COE322 Final Report: High Performance Linear Algebra

Nick Delurgio (npd429), Pavan Shukla (pas3488), Zoelle Wong (zfw65)

December 2021

Contents

1 Abstract

A high performance linear algebra solver with subroutines for addition, multiplication, recursive multiplication and other utility functions were created. The solver was then applied to a fluid mechanics application, where the solver's addition subroutine was compared to an algorithm with O(3) complexity for various matrix dimensions. The multiplication and recursive multiplication subroutines were also applied to a fluid mechanics application, and submatrix dimensions and recursive cutoff points were tested. It was found that the addition subroutine was faster than naive subroutine for adding matrices, however more testing on larger matrices are needed. Multiplication with a recursive algorithm was found to be faster than the naive algorithm for matrix sizes larger than 512x512 or 2048x2048, depending on the cutoff point. The largest savings in execution time was 76%, when the cutoff point was set to 16. However, this cutoff may be specific to the cache size of the computer that this computer used.

2 Introduction

As technological demands increase, so too does the need for high-performance linear algebra solvers. Applications related to linear algebra are found in a wide range of industries (semi-conductor, health, aerospace etc.), however the increasing demand for real-time results with larger, more complicated data sets has created a need for more computational efficient solvers. While custom application-specific hardware can achieve orders of magnitude in efficiency, the challenge of maintaining such efficiency to a broader class of operations remains. [1] These demands have led to research in parallelism, supercomputers, and surrogate modelling. However, effective programmers in high-performance computing (HPC) are rare because HPC code development depends on individuals with expert knowledge in HPC architecture and the application domain. [2] Such a dilemma is detrimental to many scientific fields, especially since current applications are becoming increasingly complex. Though much work has been done in this field to address these needs, this paper seeks to look at the fundamental principles for designing a linear algebra solver in C++.

This solver was then applied to solve a simple force analysis for a fully, developed three-dimensional Pouiseulle flow with a given velocity profile. This application involved two steps: (1) comparing the execution time for adding small matrices and (2) comparing the execution time between a recursive and conventional matrix multiplication algorithm. This analysis involved simplification of the Navier-Stokes equations (Figure 2) and the computation of discrete points along a pipe.

$$\begin{aligned}
x:\ & \rho\left(\partial_t u_x + u_x\,\partial_x u_x + u_y\,\partial_y u_x + u_z\,\partial_z u_x\right) \\
&\quad = -\partial_x p + \mu\left(\partial_x^2 u_x + \partial_y^2 u_x + \partial_z^2 u_x\right) + \frac{1}{3}\mu\,\partial_x\left(\partial_x u_x + \partial_y u_y + \partial_z u_z\right) + \rho g_x \\
y:\ & \rho\left(\partial_t u_y + u_x\partial_x u_y + u_y\partial_y u_y + u_z\partial_z u_y\right) \\
&\quad = -\partial_y p + \mu\left(\partial_x^2 u_y + \partial_y^2 u_y + \partial_z^2 u_y\right) + \frac{1}{3}\mu\,\partial_y\left(\partial_x u_x + \partial_y u_y + \partial_z u_z\right) + \rho g_y \\
z:\ & \rho\left(\partial_t u_z + u_x\partial_x u_z + u_y\partial_y u_z + u_z\partial_z u_z\right) \\
&\quad = -\partial_z p + \mu\left(\partial_x^2 u_z + \partial_y^2 u_z + \partial_z^2 u_z\right) + \frac{1}{3}\mu\,\partial_z\left(\partial_x u_x + \partial_y u_y + \partial_z u_z\right) + \rho g_z.
\end{aligned}$$

Figure 1: Navier-Stokes Equations in Cartesian coordinates, expanded vector form

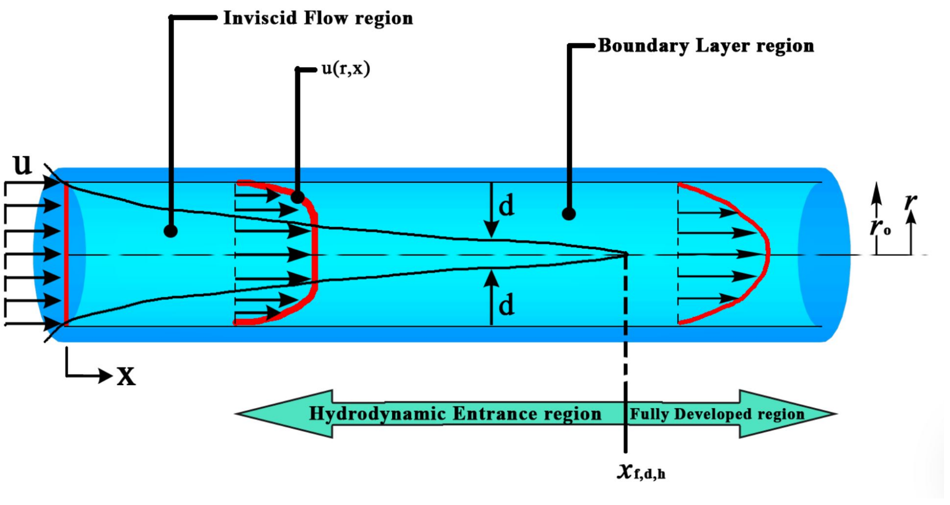


Figure 2: 2D schematic of Poiseuille Flow. U denotes the velocity profile, r variable radius from the pipe wall, r_0 the initial radius of the pipe, x the distance along the pipe, and x_f the critical location where the flow becomes fully developed. Note that the Hagan-Poiseuille Equations characterize pressure driven flows which are most commonly seen with flow inside pipes.

3 Methods

The architecture of the presented algebra relied heavily on lecture and class notes [3], [4]. The program and files related to this project can be found in the submitted code repository with Figures and 3 4 illustrating the overall project architecture.

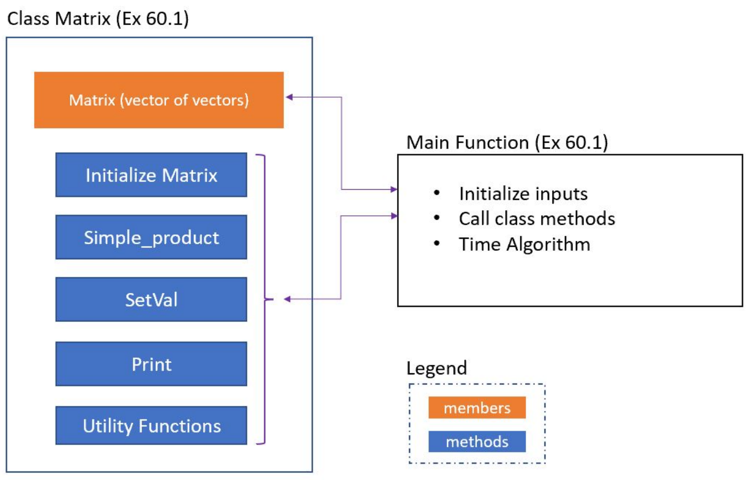


Figure 3: simple_matrix.cpp program design based on project criteria for Exercise 60.1

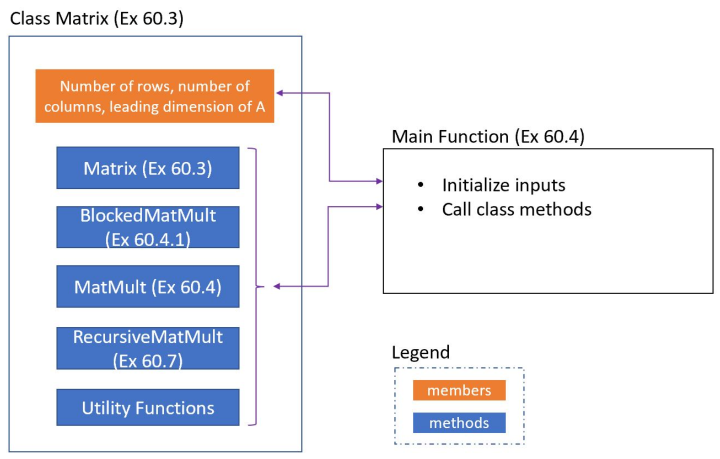


Figure 4: matrix.cpp program design based on project criteria for Exercises 60.2-9

4 Results

4.1 Exercises

The following section will answer in detail the exercise questions specified in the textbook.

4.1.1 Exercise 60.1

Suppose we have matrix A, $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}$ and matrix B, $\begin{bmatrix}1 & 2\\3 & 4\\5 & 6\end{bmatrix}$, their product P would be $\begin{bmatrix}22 & 28\\49 & 64\end{bmatrix}$. If one were to use the "inner-product based" method to compute this product, one would find that the computation takes an average of 10.6 microseconds when computed five times. If one were to change the "inner-product based" method such that the matrix was first multiplied by columns followed by rows, then this variant would take an average of 9.8 microseconds when computed five times. Both methods computed the correct product P however, the variant method was almost 1 microsecond faster than the "inner-product based" method. Therefore, one could argue that column by row matrix multiplication is faster than row by matrix multiplication.

