APPM 4600 Sparse Solvers Rough Draft

Christopher Gonzalez-millan, Aaron Schwan

April 2023

1 Introduction and Background

The Thomas Algorithm is an extension of LU-Decomposition for a sparse matrix focusing on the tridiagonal matrices. The form of the Thomas algorithm is very useful for matrix systems with strong diagonal components, such as finite element analysis for a stiffness solver, modeling tidal flows, and even computational fluid dynamics [1, 3, 4]. The Thomas algorithm has an ideal case where it allows for the efficient solving of a $n \times n$ matrix using a $\mathcal{O}(n)$ rather than a $\mathcal{O}(n^3)$ order such as Gaussian elimination the other benefit is that it also requires less memory and storage for a sparse matrix than a fully populated one with reduction techniques readily available in multiple programming languages.

$$\mathcal{T}\vec{x} = \vec{r} \tag{1}$$

Equation 1: The basic form of the Thomas algorithm [4]

With all these benefits, there is a cost. That is the fact that this is not a stable solution for all matrices and instead requires the ideal case of a tridiagonal matrix shown by Eqn. 2 however, there are common extensions to account for other matrix topologies. This stability is an issue that can be quickly checked to ensure a matrix is appropriately conditioned for this algorithm.

$$\mathcal{T} = \begin{pmatrix} b_1 & c_1 & 0 & \cdots & \cdots & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & & & \vdots \\ 0 & a_3 & b_3 & c_3 & \ddots & & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & 0 \\ \vdots & & & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \cdots & \cdots & \cdots & 0 & a_n & b_n \end{pmatrix} \tag{2}$$

Equation 2: An example of a Tridiagonal matrix

To solve Eqn. 1, we have two steps we need to follow. First, we break the $\mathcal{T}$ matrix into its upper and lower triangular matrix forms L and U Eqn. 3. Then we perform a forward and backward solution for both matrices.

$$\mathcal{T} = L \cdot U \tag{3}$$

Equation 3: LU decomposition

We can use a use two convient formulas to convert $\mathcal{T}$ into a LU matrix Eqn. 5 and Eqn. 4.

$$b'_k = b_k - a'_k c_{k-1} \tag{4}$$

Equation 4: adjusted b values for LU decomposition

$$a'_k = \frac{a_k}{b'_{k-1}} \tag{5}$$

Equation 5: adjusted a values for LU decomposition

From the above equations, we can quickly see that there is no need for updating c_k values, which allows for another increase in speed and storage.

With the matrix decomposed, we can simply forward substitute using Eqn. 6

$$r'_k = r_k a'_k r_{k-1} \tag{6}$$

Equation 6: Forward substitution

and finally, we can back substitute using Eqn. 7

$$x_k = \frac{r'_k - c_k x_{k+1}}{b'_k} \tag{7}$$

Equation 7: backward substitution

From this, we can see how the system is more memory efficient because of the fact we only need to store five vectors, and we can update them inside the vector. This will have larger and larger amounts of data saving as we increase the size of the matrix. Something also to note is when we exit the boundary conditions, we will have zeros for example, in Eqn. 7 x_{n+1} at the very end will be 0, which allows for a reduction in the equation to $x_n = r'_n/b'_n$.

2 Finite Element Analysis

2.1 Introduction

One commonly used method in numerical analysis is the finite element method (FEM). It has a wide application in physics and engineering because it is a flexible and adaptable tool for solving differential equations numerically. The basic idea of FEM is to divide the domain of our problem into small and individual patches where it is easier to find solutions that satisfy the differential equation. We can then use boundary conditions such that each patch is affected by its nearest neighbors. Similar to a divide and conquer algorithm, this allows us to find a global solution by solving smaller problems first.

2.2 Discretization and Coordinate System

The first step of using a FEM is discretization. To be able to visualize this process, we will look at the two-dimension irregularity of a domain. In this dimension the best to use a mesh of triangles to create subdomains. The number of triangles is arbitrary, but the size of triangles is determined by the resolution required within that subdomain. Locations where there are larger changes in the dependent variable should expect more triangles. This is done in an attempt to get higher accuracy.

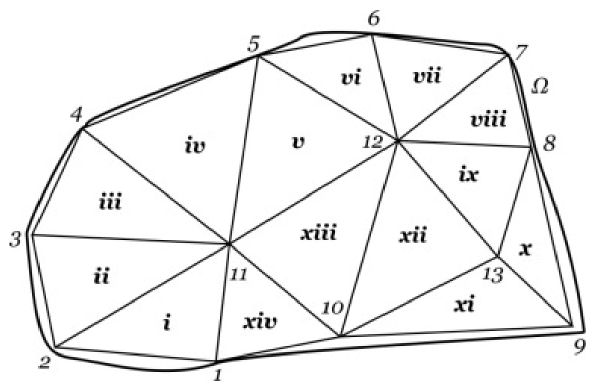

Figure 1: Mesh Discretization

In Fig. 1, we find a visualization of the democratization of an irregular domain. At the beginning of the FEM process, a number is assigned to each triangle in this case, that is done with Roman numerals. Similarly, each vertex is labeled by a different number in this case, Arabic numerals are used.

