [lr, #12]
    ldr    r6, [r1, #60]
    ldr    r5, [r3, #12]
    mla    lr, r6, lr, r5
    str    lr, [r3, #12]
    bne    .L9
    pop    {r4, r5, r6, r7, pc}
    .size    matrix_multiply, .-matrix_multiply
    .align    2
```

## 11.3 Custom ASIP

Below is the assembly generated for the relevant sections of our code which would be impacted by using the ASIP instruction described in section 3.5. This section should be compared with Appendix C - 10.4 to better understand the performance impact of using the ASIP instruction.

Note: The ‘ASIP’ instruction is a placeholder for the actual ASIP instruction.

```
ldrsh    r3, [fp, #-32]
    lsl    r2, r3, #16
    ldrsh    r3, [fp, #-30]
    orr    r3, r2, r3
    str    r3, [fp, #-48]
    ldrsh    r3, [fp, #-34]
    lsl    r2, r3, #16
    ldrsh    r3, [fp, #-36]
    orr    r3, r2, r3
    str    r3, [fp, #-52]
    ldr    r3, [fp, #-48]
    ldr    r2, [fp, #-52]
    .syntax divided
@ 298 "src/svd.c" 1
    ASIP    r3, r3, r2

@ 0 "" 2
    .arm
    .syntax unified
    str    r3, [fp, #-56]
    ldr    r3, [fp, #-56]
    lsr    r3, r3, #20
    str    r3, [fp, #-60]
```

```
    ldr    r3, [fp, #-56]
    asr    r3, r3, #17
    and    r3, r3, #1
    str    r3, [fp, #-64]
    ldr    r3, [fp, #-56]
    asr    r3, r3, #3
    ubfx    r3, r3, #0, #12
    str    r3, [fp, #-68]
    ldr    r3, [fp, #-56]
    and    r3, r3, #7
    str    r3, [fp, #-72]
```

## 11.4 Without ASIP

Below is the assembly generated for the relevant sections of our code which would be impacted by using the ASIP instruction described in section 3.5. This section should be compared with Appendix C - 10.3 to better understand the performance impact of using the ASIP instruction.

```
ldrh    r2, [fp, #-32]
    ldrh    r3, [fp, #-30]
    add    r3, r2, r3
    uxth    r3, r3
    sxth    r0, r3
    ldrh    r2, [fp, #-36]
    ldrh    r3, [fp, #-34]
    sub    r3, r2, r3
    uxth    r3, r3
    sxth    r3, r3
    mov    r1, r3
    bl    fixed_point_div
    str    r0, [fp, #-48]
    ldr    r0, [fp, #-48]
    bl    arctan_lookup
    mov    r3, r0
    strh    r3, [fp, #-50]    @ movhi
    ldrh    r2, [fp, #-32]
    ldrh    r3, [fp, #-30]
    sub    r3, r2, r3
    uxth    r3, r3
    sxth    r0, r3
    ldrh    r2, [fp, #-36]
    ldrh    r3, [fp, #-34]
    add    r3, r2, r3
```

```
uxth    r3, r3
sxth    r3, r3
mov    r1, r3
bl    fixed_point_div
str    r0, [fp, #-48]
ldr    r0, [fp, #-48]
bl    arctan_lookup
mov    r3, r0
strh    r3, [fp, #-52]    @ movhi
ldrsh    r2, [fp, #-50]
ldrsh    r3, [fp, #-52]
sub    r3, r2, r3
asr    r3, r3, #1
str    r3, [fp, #-56]
ldrsh    r2, [fp, #-50]
ldr    r3, [fp, #-56]
sub    r3, r2, r3
str    r3, [fp, #-60]
ldr    r0, [fp, #-56]
bl    sin_lookup
mov    r3, r0
str    r3, [fp, #-64]
ldr    r0, [fp, #-56]
bl    cos_lookup
mov    r3, r0
str    r3, [fp, #-68]
ldr    r0, [fp, #-60]
bl    sin_lookup
mov    r3, r0
str    r3, [fp, #-72]
ldr    r0, [fp, #-60]
bl    cos_lookup
mov    r3, r0
str    r3, [fp, #-76]
```