4.1.2 Exercise 60.2

For a matrix A of dimensions MxN where the row index is given by i and the column index by j, there is a submatrix A′ with dimensions M′xN′, where M′ $\leq$ M and N′$\leq$ N. A one dimensional array X of length M′N′ is indexed by k and contains the entries of A′ arranged columnwise. That is,

$$X[k] = A'[i,j]$$

The leading dimension of A is called LDA, and LDA=M for columnwise array storage. If A′ is located at the top left of A so that it contains A[0,0], then the location in X of the entry A′$[i,j]$ is given by

$$k = i + M * j = i + LDA * j$$

For an arbitrarily located submatrix with contiguous rows and columns, and where the top left entry is defined as $A'[i,j] = A'[i_0, j_0]$, the corresponding index of X can be computed using a simple coordinate transformation:

$$i' = i - i_0 \quad \text{and} \quad j' = j - j_0$$

For example, in a case where M and N are both even integers, and A is partitioned into four submatrices of equal size with two submatrices on the top half and two submatrices on the bottom half, the coordinate transformation for the submatrix located on the bottom right is given by

$$i' = i - M/2 \quad \text{and} \quad j' = j - N/2$$

The index k is then given by

$$k = j' + M * i' = j' + LDA * i'$$

If X is arranged rowwise, then LDA=N, and the X index of the entry A$'[i,j]$ for a submatrix located at the top left of A is given by

$$k = j + N * i = j + LDA * i$$

And for the arbitrarily located submatrix,

$$k = i' + N * j' = i' + LDA * j'$$

4.1.3 Exercise 60.3-60.7

Please see "matrix.cpp" in the submitted code repository or in Appendix A.2. Note for Exercise 60.6, subroutines for adding and multiplying matrices for user inputted submatrices (see comments in main function). These subroutines will be faster than the routine in Exercise 60.1 because the algorithm in Exercise 60.1 uses an algorithm complexity with O(3) whereas the written subroutines uses an algorithm with O(2) complexity. Moreover, because the subroutines uses pointers to access data, less memory is allocated.

4.1.4 Exercise 60.8

Cache memory is a form of data storage that is much faster than the main memory. The cache memory is located much closer to the processor of a computer than the main memory, so there is a physically shorter distance that data must traverse in order to be accessed by the processor. However, cache memory is very small, and the data it stores rapidly changes as new processes are executed. When the processor makes a query for a piece of data, it first checks the cache to see if the data is already stored in the cache. If the data is found, then the processor reads the data from the cache. This case is known as a cache hit. If the data is not found, then the data is copied from the main memory to the cache memory, where it is read by the processor. This case is known as a cache miss. The cache miss rate is then defined as

$$\textit{cache miss rate} = \frac{\textit{total misses}}{\textit{total misses} + \textit{total hits}}$$

Thus, in order to optimize the performance of any task, it is necessary to minimize the cache miss rate. In other words, it is desired that the cache memory already contains the accessed data as often as possible. In this assignment, it is of interest to optimize the computation of a matrix-matrix product of the form

$$C = A \cdot B$$

From the textbook, code for a naive matrix-matrix product implementation can be written as

```
for (i=0; i<a.m; i++)
  for (j=0; j<b.n; j++)
    s = 0;
    for (k=0; k<a.n; k++)
      s += a[i,k] * b[k,j];
    c[i,j] = s;
```

Examining the code, it can be seen that the variable s is used to calculate C$[i, j]$ at each iteration of the inner loop. The variable s is calculated by iterating over the matrices A and B using a triple nested loop. For each iteration of the inner loop, new elements of both A and B are accessed and summed. That is, each computation of an element in C requires new data to be accessed. Therefore, unless the matrices being multiplied are very small, it is unlikely that values of A and B which can be reused are still stored in the cache memory.
The recursive method of matrix multiplication operates much differently. Before any scalar multiplication is executed, each matrix being multiplied is divided into smaller submatrices. If the submatrices are still larger than a predefined cutoff point, then the submatrices are divided into smaller submatrices. In this assignment, the recursive code divides matrices and submatrices on a 2x2 block form as shown:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

And each submatrix of C can be calculated as

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Each block matrix is further partitioned into smaller submatrices using the same process until both dimensions of a submatrix are less than four (M,N <4). When the cutoff point is reached, the naive algorithm, which employs scalar multiplication, is executed. Therefore, by the time any scalar multiplication is used, the scalars to be multiplied have already been stored in the cache memory as a submatrix. Thus, the recursive method will always lead to data reuse.

4.1.5 Exercise 60.9

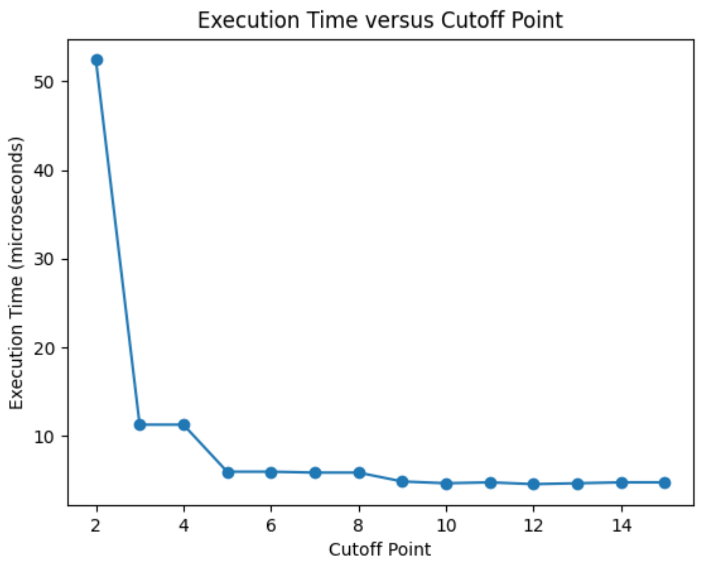
The purpose of the recursive method for matrix multiplication is to ensure that by the time any scalar products are executed, all the necessary scalar data are already stored in the cache memory. After a certain amount of recursion, the cache memory is large enough to contain all the scalar data being multiplied. At this point, continuing to recurse will not lend any of the previous benefits, as the original purpose of the recursion, to reduce the data to a size that can be stored in the cache memory, has already been fulfilled.
The cache sizes of the computer used in the experiments are as follows:

Level 1 (Registry): 768 KB

Level 2 (Cache Memory): 4 MB

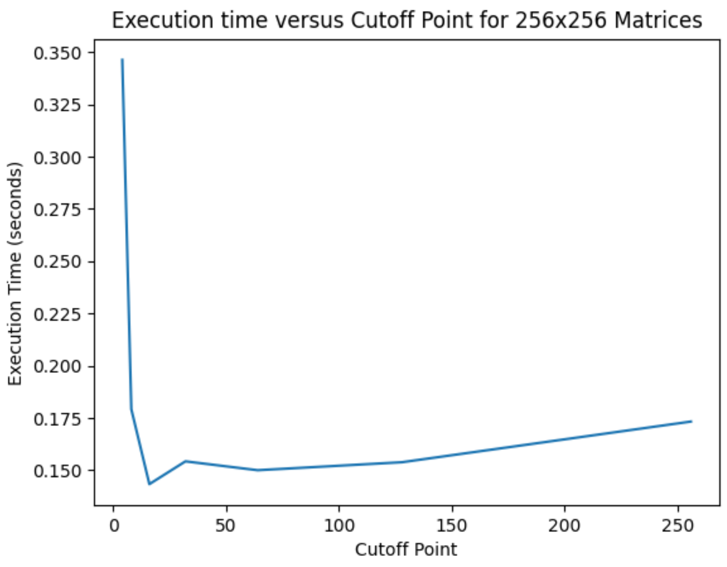
Level 3 (Main Memory) : 16 MB

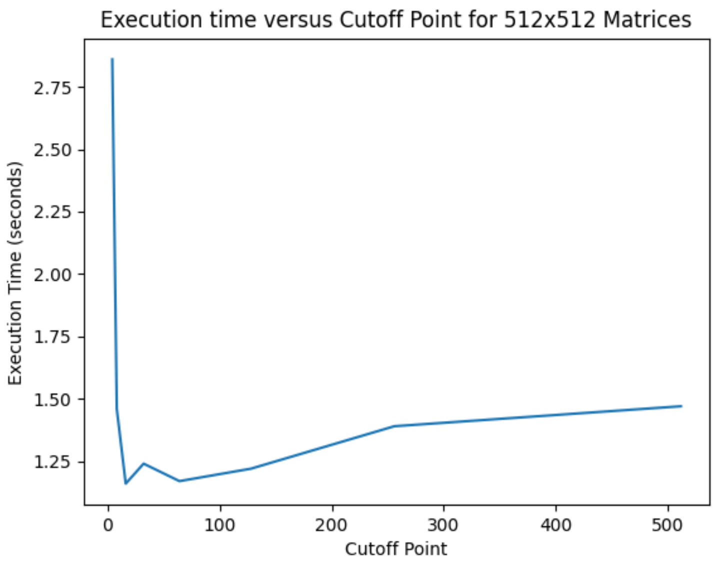
Various cutoff points for the recursive method were tested and compared. The execution times were plotted as a function of the cutoff point in the following figure.