The next step in the discretization process is a way to describe each subdomain in an equation. This could easily be done in x- and y-coordinate with pieces-wise equations describing each side of the triangle. The issue with this approach is that it makes each triangle completely dependent on its neighbor. In addition, this would increase the complexity of the differential equation problem and drive us away from the linear system that we desire. The approach to this is the use of Lagrangian Coordinates.

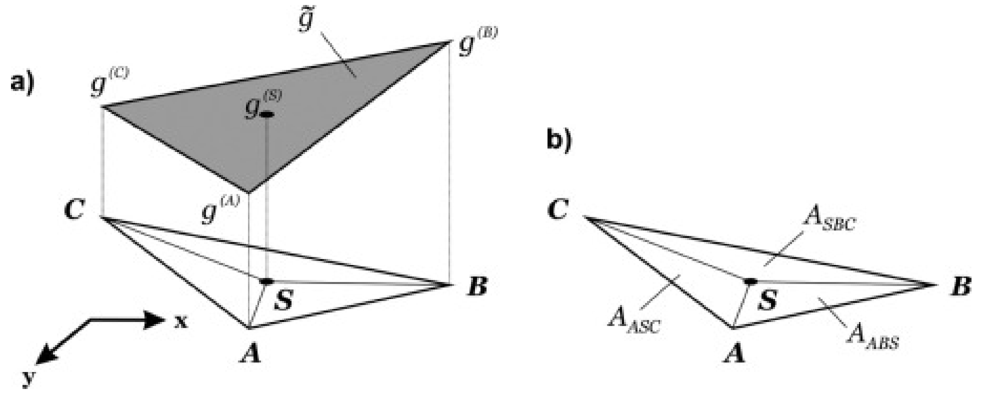

Figure 2: Lagrange Coordinate System

$$\begin{aligned}
\xi_A &= \frac{A_{ASC}}{A_{ABC}} = \frac{A_{ASC}}{A_{ASC} + A_{ABS} + A_{SBC}} \\
\xi_B &= \frac{A_{ABS}}{A_{ABC}} = \frac{A_{ABS}}{A_{ASC} + A_{ABS} + A_{SBC}} \\
\xi_C &= \frac{A_{SBC}}{A_{ABC}} = \frac{A_{SBC}}{A_{ASC} + A_{ABS} + A_{SBC}} \\
\xi_A + \xi_B + \xi_C &= \frac{A_{ASC} + A_{ABS} + A_{SBC}}{A_{ASC} + A_{ABS} + A_{SBC}} = 1
\end{aligned} \tag{8}$$

Equation 8: Lagrangian surface coordinates[2].

This method is easy to set up. It establishes a coordinate system that is specific to the triangle edges and can only be applied within the triangle itself. When two coordinates have a value of zero, the resulting point corresponds to one of the triangle's vertices. On the other hand, if only one coordinate has a value of zero, the point falls on the edge of the triangle. This can be mathematically formulated in Eqn. 8

$$\tilde{g}^{(j)}(\xi_A, \xi_B, \xi_C) = g^{(j,A)}\xi_A + g^{(j,B)}\xi_B + g^{(j,C)}\xi_C \tag{9}$$

Equation 9: Local Reconstruction Function [2, (eq 32.1)].

This reconstruction function is applicable specifically within triangle j, where j refers to the triangle's index in the figure.Fig. 2. Note that $g(j, A, B, C)$ are the values of the dependent function at vertices A, B, and C.

paragraph about how this coordinate system is used to find the Galerkin method

2.3 Galerkin Method

It is important to note that to be able to define the Lagrange coordinates, you would have to completely define the domain of the problem. If we knew the domain of the problem finding the solution to the differential equation would not be that difficult. Therefore the Galerkin method is used, which is a numerical technique used to approximate solutions to differential equations. The basic idea of this method is to use a weighted residual equation which is formed by multiplying the differential equation with a weight function.Eqn. 9 The weighted functions are chosen to satisfy certain properties. In general, FEM problems choose to use localized-support functions, which refer to functions that exist only within a narrow interval and have a value of zero outside of that interval. In one-dimensional FEM, hat functions are most commonly used.