As can be seen in the figure, execution time initially decreases with cut off point before leveling off at around a cutoff point of 5 (M,N < 5) and an execution time of approximately 4.8 microseconds. Though this behavior is unexpected, it is likely due to the size of the cache memory. The cache memory is likely large enough to contain all the data being multiplied before any partitioning has taken place. In other words, the point where recursion is no longer beneficial has been reached before the matrix is partitioned at all. In that case, any execution time beyond 4.8 microseconds would actually be spent on the recursive partitioning of the matrices, hence why the largest execution times are at the lowest cutoff points. However, it is expected that this would not be the case were the matrices larger. The matrices tested were relatively small, both having dimensions 8x8, so the efficiency of the recursive algorithm was not relevant. As a result, further testing was conducted with much larger matrices.
Though the naive and recursive methods are not being compared in this exercise, the rationale behind using larger matrices is to test higher cutoff points so that the cache memory will not be able to store all values of the initial matrices. The earlier matrices tested were small enough such that even a cutoff point containing the entire matrix, that is a case equivalent to naive matrix multiplication, could still be executed entirely using values initially stored in the cache memory. Since the scalar multiplications could be carried out completely in the cache memory, lower cutoff points and more recursion only increased run time, and higher cutoff points resulted in less recursion and lower run time. Using larger matrices and higher cutoff points, it will be possible to observe the performance of the recursive method at various cutoff points for cases when all data cannot be

stored in the cache memory. Then the benefits of the recursive method can be observed, and the optimal cutoff point for this particular processor can be found. The optimal cutoff corresponds to a point where the cache memory is full, so that no cache memory is unused and no excess data has to be accessed from the main memory. The discrete derivative of execution time with respect to cutoff will be positive to the left and right of the point, and the execution time will be at a minimum. The results for multiplication of two 256x256 matrices and two 512x512 matrices are shown below.





Cutoff points were incremented by powers of 2 starting with 4 (n=4,8,..,N) up to the actual dimension of the square matrices (n=N=M). These graphs also display a high execution time at the lowest cutoff points. However, it is clear the minima of both graphs occurs at a cutoff of 16. After 16, the graphs show distinctly positive trends in execution time. 16 may not be the actual optimal cutoff point, as the size of the x-axis exponentially increases. As a result, not all points were tested, however the optimal cutoff point almost certainly lies between 8 and 32.

5 Application: Fully Developed Poiseuille Flow

5.1 Why Poiseuille Flow?

Poiseuille flow is an important phenomena in fluid mechanics because its governing principles provide a model for understanding pipe flow. From heart valves to oil rigs, modelling the the characteristics of flow behavior under varying pipe conditions is pertinent for creating reliable and robust technology. While the fluid characteristics of Poiseuille flow are largely understood, its interactions amongst structural moving boundaries and dynamic acoustic fields are not. Solving the Navier-Stokes equations for these problems often do not contain an analytical solution, thus requiring solvers to rely on iterative, numerical methods. Hence, integrating high performance linear algebra with this application is relevant because problems related to Poiseuille flow are demanding longer simulation execution times. Therefore, this section aims to compare the execution times of our solver's addition and recursive multiplication subroutines with traditional textbook methods. While we hypothesize our solver will have a faster execution time, we seek to quantify the time savings. Such an analysis will be beneficial for future fluid analysis software architecture.

5.2 Testing Solver's Matrix Addition Subroutine

5.2.1 Problem Set-up

We will now apply our linear algebra solver to calculate pressure forces at a known distance from the wall of a pipe. Starting with the Navier-Stokes Equations (seen in Figure 2), we will assume that the flow is fully developed, steady, incompressible, and laminar. We will also ignore wall temperatures and surface roughness to simplify our analysis. Finally, we will look at two points on the flow: a point P1(x_1, y_1, z_1) on the pipe wall and a point P2(x_2, y_2, z_2) on the center line velocity. With these assumptions we can reduce our equations to a matrix representation, where every three rows in our matrix corresponds to the x,y, and z component:

$$\begin{bmatrix} \frac{\partial P}{\partial x_1} \\ \frac{\partial P}{\partial y_1} \\ \frac{\partial P}{\partial z_1} \\ \frac{\partial P}{\partial x_2} \\ \frac{\partial P}{\partial y_2} \\ \frac{\partial P}{\partial z_2} \end{bmatrix} = \begin{bmatrix} g_x \\ g_y \\ g_z \\ g_x \\ g_y \\ g_z \end{bmatrix} + 2\mu \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} & \frac{\partial v}{2\partial x} + \frac{\partial u}{2\partial y} \\ \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} & \frac{\partial v}{\partial y} & \frac{\partial w}{2\partial y} + \frac{\partial v}{2\partial z} \\ \frac{\partial v}{2\partial x} + \frac{\partial u}{2\partial y} & \frac{\partial w}{2\partial y} + \frac{\partial v}{2\partial z} & \frac{\partial w}{\partial z} \\ \frac{\partial u}{\partial x} & \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} & \frac{\partial v}{2\partial x} + \frac{\partial u}{2\partial y} \\ \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} & \frac{\partial v}{\partial y} & \frac{\partial w}{2\partial y} + \frac{\partial v}{2\partial z} \\ \frac{\partial v}{2\partial x} + \frac{\partial u}{2\partial y} & \frac{\partial w}{2\partial y} + \frac{\partial v}{2\partial z} & \frac{\partial w}{\partial z} \end{bmatrix} \tag{1}$$

In Equation 1, g_x, g_y and g_z represent the individual force components due to gravity, which we will consider to be approximately $9.81ms^2$ for this analysis. We will assume the working fluid to be water, resulting in μ to be approximately 10^-4. We will also assume an arbitrary velocity profile as the following:

$$U = \frac{1}{2\mu}(y^2 - hy)\hat{i} + (y^3 + hy)\hat{j} + y^3 + x\hat{k} \tag{2}$$

Where in Equation 2, h is the diameter of the pipe, x the location along the pipe, and μ the dynamic viscosity of the liquid. For now we will assume P1 to have coordinates $(100, 0, 1)$ and for P2 to have coordinates $(105, 0.5, 1)$. We will assume that the pipe has a length y of 200 m and diameter h of 1 m. Now, if we were to ignore P2 and assume the flow to be two dimensional at P1, then substituting Equation 2 into 1, will reduce to the following if we use a state-space representation:

$$\begin{bmatrix} \frac{\partial P}{\partial x_1} \\ \frac{\partial P}{\partial y_1} \end{bmatrix} = \begin{bmatrix} g_x \\ g_y \end{bmatrix} + 2\mu \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} \\ \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} & \frac{\partial v}{\partial y} \end{bmatrix} \tag{3}$$

$$\begin{bmatrix} \frac{\partial P}{\partial x_1} \\ \frac{\partial P}{\partial y_1} \end{bmatrix} = \begin{bmatrix} g_x & 0 \\ 0 & g_y \end{bmatrix} + \begin{bmatrix} 400\mu & 0 \\ 0 & \frac{7\mu}{2} \end{bmatrix} \tag{4}$$

Such a matrix representation is ideal because we now have a linear, matrix equation $C = A + B$, where the matrix C represents our forces due to pressure, A represents forces due to gravity, and B represents the forces due to shear. Note that B varies with y, the distance from the bottom wall to free stream conditions. Because Equation 1 is a 6x3 matrix, we can make assumptions about P1 and P2 as one, two, or three dimensional points to create three other combinations of submatrices. Hence, the tested cases and their corresponding assumptions can be seen below.

Case 2: Here we will be adding two 3x3 matrices if we neglect P2 and assume P1 is a three dimensional point. This assumption would be useful in the event that we would only need to consider the force at the center of the pipe, or where the fluid's velocity is greatest. Hence submatrices A and B are the following if simplified.

$$\begin{bmatrix} \frac{\partial P}{\partial x_1} \\ \frac{\partial P}{\partial y_1} \\ \frac{\partial P}{\partial z_1} \end{bmatrix} = \begin{bmatrix} g_x & 0 & 0 \\ 0 & g_y & 0 \\ 0 & 0 & gz \end{bmatrix} + \begin{bmatrix} 400\mu & 0 & 3001\mu \\ 0 & \frac{7\mu}{2} & \frac{3\mu}{4} \\ 3001 & \frac{3\mu}{4} & 0 \end{bmatrix} \tag{5}$$

Case 3: Here we will be adding two 4x3 matrices if we include P2 and assume P1 is a three dimensional point. In this case, we would only consider the x-component of P2 because we could make the simplifying assumption that its y- and z- component are 0 since P2 is located on the wall. Hence submatrices A and B are the following if simplified.