2.3.1 Hat Functions

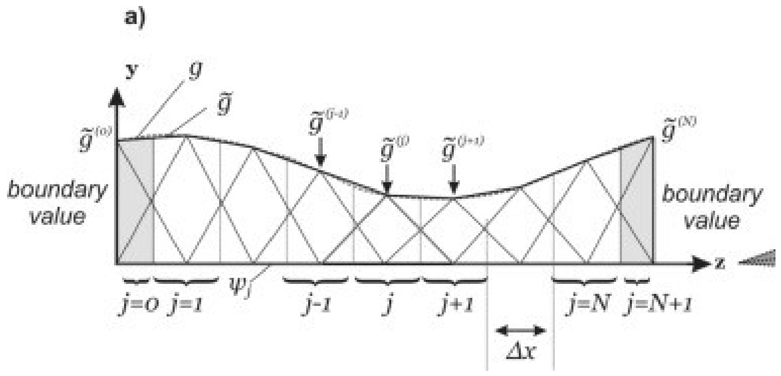

Figure 3: Hat Functions

To begin, the domain is partitioned into intervals that may vary in size. Within each interval j, a piecewise-linear hat function ψj is constructed such that it attains the central value $\tilde{g}^{(j)}$ of the interval. This results in localized support for the solution function $\tilde{g}$.

2.4 FEM-Galerkin Method

Galerkin method assumes that the differential equations take this weak from where L is a function of the independent variables $x, g, \frac{dg}{dx}, \frac{d^2g}{dx^2}$ which are functions of the real independent variable x. The notation has been adapted to accommodate multiple dependent variables, denoted as $x_1, x_2, ..., x_N$ and the domain has been redefined as Ω.

$$\int \psi_j(\vec{x}) L(\vec{x}, g, \frac{dg}{dx}, \frac{d^2g}{dx^2}) d\Omega = 0 \tag{10}$$

Equation 10: FEM-Galerkin Method [2, (eq 32.17)]

From this weak form, we are able to derive identities for when the differential equation takes on the forms of Sources, Sinks, Diffusion, and Divergence. Source and sink terms result in constants that can be integrated directly over the domain Eqn. 10. In most cases, the divergence of the dependent variable g leads to an integral that cannot be further simplified [2, (eq. 32.19)]. The third scenario we will address pertains to diffusion effects involving the quantity of the dependent variable, which are characterized by Laplace operators [equation reference]. In [chapter 29], an in-depth proof can be found that justifies why the Gerlakin method functions, while an in-depth expiation of how to derive this integral can be found in [2].

$$\begin{aligned} \int \psi_j(\vec{x}) \cdot c \, d\Omega &= c \int \psi_j(\vec{x}) \, d\Omega \\ &\int_\Omega \psi_j(\vec{x}) \, (\vec{\nabla} \cdot g) \, d\Omega \\ \int_\Omega \psi_j(\vec{x}) \, \Delta g \, d\Omega &= - \int_\Omega (\vec{\Delta} \cdot \psi_j)(\vec{\Delta} \cdot g) \, d\Omega \end{aligned} \tag{11}$$

Equation 11: $\vec{r_0}$ calculation

Poiseuille flow in a planar infinitesimally extended channel.jpg

2.5 FLOW IN INFINITESIMALLY EXTENDED CHANNELS

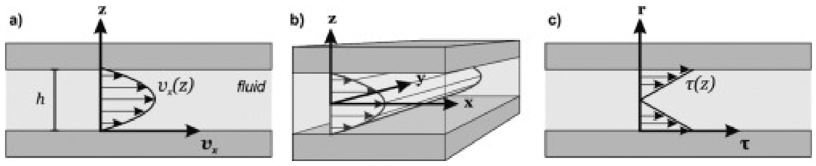

Figure 4: Poiseuille flow in a planar infinitesimally extended channel [2]

A common 1-dimensional problem that is looked at for FEM problems is the Poiseuille flow in the planar infinitesimally extended channel. In layman's terms, this Poiseuille flow explains the pressured-induced flow in a long duct, usually a pipe. Poiseuille flow in a planar infinitesimally extended channel describes this flow between two parallel plates, where the fluid velocity is directly proportional to the pressure gradient and the fourth power of the distance from the plates. This means that the fluid flows faster near the center of the channel and slower near the walls. The flow is characterized by a parabolic velocity profile, with zero velocity at the walls and maximum velocity at the center of the channel Fig. 4. The flow within infinitesimally extended channels is regarded as one-dimensional since we assume that all variations along the y-axis are 0, as shown in [reference to flow figure]. With these assumptions and the naiver-stokes, a second-order ODE can be derived that is able to model this behaviorEqn. 12. A more in-depth proof can be found in [reference to chapter 16.2].

$$\frac{d^2 v_x}{dz^2} = \frac{1}{\eta} \frac{dp}{dx} \tag{12}$$

Equation 12: 1D Naiver-Stokes with no-slip boundary condition.

In this ODE, the function $v_x(z)$ describes the velocity profile for the planer infinitesimal extend channel in the x-axis. As we can see in [reference to flow], we expect this function to be a parabola with the maximum velocity at the center. Therefore, this function's second derivative references the flow's sheer, which can be visualized in Fig. 4. The function $p(z)$ references the flow or pressure in the channel, and the first derivative references the change of this pressure through the channel. Like in most flow problems, the Greek eta η references the viscosity of the fluid we are trying to describe in the problem. Note that most flow problems like to assume that the liquid is 20 degrees Celsius water.

This paper will take a look at this problem with the following assumptions being made. First, the viscosity (η) will take on the value of $1m\ Pas$. Second, the change in pressure ($\frac{dp}{dx}$) is a constant with the value of $-0.001\text{mbar}\,\text{mm}^{-1}$. These values allow us to rewrite Eqn. 12 to Eqn. 13 where the domain is $-1 \leq z \leq 1$

$$\frac{d^2 v_x}{dz^2} + 100 = 0 \tag{13}$$

Equation 13: Problem setup ODE

The next step is to define the hat function of this function to be able to determine a basis for the problem. first is to split the domain into intervals of equal size according to [reference hat visual]. Note from the ODE that $\tilde{g}$ is a function that is dependent only on Z as it is the only independent variable.

$$\psi_j(z) = \tilde{g}^{(j)} \begin{cases} \frac{z - z^{(j-i)}}{z^{(j)} - z^{(j-i)}} & \text{if } z^{(j-1)} \leq z \leq z^{(j)} \\ \frac{z^{(j-+i)} - z}{z^{(j+i)} z^{(j)}} & \text{if } z^{(j)} \leq z \leq z^{(j+1)} \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

Equation 14: Shape Function

Our approximation function $\tilde{g}$ on the domain is thus given by [reference to equation].

$$\tilde{g}(z) = \sum_{j=1}^{N} \tilde{g}^{(j)} \psi_j(z) \tag{15}$$

Equation 15: Weights

The integration equation depicted in [2, (Eq 32.17)] is established at this point. Note given that our differential operator only has a single entry and contains a constant and a diffusion term, we can use the identities described in [2]. This allows us to rewrite are integral as [2, (Eq 32.18)].

$$\begin{aligned}
&\int_{\Omega_j} \psi_j(z)\left(\frac{d^2\tilde{g}}{dz^2}+100\right) d\Omega = 0 \\
&\int_{\Omega_j} \frac{\psi_j}{dz}\frac{d\tilde{g}}{dz}\, d\Omega = 100\int_{\Omega_j} \psi_j(z)\, d\Omega
\end{aligned} \tag{16}$$

Equation 16: Integration of Galerkin Method [2, (eq 32.24)]

Finally, the approximation function of $\tilde{g}$, the weight function, and the basis function (hat functions) have been clearly defined. Using the fact that equally spaced intervals were purposely used alongside some calculus the Glerkin integral can go through several simplifications that are outlined in-depth in [2]. This results in the integral found in Eqn. 17 where the variable j denotes each interval in the hat functions.

$$\int_{\Omega_j} \frac{\psi_j}{dz}\sum_{i=i}^{N} \tilde{g}^{(i)}\frac{\psi_i}{dz}\, d\Omega = 100\Delta z \tag{17}$$

Equation 17: Discretized integration [2, (eq 32.32)]

2.6 Matrix Notation

Up to this point, we have a second-order ODE flow problem. The domain of this problem has been democratized, and using the Galerkin method, we have set up an approximation for the ODE. At this point, the only thing left to do is to define the system of equations that are described by Eqn. 16 and solve, keeping in mind the boundary conditions given to us by the example problem.

Now intuitively, we can build a reference on how the matrix will be built using the Eqn. 17. Taking into account the left-hand side, in particular, $\sum_{i=i}^{N} \tilde{g}^{(i)}\frac{\psi_i}{dz}$ it is important to recall that the local support functions were designed with an overlap of one interval. As a result, we can break the domain into three, which defines three integrals. There is the left-hand side that functions that define the interval from $z(j) - \Delta z$ to $z(j)$. Next is the right-hand side function that defines the interval from $z(j)$ to $z(j) + \Delta z$. Finally, the hat function defines the whole interval, which can be described from $z(j) - \Delta z$ to $z(j) + \Delta z$. This means that we can simplify the Galerkin integral into Eqn. 10. A more depth formulation of how we arrived at this result can be found in *Finite Element Method Chapter 32*[2].