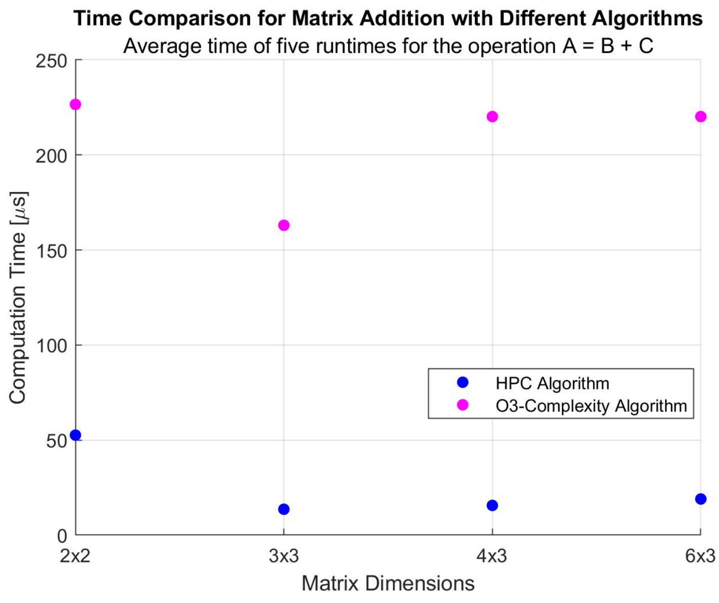
$$\begin{bmatrix} \frac{\partial P}{\partial x_1} \\ \frac{\partial P}{\partial y_1} \\ \frac{\partial P}{\partial z_1} \\ \frac{\partial P}{\partial x_2} \end{bmatrix} = \begin{bmatrix} g_x & 0 & 0 \\ 0 & g_y & 0 \\ 0 & 0 & g_z \\ g_x & 0 & 0 \end{bmatrix} + \begin{bmatrix} 400\mu & 0 & 3001\mu \\ 0 & \frac{7\mu}{2} & \frac{3\mu}{4} \\ 3001 & \frac{3\mu}{4} & 0 \\ 400\mu & \frac{-1}{2} & 3001\mu \end{bmatrix} \tag{6}$$

Case 4: Here we will be adding two 6x3 matrices if we include all three x-,y-, and z- components of P1 and P2. This assumption would be useful if one had to consider the pipe's surface roughness or if turbulent eddies are being formed at the boundary layer of the wall. Hence submatrices A and B are the following if simplified.

$$\begin{bmatrix} \frac{\partial P}{\partial x_1} \\ \frac{\partial P}{\partial y_1} \\ \frac{\partial P}{\partial z_1} \\ \frac{\partial P}{\partial x_2} \\ \frac{\partial P}{\partial y_2} \\ \frac{\partial P}{\partial z_2} \end{bmatrix} = \begin{bmatrix} g_x & 0 & 0 \\ 0 & g_y & 0 \\ 0 & 0 & g_z \\ g_x & 0 & 0 \\ 0 & g_y & 0 \\ 0 & 0 & g_z \end{bmatrix} + \begin{bmatrix} 400\mu & 0 & 3001\mu \\ 0 & \frac{7\mu}{2} & \frac{3\mu}{4} \\ 3001 & \frac{3\mu}{4} & 0 \\ 400\mu & \frac{-1}{2} & 3001\mu \\ \frac{-1}{2} & 2\mu & 0 \\ 3001 & 0 & 0 \end{bmatrix} \tag{7}$$

These matrix equations were then solved and timed using a timing function from the C++ chron library. See Appendix A.3 for details.

5.2.2 Results and Discussion



In Figure 5.2.2, we can see that employing a method that copies elements into a new array takes roughly 200 microseconds more time than corresponding a submatrix to a subarray. When corresponding a submatrix to a subarray, We see that summing 2 2x2 matrices took roughly 50 microseconds, but the subsequent matrices took roughly 20 microseconds. When using a triple nested for-loop or an algorithm with an O(3) complexity, we see that adding 2 3x3 matrices takes the least amount of time. However in both algorithms, we see a spike in computation time for computing the first case of submatrices followed by a dip for the second case, and an increase for the last 2 cases. While one would expect adding a 2x2 matrix to take the least amount time, the spike in computation time followed by a decrease may arise from testing dimension sizes that are too small as mentioned in Section 4.1.5.

5.3 Testing Solver's Matrix Multiplication

5.3.1 Problem Set-Up

Continuing with this problem set up, we will now compare the solver's base multiplication and recursive multiplication subroutine. In the context of Poiseuille Flow, we will examine a single point of a larger pipe with an arbitrary diameter of 50m with infinite length. Assuming a pipe with infinite length will ensure that the flow becomes fully developed. Because we are interested at the center line velocity we will look at the position $(x, y, z) = (100, 50, 50)$. However, we will no longer make the assumptions of irrotational, incompressible flow. We will also now account for thermal wall heating and surface roughness. Finally, we will also assume the flow is non-isotropic and that other impurities may be mixed with the fluid. As a result, we will now have to include more terms in our force equations to account for perturbations in the flow. Our problem will now look like the following:

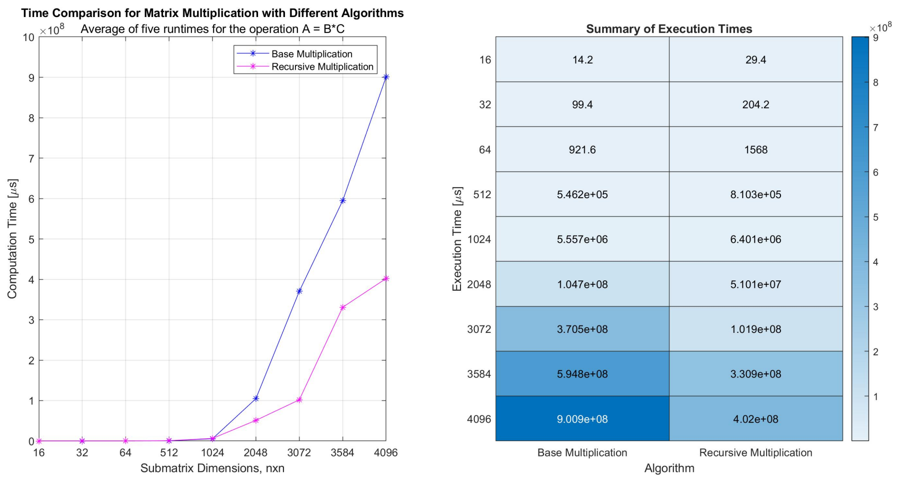
$$\begin{bmatrix} \frac{\partial P}{\partial x} & \cdots \\ \frac{\partial P}{\partial y} & \cdots \\ \frac{\partial P}{\partial z} & \cdots \\ \cdots & \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} & \frac{\partial v}{2\partial x} + \frac{\partial u}{2\partial y} & \cdots \\ \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} & \frac{\partial v}{\partial y} & \frac{\partial w}{2\partial y} + \frac{\partial v}{2\partial z} & \cdots \\ \frac{\partial v}{2\partial x} + \frac{\partial u}{2\partial y} & \frac{\partial w}{2\partial y} + \frac{\partial v}{2\partial z} & \frac{\partial w}{\partial z} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix} * 2\mu \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} & \frac{\partial v}{2\partial x} + \frac{\partial u}{2\partial y} & \cdots \\ \frac{\partial u}{2\partial y} + \frac{\partial v}{2\partial x} & \frac{\partial v}{\partial y} & \frac{\partial w}{2\partial y} + \frac{\partial v}{2\partial z} & \cdots \\ \frac{\partial v}{2\partial x} + \frac{\partial u}{2\partial y} & \frac{\partial w}{2\partial y} + \frac{\partial v}{2\partial z} & \frac{\partial w}{\partial z} & \cdots \\ \cdots & \cdots & \cdots & \end{bmatrix} \tag{8}$$

Here, Equation 8 takes the form $C = A * B$, where the matrix C is our force matrix, A our ”scaling matrix” that accounts for surface roughness and C, our shear matrix containing other perturbed fluid properties.

5.3.2 Methods

While an infinite amount of factors can affect a fluid particle's motion (which can lead to an infinite amount of terms in our matrix equations), we will constrain our problem to have an even number of row and columns inclusive of shear forces. This constraint is based on our solver, as it partitions submatrices into equal dimension block matrices. For our analysis, the elements in Matrix A were randomized numbers between 0 and 10^-5 created by a random generator function from the C++ random library. The elements in Matrix B were computed by first calculating the shear tensor followed by inserting randomized numbers between 0 and 10^-5. These perturbations will be very small since boundary layer effects are usually 1-3 orders of magnitude smaller than the fluid's dynamic viscosity. Matrix A and B with an LDA of 8192 were created using subroutines in a headerfile (See Code Appendix, A.4) and then inserted into the main Program (See Code Appendix, A.5). The dimension, n, for the submatrices were then varied on a 2^n scale and ran 5 times. The average of these execution times were computed from a function in a separate header file (See Code Appendix, A.6). The functions were also timed using functions from the C++ chron library.