Next, we need to take into account the boundaries of the channel to describe the rest of the matrix. we note from [reference hat image] that the left and right-hand boundaries are only defined by two hat functions. From the previous result, it is easy to formulate that the left-hand side will be defined by the first minus the second hat function. As for the right-hand boundary, we have to note that number of intervals has been kept ambiguous and denied as N. intuitively we can note that we would expect the left-hand side boundary to be defined by the second to last hat function minus the last functionEqn. 18. A more depth formulation of how we arrived at this result can be found in *Finite Element Method Chapter 32*[2].

$$\begin{aligned}
\tilde{g}^{(0)} - \tilde{g}^{(1)} &= \frac{dp}{dx}\frac{1}{2\eta}(\Delta z)^2 j = 0 && \text{left-hand boundary} \\
-\tilde{g}^{(j-1)} + 2\tilde{g}^{(j)} + \tilde{g}^{(j+1)} &= \frac{dp}{dx}\frac{1}{\eta}(\Delta z)^2 && \text{all non-boundary intervals} \\
\tilde{g}^{(N-1)} - \tilde{g}^{(n)} &= \frac{dp}{dx}\frac{1}{2\eta}(\Delta z)^2 && \text{right-hand boundary}
\end{aligned} \tag{18}$$

Equation 18: Boundary conditions for Galerkin method.

This leaves us with a definition for an N size matrix with intervals of size Δz. At this point, you can decide how large or small you want this number to be, but as N goes to infinity and as Δz goes to zero, your approximation gets closer and closer to the real solution. Boundary conditions can also be taken into account that can change the matrix, such as Neumann conditions. In the case of our example, the boundary values are already defined. Since the no-slip condition is imposed at the boundaries, we can conclude that the velocity at these points is zero. This means that we only need to consider N equations instead of $N + 2$, as we can eliminate the first (left-hand boundary) and last(right-hand boundary) equations resulting from the boundary conditions.

For example, if we were to define $N = 5$ and $\Delta z = 0.2$ we would be left with a regular linear system of type $\boldsymbol{A}\vec{x} = \vec{\boldsymbol{b}}$ which is visualized in [matrix equation]. Note that the stiffness matrix in FEM is denoted as matrix $\boldsymbol{A}$, while the load vector is referred to as vector $\vec{\boldsymbol{b}}$.

$$\begin{pmatrix} -2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} \tilde{g}^{(1)} \\ \tilde{g}^{(2)} \\ \tilde{g}^{(3)} \\ \tilde{g}^{(4)} \\ \tilde{g}^{(5)} \end{pmatrix} = \begin{pmatrix} 4 \\ 4 \\ 4 \\ 4 \\ 4 \end{pmatrix} \tag{19}$$

Equation 19: An example of a Tridiagonal matrix

Finally, at this point, we can see in 1-D FEM with piece-wise linear elements, you can get a tri-diagonal stiffness matrix, equivalent to a second-order finite difference method. By applying the Thomas algorithm, we can efficiently solve the nodal values and obtain the solution to the problem. This approach is commonly used in various fields, including structural analysis, fluid dynamics, and heat transfer.

3 Numerical Implementation

The Numerical analysis has a couple of major speedups we can take advantage of with the Thomas algorithm. The first one is we can notice that the solution requires that the pipe has a non-slip condition on the pipe. This means that the solution will be $V_x = 0$ at $x = \pm l$. This is a minor speedup initially. However, by removing the outer values of the Galerkin matrix, we can clearly see that for all values $a_k = -1$, $b_k = 2$, and $c_k = -1$, allowing us to simplify the forward substitution for this problem.

$$\alpha_k = b_i - \frac{a_i c_{i-1}}{\alpha_{i-1}} = 2 - \frac{1}{\alpha_{i-1}} \tag{20}$$

From the setup of the Thomas algorithm, we can also find that $\alpha = 2$ and Eqn. 20 we can find a telescoping series for $alpha$ shown in Eqn. 21.

$$\alpha_i = \frac{i+2}{i+1} \tag{21}$$

Equation 21: Simplified α

$$r_0 = \frac{dp}{dz}\frac{1}{\eta} \tag{22}$$

Equation 22: Simplified vector notation for the Gerlakin Method

This is already a massive advantage, but we can do the same definitions for the calculation of β. We can first recognize that $\beta_0 = r_0/2$ where r_0 is the common value for the solution vector shown in Eqn. 22.

$$\beta_i = \frac{r_i - \alpha_i \beta_{i-1}}{\alpha_i} = \frac{r_0(i+1) - b_{i-1}(i+2)}{i+2} \tag{23}$$

By setting $\beta_{-1} = 0$ due to the no-slip condition. We can simplify the Eqn. 23 into Eqn. 24.

$$\beta_{j+1} = r_0 \frac{i+1}{2} \tag{24}$$

Equation 24: Simplified β

These simplifications make it so that we no longer have a need for forward substitution with the Thomas algorithm. This has two major advantages the first is the speed of the calculation, as we are no longer required to perform the forward substitution to solve the problem and instead have the answer. The second advantage is that we have now decoupled the resolution of the simulation from a majority of the memory required for the calculations.