5.3.3 Results and Discussion



	Base Multiplication	Recursive Multiplication
16	14.2	29.4
32	99.4	204.2
64	921.6	1568
512	5.462e+05	8.103e+05
1024	5.557e+06	6.401e+06
2048	1.047e+08	5.101e+07
3072	3.705e+08	1.019e+08
3584	5.948e+08	3.309e+08
4096	9.009e+08	4.02e+08

Figure 5: Results for creating a submatrix from a larger matrix with LDA of 8192. Recursive cutoff point is 4

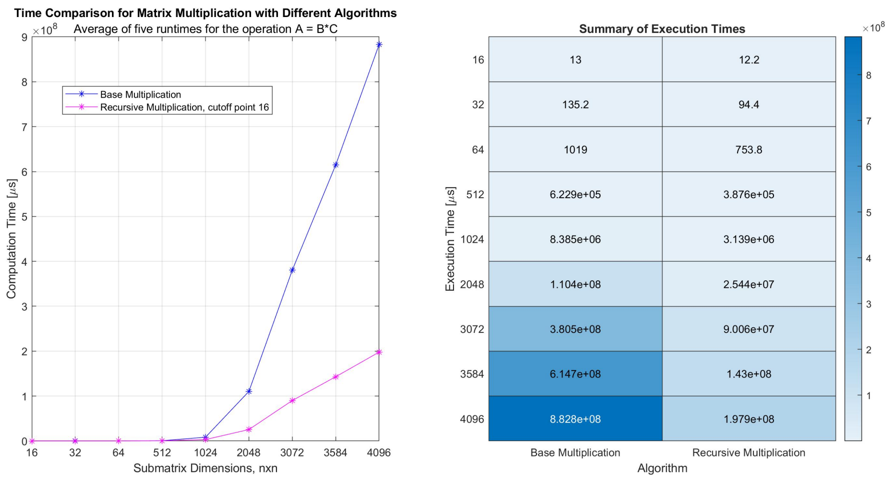


Figure 6: Results for creating a submatrix from a larger matrix with LDA of 8192. Recursive cutoff point is 16

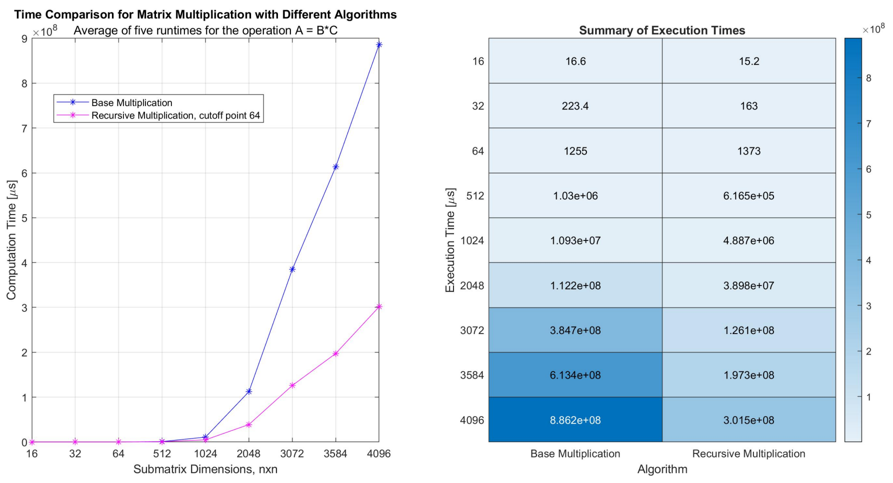
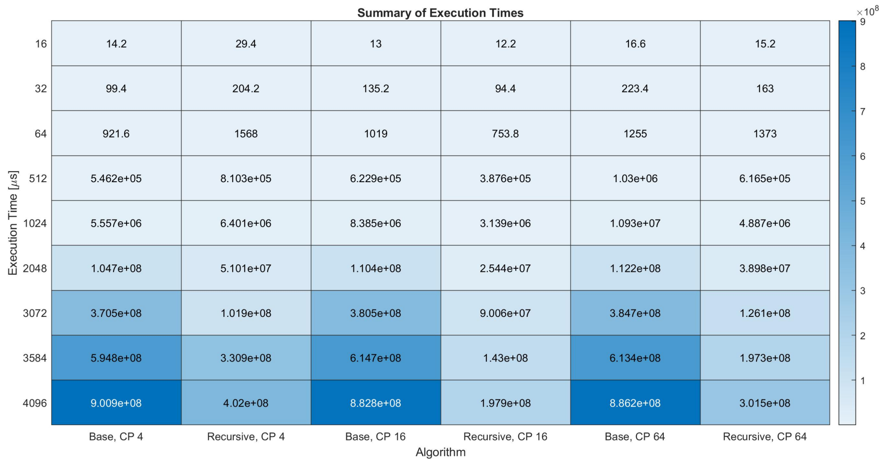


Figure 7: Results for creating a submatrix from a larger matrix with LDA of 8192. Recursive cutoff point is 64


Summary of Execution Times

Execution Time [μs]	Base, CP 4	Recursive, CP 4	Base, CP 16	Recursive, CP 16	Base, CP 64	Recursive, CP 64
16	14.2	29.4	13	12.2	16.6	15.2
32	99.4	204.2	135.2	94.4	223.4	163
64	921.6	1568	1019	753.8	1255	1373
512	5.462e+05	8.103e+05	6.229e+05	3.876e+05	1.03e+06	6.165e+05
1024	5.557e+06	6.401e+06	8.385e+06	3.139e+06	1.093e+07	4.887e+06
2048	1.047e+08	5.101e+07	1.104e+08	2.544e+07	1.122e+08	3.898e+07
3072	3.705e+08	1.019e+08	3.805e+08	9.006e+07	3.847e+08	1.261e+08
3584	5.948e+08	3.309e+08	6.147e+08	1.43e+08	6.134e+08	1.973e+08
4096	9.009e+08	4.02e+08	8.828e+08	1.979e+08	8.862e+08	3.015e+08

Figure 8: Table comparing execution times for different matrix dimensions and recursive cutoff points (abbreviated to 'CP')

Here we can see that the recursive algorithm gives a performance improvement because the base multiplication algorithm has a O(3) complexity whereas the recursive algorithm has a complexity smaller than O(3). Recursive algorithm requires more memory than the basic multiplication algorithm, because the matrices must have their dimension expanded to the next power of 2. Therefore for smaller matrices, the recursive algorithm will require more memory than the naive algorithm and require more execution time. Hence at a certain threshold, the costs of adding smaller matrix blocks will outweigh the savings of multiplying a larger block.

Varying the recursion cutoff point to a higher value also resulted in the recursive algorithm to be more efficient for smaller matrices. At a cutoff point of 4, the recursive algorithm became about 51% more efficient than the base multiplication algorithm for a 2048x2048 matrix. However at a cutoff point of 16, the recursive algorithm became roughly 76% more efficient for a 512x512 matrix. At a cutoff point at 64, the recursive algorithm became about 67% more efficient for a 512x512 matrix. This result implies that the greatest savings in execution time would be at a cutoff point of roughly 16, or a number slightly greater than 16. Such a cutoff point is ideal because copying a matrix size of 16 was faster than partitioning a matrix to sizes less than 16. However this cutoff point may be specific to the cache size of the computer (in this case, ISP was used on TACC's server). Because cache size varies with processor, these time savings are also dependent on the computer this experiment used. While the recursive algorithm demonstrates an increased performance, optimization of the cutoff point would depend on the computer's cache size. Though one could alter the program to be cache oblivious, large time savings would be seen in the naive algorithm but not necessarily in the recursive algorithm. [4]

6 Conclusion

In conclusion, a high performance linear algebra solver with subroutines for addition, multiplication, recursive multiplication and other utility functions were created. Analysis of the solver found that multiplication of small matrices are more unlikely to be stored in the cache memory. However the solver's recursive multiplication subroutine will always reuse cache memory because matrices are being continuously divided into submatrices to perform multiplication.

Moreover, the efficiency of this solver's addition, matrix, and recursive multiplication subroutine was applied in a Poiseuille Flow application for an arbitrary velocity profile and known parameter space. The addition subroutine was faster than naive subroutine for adding matrices, however more testing on larger matrices are needed. Recursive multiplication routine was also faster than the base multiplication subroutine. The switch off point, where the recursive multiplication routine became faster than the base multiplication subroutine, was greatest when the the cutoff point was increased to 16. Raising the cutoff point to 64 did not change the switch off point nor reduce execution time. However these cutoff points are cache-specific to the computer this experiment used.

6.1 Future Work

Improving this solver would involve designing subroutines for other matrix operations (inverse, dot and cross products etc.) and parallelizing parts of the code to increase efficiency. Profiling the program would also provide insight for areas of decreasing CPU usage and determining algorithms that spend the most amount of computational time. Moreover, comparing the solver's speed to other other programming languages (MATLAB, Python, Julia etc.) would be insightful because other languages depend on alternative libraries and back-end processes to perform matrix operations speedily. Future experiments should also test the addition subroutine for larger matrices on an order of magnitude that ranges from $10 - 10^4$. Finally, future experiments should test the matrix subroutines more than 5 times since a majority of the matrix elements were computed randomly. Ultimately, these improvements would grant more flexibility to solve other problems involving more complex matrix operations.

References

[1] D. E. Knuth, "Asap 2011 - 22nd ieee international conference on application-specific systems, architectures and processors," *ASAP 2011 - 22nd IEEE International Conference.*

[2] J. Carver, S. Asgari, V. Basili, L. Hochstein, J. K. Hollingsworth, and M. Zelkowitz, "Studying code development for high performance computing: the hpcs program," in *In Proceedings of the International Workshop on Software Engineering for High Performance Computing Systems Applications (SE-HPCS '04*, 2004.

[3] V. Eijkhout, *Introduction to Scientific Programming in C++17/Fortran2003.* 2021.

[4] R. v. d. G. Victor Eijkhout, Edmond Chow, *Introduction to High Performance Scientific Computing.* 2020.

A Code Appendix

The following sections include the program files that were primarily used for the analysis in this paper. To see all the source files, however, please refer to the submitted code repository.

A.1 " simple_matrix.cpp", Program used for Exercise 60.1

```
#include <iostream>
#include <vector>
#include <chrono>
using namespace std::chrono;

using namespace std;

class matrix {
    private:
    vector<vector<double>> A;
    public:
     // left public for ease of computations

    //CONSTRUCTOR
    matrix(vector<vector<double>> matrix) {
        A = matrix;
    }
    matrix(int rows,int cols,double num) { //initializes matrix to num with m rows
    and n cols
        vector<double> nums;
        for(int i = 0; i < cols; i++) {
            nums.push_back(num);
        }
        for(int i = 0; i < rows; i++) {
            A.push_back(nums);
        }
    }

    int nrows() {
        return A.size();
    }

    int ncols() {
        return A[0].size();
    }

    double getVal(int row,int col) {
        return A[row][col];
    }

    void setVal(int row, int col, double num) {
        A[row][col] = num;
    }

    void print() {
        for(int i = 0; i < nrows(); i++) {
            for(int j = 0; j < ncols(); j++) {
                cout << getVal(i,j) << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

    matrix simple_product(matrix B) { // Simple, computationally expensive way to multiple matrices. A.simple_product(B) = A*B. Solves Exercise 60.1.
        matrix C(nrows(), B.ncols(), 0); //Allocates size of solution matrix C
        for(int i = 0; i < nrows(); i++) {
            for(int j = 0; j < B.ncols(); j++) {
                double sum = 0;
                for (int k = 0; k < ncols(); k++) {
                    sum += getVal(i,k) * B.getVal(k,j);
                }
                C.setVal(i,j,sum);
            }
        }
        return C;
    }

    // Exercise 60.1
    matrix simple_product_variant(matrix B) { // Simple, computationally expensive way to multiple matrices. A.simple_product(B) = A*B. Solves Exercise 60.1.
        matrix C(nrows(), B.ncols(), 0); //Allocates size of solution matrix C
        for(int i = 0; i < B.ncols(); i++) {
            for(int j = 0; j < B.nrows(); j++) {
                double sum = 0;
                for (int k = 0; k < B.nrows(); k++) {
                    sum += getVal(i,k) * B.getVal(k,j);
                }
                C.setVal(i,j,sum);
            }
        }
        return C;
    }
};

int main() {
    matrix A({{1,2,3},{4,5,6}});
    A.print();
    matrix B({{1,2},{3,4},{5,6}});
    B.print();
    // time product function
    auto start = high_resolution_clock::now();
        matrix C = A.simple_product_variant(B); // call function
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    cout << "Time taken by function: "
         << duration.count() << " microseconds" << endl;

    C.print();
}
```

A.2 ”matrix.cpp”, Program used for Exercises 60.1-60.9

```
//Compile line: icpc -I${TACC_GSL_INC} final/matrix.cpp
#include <iostream>
#include <vector>
#include <array>
//<<<<<<< HEAD
//#include <..\GSL-main\include\gsl\span>
//#include "GSL-main"///gsl-lite.hpp"
//=======
#include "gsl/gsl-lite.hpp"
//>>>>>>> 36156feaf67fd9fab30f02627e9a3a201deb7161

#define INDEX(i,j,lda) (j)*(lda) + (i) //Computationally efficient method of indexing
     through data as it does not have the overhead for calling a function.


using namespace std;
using gsl::span;



class Matrix {

    private:

    int m,n,lda;
    span<double> data;

    public:

    //Getters
    int getrows() { return m; }
    int getcols() { return n; }
    int getlda() { return lda; }

    //Constructors (ex. 60.3)
    Matrix(int m, int lda, int n, double *data) {
        if (lda < m) {
            cout << "Error creating matrix: LDA < m" << endl;
            throw(1);
        }

        this->m = m;
        this->lda = lda;
        this->n = n;
        this->data = span<double> (data,lda*n); //Use a span as to not allocate extra
     memory

    }

    //return element function (ex. 60.3)
    double& at(int i, int j) { //Using & allows you to change the data at this
    element, not just access it.
        if(i >= m || j >= n || i < 0 || j < 0) { //Ensure index is in bounds before
    returning
            cout << "Error: index out of bounds" << endl;
            throw(1);
        }
        return data[j*lda + i];

    }
    auto get_double_data() { //Returns a pointer torwards the entire data set,
    reducing overhead
        double *adata;
        adata = data.data();
        return adata;
    }
    //Addition method (ex. 60.4)
    void addMatrices(Matrix& B, Matrix& out) {
        if (this->getrows() != B.getrows() || this->getcols() != B.getcols()) { // Ensure the matrix addition is legal
            cout << "Error using addMatrices: Matrices do no have the same dimensions" << endl;
            throw(1);
        }

        auto adata = this->get_double_data();
        auto bdata = B.get_double_data();
        auto cdata = out.get_double_data();
        for(int j = 0; j < this->getcols(); j++) {
            for(int i = 0; i < this->getrows(); i++) {
                #ifdef DEBUG
                    cdata[INDEX(j,this->getrows(),i)] = this->at(i,j) + B.at(i,j); // Slightly more overhead, but good for debugging
                #else
                    cdata[INDEX(j,this->getrows(),i)] = adata[INDEX(j,this->getlda(),i)] + bdata[INDEX(j,B.getlda(),i)]; //Uses the INDEX definition, which is predetermined.
                #endif
            }
        }
    }


    //Submatrices support (ex. 60.6)
    Matrix Left(int j) {
        return Matrix(this->m,this->lda,j,this->get_double_data());
    }
    Matrix Right(int j) {
        return Matrix(this->m,this->lda,n-j,this->get_double_data() + lda*j);
    }
    Matrix Top(int i) {
        return Matrix(i,this->lda,this->n,this->get_double_data());
    }
    Matrix Bot(int i) {
        return Matrix(m-i,this->lda,this->n,this->get_double_data() + i);
    }

    //Multiplication functions
    void MatMult(Matrix& other, Matrix& out) { //Basic multiplication function with O(n^3)
        auto adata = this->get_double_data();
        auto bdata = other.get_double_data();
        auto cdata = out.get_double_data();
        for(int i = 0; i < this->m; i++) {
            for(int j = 0; j < other.getcols(); j++) {
                for (int k = 0; k < this->n; k++) {
                    #ifdef DEBUG
                        out.at(i,j) += this->at(i,k) * other.at(k,j); //slower
                    #else
                        cdata[INDEX(i,j,out.getlda())] += adata[INDEX(i,k,this->lda)] * bdata[INDEX(k,j,other.getlda())]; //faster
                    #endif
                }
            }
        }
        return;
    }


    void BlockedMatMult(Matrix& other, Matrix& out) { //Definition of the blocked method, which splits up the matrices into groups of 4 for the multiplication process

        //None of these matrices need to allocate new memory; all point torwards old memory
        Matrix atl = this->Left(this->getcols()/2).Top(this->getrows()/2);
        Matrix atr = this->Right(this->getcols()/2).Top(this->getrows()/2);
        Matrix abl = this->Left(this->getcols()/2).Bot(this->getrows()/2);
        Matrix abr = this->Right(this->getcols()/2).Bot(this->getrows()/2);

        Matrix btl = other.Left(other.getcols()/2).Top(other.getrows()/2);
        Matrix btr = other.Right(other.getcols()/2).Top(other.getrows()/2);
        Matrix bbl = other.Left(other.getcols()/2).Bot(other.getrows()/2);
        Matrix bbr = other.Right(other.getcols()/2).Bot(other.getrows()/2);

        Matrix otl = out.Left(out.getcols()/2).Top(out.getrows()/2);
        Matrix otr = out.Right(out.getcols()/2).Top(out.getrows()/2);
        Matrix obl = out.Left(out.getcols()/2).Bot(out.getrows()/2);
        Matrix obr = out.Right(out.getcols()/2).Bot(out.getrows()/2);

        atl.MatMult(btl,otl);
        atr.MatMult(bbl,otl);

        atl.MatMult(btr,otr);
        atr.MatMult(bbr,otr);

        abl.MatMult(btl,obl);
        abr.MatMult(bbl,obl);

        abl.MatMult(btr,obr);
        abr.MatMult(bbr,obr);

    }

    //Ex 60.7
    void RecursiveMatMult(Matrix& other, Matrix& out) { //Same as BlockedMatMult, but with recursion to keep breaking down the matrix sizes until reaching sufficient detail

        if(this->getrows() < 4 && this->getcols() < 4 && other.getrows() < 4 && other.getcols() < 4) { //Stops the recursive process once the matrix size is < 4
            this->MatMult(other,out);
        }
        else {
            Matrix atl = this->Left(this->getcols()/2).Top(this->getrows()/2);
            Matrix atr = this->Right(this->getcols()/2).Top(this->getrows()/2);
            Matrix abl = this->Left(this->getcols()/2).Bot(this->getrows()/2);
            Matrix abr = this->Right(this->getcols()/2).Bot(this->getrows()/2);

            Matrix btl = other.Left(other.getcols()/2).Top(other.getrows()/2);
            Matrix btr = other.Right(other.getcols()/2).Top(other.getrows()/2);
            Matrix bbl = other.Left(other.getcols()/2).Bot(other.getrows()/2);
            Matrix bbr = other.Right(other.getcols()/2).Bot(other.getrows()/2);

            Matrix otl = out.Left(out.getcols()/2).Top(out.getrows()/2);
            Matrix otr = out.Right(out.getcols()/2).Top(out.getrows()/2);
            Matrix obl = out.Left(out.getcols()/2).Bot(out.getrows()/2);
            Matrix obr = out.Right(out.getcols()/2).Bot(out.getrows()/2);

            atl.RecursiveMatMult(btl,otl);
            atr.RecursiveMatMult(bbl,otl);

            atl.RecursiveMatMult(btr,otr);
            atr.RecursiveMatMult(bbr,otr);

            abl.RecursiveMatMult(btl,obl);
            abr.RecursiveMatMult(bbl,obl);

            abl.RecursiveMatMult(btr,obr);
            abr.RecursiveMatMult(bbr,obr);
        }
    }

    //for testing purposes
    void print() {
        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                cout << this->at(i,j) << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

    void printdata() { //prints the data of the full matrix, not just the submatrix.
    Useful for testing
        for(int i = 0; i < lda*n; i++) {
            cout << data[i] << " ";
        }
        cout << endl << endl;
    }

};



int main() {
    int m = 2;
    int lda = 3;
    int n = 2;
    vector<double> data1 = {1,3,5,2,4,6};
    vector<double> data2 = {1,2,3,4};
    vector<double> data3 = {2,2,3,4,5,6,7,8,9,10,11,12};
    vector<double> data4 = {2,2,3,4,5,6,7,8,9};
    Matrix m1(3,4,3,data3.data());
    Matrix m2(3,3,3,data4.data());
    Matrix m3(3,3,3,data4.data());
    m1.print();
    m2.print();
    m1.addMatrices(m2,m3);
    m3.print();

    vector<double> data5 = {1,2,3,4,5,6,7,8,9,10,10,12,13,14,15,16};
    Matrix m4(3,4,4,data5.data());
    //m4.print();
    Matrix l1 = m4.Right(2);
    //l1.print();

    m1.print();
    m2.print();
    Matrix m5(3,3,3,vector<double>(9,0).data());
    m1.MatMult(m2, m5);
    m5.print();

    vector<double> data6 = {1,2,3,4,5,6,7,8,9};
    vector<double> data7 = {2,3,4};
    Matrix m6(3,3,3,data6.data());
    Matrix m7(3,3,1,data7.data());
    Matrix m8(3,3,1,vector<double>(3,0).data());
    m6.MatMult(m7,m8);
    m6.print();
    m7.print();
    m8.print();

    vector<double> r4c5 = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
    vector<double> r5c4 = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
    vector<double> r4c4 = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

    Matrix m20a(4,4,5,r4c5.data());
    Matrix m20b(5,5,4,r4c5.data());
    //Matrix m20c(4,4,4,vector<double>(16,0).data());
    Matrix m20c(4,4,4,r4c4.data());
    cout << "M20A" << endl;
    m20a.print();
    cout << "M20B" << endl;
    m20b.print();
    m20a.BlockedMatMult(m20b,m20c);

    vector<double> r8c8 =
    {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,

    vector<double> r8c82 =
    {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,

    vector<double> r8c83(64,0);

    Matrix m24a(8,8,8,r8c8.data());
    Matrix m24b(8,8,8,r8c82.data());
    Matrix m24c(8,8,8,r8c83.data());
    m24a.print();
    m24b.print();
    m24a.RecursiveMatMult(m24b,m24c);

    m24c.print();

    return 0;
}
```