This lack of memory allows for a much finer resolution to be achieved while not utilizing more space than is strictly required. For example, if there is a desire for high resolution at one point, it is now possible to step over to the point on the axis only, saving the values of the previous steps. This is opposed to making a matrix $n \times n$ where $n = 1/dz$ this means that we utilize $1/n$ the space of a given resolution, or in other words, compared to the Gauss's row elimination implementation, we can have a n times finer resolution for the same amount of memory, given we have not run into floating point precision.

So far, the implementations have been speed-ups solely based on further simplification, but with the actual implementation of this algorithm, we will want to mirror across the y-axis. This is because it is an even-mode solution. This also maintains a higher precision due to the fact that during the process of back substitution, we will accrue rounding and truncation errors we want to limit the number of times we recursively find n. It is also noted that because of the well-behaved nature of this problem, we could find an index representation of V_x by a similar process of finding α and β however, it was not a readily generalized method. Although the values for our problem have required $a_k = -1$, $b_k = 2$ and, $c_k = -1$ it is not an uncommon situation for at least for 1D FEA that the values of a, b and, c are constant.

3.1 CPU vs GPU Implementation

We are looking at a highly linear process for CPU optimization where each line can be executed immediately after the other. This means we want to minimize the time spent on every step as much as possible. This involves limiting the amount of new variable definitions, list lengths, and the number of function calls. This in code looks like inserting α and β directly into the equation for V_x and allocating the vector for V_x.

If we want to optimize for use on a GPU, we need to allow for more parallelized tasks. This looks like defining α and β as functions instead so that the calculation of V_x is broken up to be calculated by several parts of the GPU. By calculating two different lists, we have increased the total memory needed for a given resolution but at a major time that may be more important for a given solution.

4 Results

We have the results of the numerical approximation of an infinitesimal fluid duct can be seen in Fig. 5. We can observe that this method happens to overestimate the analytical function for all values of the estimate.

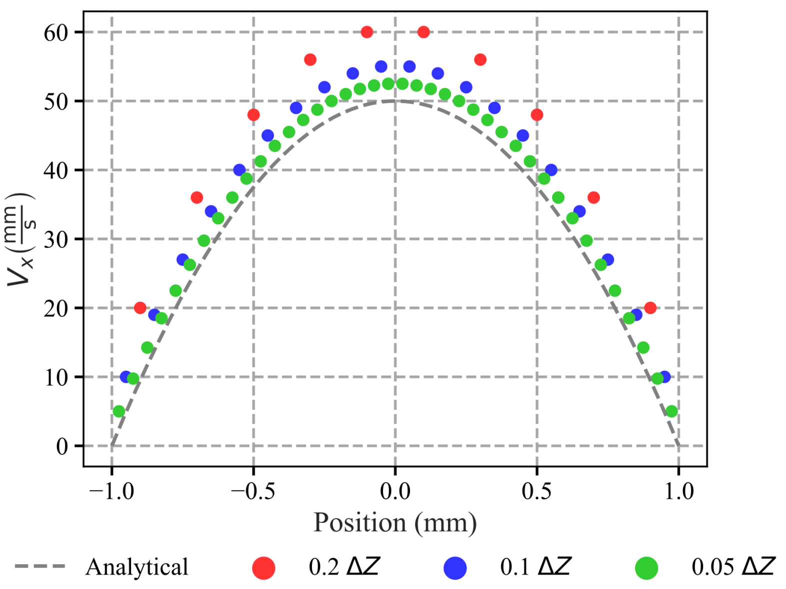

Figure 5: This shows how the analytical solution is approached by the numerical solution

A simple observation is that we do not reduce the error by the same amount each time we half the spacing between the numerical solutions. This can be seen by looking at the maximum error in Fig. 5 we can then test this by taking the maximum error for several different spacing values for $dz = 1/n$. This is done in Fig. 6 for a $n \times n$ matrix for values of $n \in \{10 \leq n \leq 20,000 | \mathbb{Z}^*\}$.

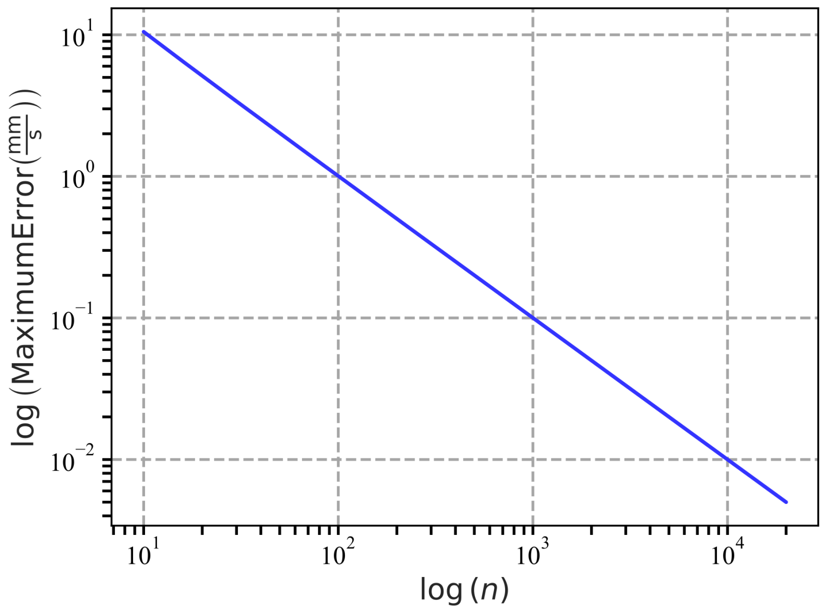

Figure 6: The maximum error on a $n \times n$ matrix solution for an infinitesimal pipe problem.

From Fig. 6, we can see that there is a linear correlation between reducing the error and increasing the scale of the matrix. This means that if we desired to reduce our error by one order of magnitude, we would need to increase our matrix dimension, n, by one order of magnitude. This means that to decrease our error by one order of magnitude, we must use 10x the previously utilized memory.

Something else we are interested in is what the best implementation is for this algorithm. To start, we compare the two Thomas algorithm implementations to a Gaussian row elimination this is shown in Fig. 7. We can see that we are doing well if we do not use the Gaussian elimination. This is an exponential increase in the time required for error reduction.

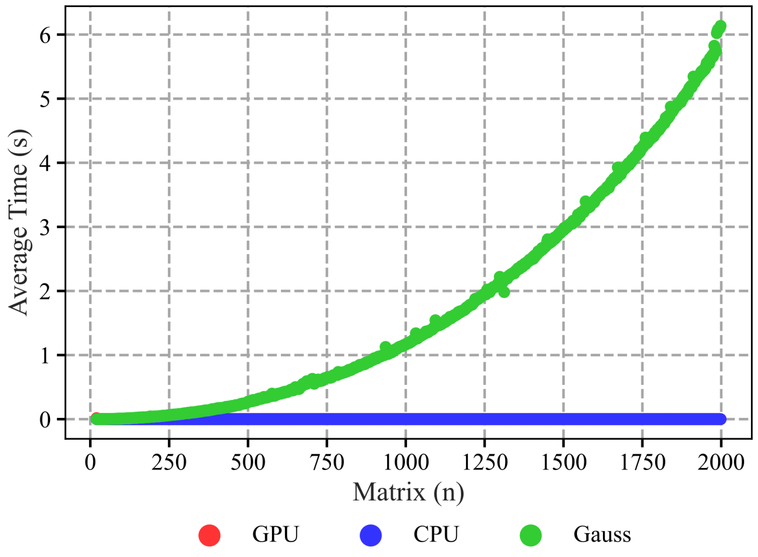

Figure 7: Comparison of the time required for a $n \times n$ matrix to be solved by 3 different implementations. GPU in red refers to using the Thomas algorithm with a GPU-oriented implementation, CPU refers to using the Thomas algorithm with a CPU-oriented implementation, and Gauss refers to solving the matrix with Gaussian row reduction.

Because the two Thomas implementations are so close Fig. 8, the CPU and GPU times were run with significantly larger matrices to highlight any differences.