A.3 "vib2.cpp", Program used for the application involving Poiseuille flow.

```
//Compile line: icpc -I${TACC_GSL_INC} final/vib2.cpp
#include <iostream>
#include <vector>
#include <array>
#include "gsl/gsl-lite.hpp"

#include <chrono>
```

```
using namespace std::chrono;

#define INDEX(i,j,lda) (j)*(lda) + (i)


using namespace std;
using gsl::span;



class Matrix {

    private:

    int m,n,lda;
    span<double> data;

    public:

    //Getters
    int getrows() { return m; }
    int getcols() { return n; }
    int getlda() { return lda; }

    //Constructors (ex. 60.3)
    Matrix(int m, int lda, int n, double *data) {
        if (lda < m) {
            cout << "Error creating matrix: LDA < m" << endl;
            throw(1);
        }

        this->m = m;
        this->lda = lda;
        this->n = n;
        this->data = span<double> (data,lda*n);

    }

    //return element function (ex. 60.3)
    double& at(int i, int j) {
        if(i >= m || j >= n || i < 0 || j < 0) {
            cout << "Error: index out of bounds" << endl;
            throw(1);
        }
        return data[j*lda + i];

    }
    auto get_double_data() {
        double *adata;
        adata = data.data();
        return adata;
    }
    //Addition method (ex. 60.4)
    void addMatrices(Matrix& B, Matrix& out) {
        if (this->getrows() != B.getrows() || this->getcols() != B.getcols()) {
            cout << "Error using addMatrices: Matrices do no have the same dimensions
    " << endl;
            throw(1);
        }

        auto adata = this->get_double_data();
        auto bdata = B.get_double_data();
        auto cdata = out.get_double_data();
        for(int j = 0; j < this->getcols(); j++) {
            for(int i = 0; i < this->getrows(); i++) {
                #ifdef DEBUG
                    cdata[INDEX(j,this->getrows(),i)] = this->at(i,j) + B.at(i,j);
                #else
                    cdata[INDEX(j,this->getrows(),i)] = adata[INDEX(j,this->getlda(),
    i)] + bdata[INDEX(j,B.getlda(),i)];
                #endif
            }
        }
    }


    //Submatrices support (ex. 60.6)
    Matrix Left(int j) {
        return Matrix(this->m,this->lda,j,this->get_double_data());
    }
    Matrix Right(int j) {
        return Matrix(this->m,this->lda,n-j,this->get_double_data() + lda*j);
    }
    Matrix Top(int i) {
        return Matrix(i,this->lda,this->n,this->get_double_data());
    }
    Matrix Bot(int i) {
        return Matrix(m-i,this->lda,this->n,this->get_double_data() + i);
    }

    //Multiplication functions
    void MatMult(Matrix& other, Matrix& out) {
        auto adata = this->get_double_data();
        auto bdata = other.get_double_data();
        auto cdata = out.get_double_data();
        for(int i = 0; i < this->m; i++) {
            for(int j = 0; j < other.getcols(); j++) {
                for (int k = 0; k < this->n; k++) {
                    #ifdef DEBUG
                        out.at(i,j) += this->at(i,k) * other.at(k,j);
                    #else
                        cdata[INDEX(i,j,out.getlda())] += adata[INDEX(i,k,this->lda)]
     * bdata[INDEX(k,j,other.getlda())];
                    #endif
                }
            }
        }
        return;
    }


    void BlockedMatMult(Matrix& other, Matrix& out) {

        Matrix atl = this->Left(this->getcols()/2).Top(this->getrows()/2);
        Matrix atr = this->Right(this->getcols()/2).Top(this->getrows()/2);
        Matrix abl = this->Left(this->getcols()/2).Bot(this->getrows()/2);
        Matrix abr = this->Right(this->getcols()/2).Bot(this->getrows()/2);

        Matrix btl = other.Left(other.getcols()/2).Top(other.getrows()/2);
        Matrix btr = other.Right(other.getcols()/2).Top(other.getrows()/2);
        Matrix bbl = other.Left(other.getcols()/2).Bot(other.getrows()/2);
        Matrix bbr = other.Right(other.getcols()/2).Bot(other.getrows()/2);

        Matrix otl = out.Left(out.getcols()/2).Top(out.getrows()/2);
        Matrix otr = out.Right(out.getcols()/2).Top(out.getrows()/2);
        Matrix obl = out.Left(out.getcols()/2).Bot(out.getrows()/2);
        Matrix obr = out.Right(out.getcols()/2).Bot(out.getrows()/2);

        atl.MatMult(btl,otl);
        atr.MatMult(bbl,otl);

        atl.MatMult(btr,otr);
        atr.MatMult(bbr,otr);

        abl.MatMult(btl,obl);
        abr.MatMult(bbl,obl);

        abl.MatMult(btr,obr);
        abr.MatMult(bbr,obr);

    }

    //Ex 60.7
    void RecursiveMatMult(Matrix& other, Matrix& out) {

        if(this->getrows() < 4 && this->getcols() < 4 && other.getrows() < 4 && other
    .getcols() < 4) {
            this->MatMult(other,out);
        }
        else {
            Matrix atl = this->Left(this->getcols()/2).Top(this->getrows()/2);
            Matrix atr = this->Right(this->getcols()/2).Top(this->getrows()/2);
            Matrix abl = this->Left(this->getcols()/2).Bot(this->getrows()/2);
            Matrix abr = this->Right(this->getcols()/2).Bot(this->getrows()/2);

            Matrix btl = other.Left(other.getcols()/2).Top(other.getrows()/2);
            Matrix btr = other.Right(other.getcols()/2).Top(other.getrows()/2);
            Matrix bbl = other.Left(other.getcols()/2).Bot(other.getrows()/2);
            Matrix bbr = other.Right(other.getcols()/2).Bot(other.getrows()/2);

            Matrix otl = out.Left(out.getcols()/2).Top(out.getrows()/2);
            Matrix otr = out.Right(out.getcols()/2).Top(out.getrows()/2);
            Matrix obl = out.Left(out.getcols()/2).Bot(out.getrows()/2);
            Matrix obr = out.Right(out.getcols()/2).Bot(out.getrows()/2);

            atl.RecursiveMatMult(btl,otl);
            atr.RecursiveMatMult(bbl,otl);

            atl.RecursiveMatMult(btr,otr);
            atr.RecursiveMatMult(bbr,otr);

            abl.RecursiveMatMult(btl,obl);
            abr.RecursiveMatMult(bbl,obl);

            abl.RecursiveMatMult(btr,obr);
            abr.RecursiveMatMult(bbr,obr);
        }
    }

    //for testing purposes
    void print() {
        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                cout << this->at(i,j) << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

    void printdata() {
        for(int i = 0; i < lda*n; i++) {
            cout << data[i] << " ";
        }
        cout << endl << endl;
    }

};



int main() {

    double g = 9.81; // gravity, [kg/m*s^2]
    double mu = 0.0001; // dynamic viscosity of water

    vector<double> data1 = {400*mu,0,3001*mu, 400*mu,-0.5,0,3001*mu,
                            0,3.5*mu, .75*mu,-0.5,2*mu,0,
                            3001*mu,.75*mu,0,3001*mu, 0, 0};//Shear Matrix, water.
    vector<double> data2 = {g,0,0,g,0,0,
                            0,g,0,0,g,0,
                            0,0,g,0,0,g};// Body Forces Matrix
    vector<double> data3 = {g,0,0,0,0,0,
                            0,0,g,0,0,0,
                            0,0,0,0,0,g}; // Dummy Matrix

     // Test adding 2 by 2 matrices
    cout << "Computing Matrix Product. Result is: " << endl;
    auto start = high_resolution_clock::now();    // time product function
        Matrix m1(2,6,2,data1.data());
        Matrix m2(2,6,2,data2.data());
        Matrix m3(2,6,2,data3.data());
        m3.addMatrices(m1,m2);
        m2.print();
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    cout << "Time taken by function: "
    << duration.count() << " microseconds" << endl;

    // Test adding 3 by 3 matrices
    cout << "Computing Matrix Product. Result is: " << endl;
        auto start1 = high_resolution_clock::now();    // time product function
        Matrix m4(3,6,3,data1.data());
        Matrix m5(3,6,3,data2.data());
        Matrix m6(3,6,3,data3.data());
        m6.addMatrices(m4,m5);
        m5.print();
    auto stop1 = high_resolution_clock::now();
    auto duration1 = duration_cast<microseconds>(stop1 - start1);
    cout << "Time taken by function: "
    << duration1.count() << " microseconds" << endl;

    // Test adding 4 by 3 matrices
    cout << "Computing Matrix Product. Result is: " << endl;
    auto start2 = high_resolution_clock::now();    // time product function
        Matrix m7(4,6,3,data1.data());
        Matrix m8(4,6,3,data2.data());
        Matrix m9(4,6,3,data3.data());
        m9.addMatrices(m7, m8);
        m8.print();
    auto stop2 = high_resolution_clock::now();
    auto duration2 = duration_cast<microseconds>(stop2 - start2);
    cout << "Time taken by function: "
    << duration2.count() << " microseconds" << endl;

    // Test adding 6 by 3 matrices
    cout << "Computing Matrix Product. Result is: " << endl;
    auto start3 = high_resolution_clock::now();    // time product function
        Matrix m10(5,6,3,data1.data());
        Matrix m11(5,6,3,data2.data());
        Matrix m12(5,6,3,data3.data());
        m12.addMatrices(m10, m11);
        m11.print();
    auto stop3 = high_resolution_clock::now();
    auto duration3 = duration_cast<microseconds>(stop3 - start3);
    cout << "Time taken by function: "
    << duration3.count() << " microseconds" << endl;

    // Tests varying dynamic viscosity at different orders of magnitude
    /*
    vector<double> data1 = {4,0,0, 0,0.0035,0,0,0};//Shear Matrix, water.
    vector<double> data2 = {g,0,0,0,g,0,0,0,g};// Body Forces Matrix
    vector<double> data3 = {g,0,0,0,0,0,0,0,g}; // Dummy Matrix
    Matrix m1(2,3,2,data1.data());
    Matrix m2(2,3,2,data2.data());
    Matrix m3(2,3,2,data3.data());
    // Test cases for a known viscosity. In this case, we're using water

    m1.print();
    m2.print();

    m3.addMatrices(m1,m2);
    m2.print();

   // Test case with varying arbitrary viscosity values within orders of magnitude
   vector<double> mu = {0, 0.0001, 0.001, 0.01, 0.1, 1, 10};// Vary viscosity within
    order of magnitudes
   for (int ii = 0; ii< mu.size(); ii++){
    vector<double> data5 = {400*mu[ii],0,0, 0,7*mu[ii]*0.5,0,0,0};//Shear Matrix
    Matrix m5(2,3,2,data5.data());
    m3.addMatrices(m1,m5);
    m5.print();

}
     */
    return 0;
}
```