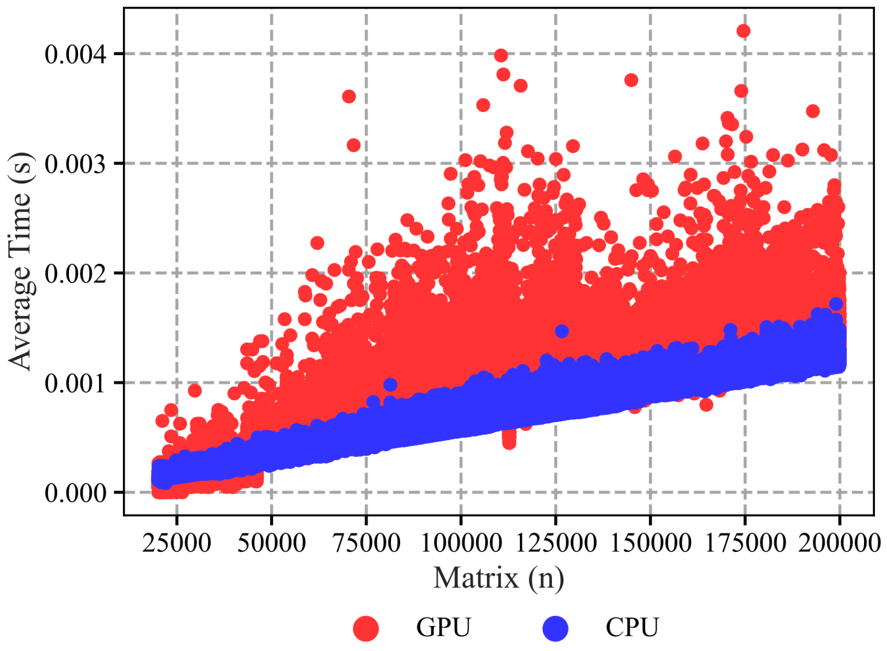

Figure 8: The Thomas algorithm implemented for GPU and CPU optimizations. We can see that the GPU may be ill-suited for this task as compared to the CPU.

We can clearly see that the CPU implementation of the Thomas algorithm is more consistent in time and smaller. We can explain this by looking at the method of the Thomas algorithm. With back substitution, each element depends on the previous, limiting us to a linear progression. This sort of algorithm is, therefore, not well set up for GPU optimization because no part can be finished without the last so the fact that the GPU is able to run in a comparable amount of time shows how much more effective the Thomas algorithm is than Gaussian row elimination for a tridiagonal matrix.

5 Conclusion

We have shown that when applied appropriately the Thomas algorithm allows for a significant performance boost in speed and memory. It has been shown that for cases such as the infinitesimal fluid flow further enhancements can be done such as solving for series of equations for α and β reducing the cost of forward substitution. With this work, we see a more linear order allowing for scaling to a higher amount of accuracy in our final solution. It is important to note these specific speed-ups are not general but not uncommon when calculating tridiagonal matrices.

References

[1] A. Povitsky. Efficient parallel-by-line methods in cfd. In D. KEYES, A. ECER, J. PERIAUX, N. SATOFUKA, and P. FOX, editors, *Parallel Computational Fluid Dynamics 1999*, pages 337–343. North-Holland, Amsterdam, 2000.

[2] Bastian E. Rapp. Chapter 32 - finite element method. In Bastian E. Rapp, editor, *Microfluidics: Modelling, Mechanics and Mathematics*, Micro and Nano Technologies, pages 655–678. Elsevier, Oxford, 2017.

[3] C.S. Yu and J. Berlamont. Modelling tidal flows in the northwest european continental shelf seas on parallel computers. In C.A. Lin, A. Ecer, J. Periaux, N. Satofuka, and P. Fox, editors, *Parallel Computational Fluid Dynamics 1998*, pages 525–532. North-Holland, Amsterdam, 1999.

[4] Yao Zhang, Jonathan Cohen, Andrew A. Davidson, and John D. Owens. Chapter 11 - a hybrid method for solving tridiagonal systems on the gpu. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 117–132. Morgan Kaufmann, Boston, 2012.

6 Thomas Algorithm Implementation

```python
import numpy as np

def thomasalgorithm(M, r):
    """
    Standard Thomas algorithm that is valid for use on the CPU
    This is version 2 which takes in a full matrix and has the
 ↪  goal of making
    a memory-efficient version.
    """
    # Getting iteration constant
    n = len(r)

    # Allocating memory for solution vector
    x = np.zeros(n)

    # Priming solution
    r[0] = r[0] / M[0, 0]
    M[0, 1] = M[0, 1] / M[0, 0]

    # Forward substitution on matrix
    for i in range(1, n - 1):
        r[i] = (r[i] - M[i, i - 1] * r[i - 1]) / (M[i, i] -
         ↪  M[i, i - 1] * M[i - 1, i])
        M[i, i + 1] = M[i, i + 1] / (M[i, i] - M[i, i - 1] *
         ↪  M[i - 1, i])

    # final step in forward substitutions
    r[n - 1] = (r[n - 1] - M[n - 1, n - 2] * r[n - 2]) / (
        M[n - 1, n - 1] - M[n - 1, n - 2] * M[n - 2, n - 1]
    )

    # First step in backward substitution
    x[n - 1] = r[n - 1]

    # Backward substitution to solve
    for i in reversed(range(n - 1)):
        x[i] = r[i] - M[i, i + 1] * x[i + 1]

    return x
```