A.4 "application_perturbedFlow.cpp"

```cpp
//Compile line: icpc -I${TACC_GSL_INC} application_perturbedFlow.cpp
#include <iostream>
#include <vector>
#include <array>
#include "gsl/gsl-lite.hpp"
#include "input_forKnownVelocityProfile.h"
#include "postProcessing.h"
#include <algorithm>


using std::fill;


#include <chrono>
using namespace std::chrono;

#define INDEX(i,j,lda) (j)*(lda) + (i)


using namespace std;
using gsl::span;



class Matrix {

    private:

    int m,n,lda;
    span<double> data;

    public:

    //Getters
    int getrows() { return m; }
    int getcols() { return n; }
    int getlda() { return lda; }

    //Constructors (ex. 60.3)
    Matrix(int m, int lda, int n, double *data) {
        if (lda < m) {
            cout << "Error creating matrix: LDA < m" << endl;
            throw(1);
        }

        this->m = m;
        this->lda = lda;
        this->n = n;
        this->data = span<double> (data,lda*n);

    }

    //return element function (ex. 60.3)
    double& at(int i, int j) {
        if(i >= m || j >= n || i < 0 || j < 0) {
            cout << "Error: index out of bounds" << endl;
            throw(1);
        }
        return data[j*lda + i];

    }
    auto get_double_data() {
        double *adata;
        adata = data.data();
        return adata;
    }
    //Addition method (ex. 60.4)
    void addMatrices(Matrix& B, Matrix& out) {
        if (this->getrows() != B.getrows() || this->getcols() != B.getcols()) {
            cout << "Error using addMatrices: Matrices do no have the same dimensions " << endl;
            throw(1);
        }

        auto adata = this->get_double_data();
        auto bdata = B.get_double_data();
        auto cdata = out.get_double_data();
        for(int j = 0; j < this->getcols(); j++) {
            for(int i = 0; i < this->getrows(); i++) {
                #ifdef DEBUG
                    cdata[INDEX(j,this->getrows(),i)] = this->at(i,j) + B.at(i,j);
                #else
                    cdata[INDEX(j,this->getrows(),i)] = adata[INDEX(j,this->getlda(),i)] + bdata[INDEX(j,B.getlda(),i)];
                #endif
            }
        }
    }


    //Submatrices support (ex. 60.6)
    Matrix Left(int j) {
        return Matrix(this->m,this->lda,j,this->get_double_data());
    }
    Matrix Right(int j) {
        return Matrix(this->m,this->lda,n-j,this->get_double_data() + lda*j);
    }
    Matrix Top(int i) {
        return Matrix(i,this->lda,this->n,this->get_double_data());
    }
    Matrix Bot(int i) {
        return Matrix(m-i,this->lda,this->n,this->get_double_data() + i);
    }

    //Multiplication functions
    void MatMult(Matrix& other, Matrix& out) {
        auto adata = this->get_double_data();
        auto bdata = other.get_double_data();
        auto cdata = out.get_double_data();
        for(int i = 0; i < this->m; i++) {
            for(int j = 0; j < other.getcols(); j++) {
                for (int k = 0; k < this->n; k++) {
                    #ifdef DEBUG
                        out.at(i,j) += this->at(i,k) * other.at(k,j);
                    #else
                        cdata[INDEX(i,j,out.getlda())] += adata[INDEX(i,k,this->lda)] * bdata[INDEX(k,j,other.getlda())];
                    #endif
                }
            }
        }
        return;
    }


    void BlockedMatMult(Matrix& other, Matrix& out) {

        Matrix atl = this->Left(this->getcols()/2).Top(this->getrows()/2);
        Matrix atr = this->Right(this->getcols()/2).Top(this->getrows()/2);
        Matrix abl = this->Left(this->getcols()/2).Bot(this->getrows()/2);
        Matrix abr = this->Right(this->getcols()/2).Bot(this->getrows()/2);

        Matrix btl = other.Left(other.getcols()/2).Top(other.getrows()/2);
        Matrix btr = other.Right(other.getcols()/2).Top(other.getrows()/2);
        Matrix bbl = other.Left(other.getcols()/2).Bot(other.getrows()/2);
        Matrix bbr = other.Right(other.getcols()/2).Bot(other.getrows()/2);

        Matrix otl = out.Left(out.getcols()/2).Top(out.getrows()/2);
        Matrix otr = out.Right(out.getcols()/2).Top(out.getrows()/2);
        Matrix obl = out.Left(out.getcols()/2).Bot(out.getrows()/2);
        Matrix obr = out.Right(out.getcols()/2).Bot(out.getrows()/2);

        atl.MatMult(btl,otl);
        atr.MatMult(bbl,otl);

        atl.MatMult(btr,otr);
        atr.MatMult(bbr,otr);

        abl.MatMult(btl,obl);
        abr.MatMult(bbl,obl);

        abl.MatMult(btr,obr);
        abr.MatMult(bbr,obr);

    }

    //Ex 60.7
    void RecursiveMatMult(Matrix& other, Matrix& out) {

        if(this->getrows() < 4 && this->getcols() < 4 && other.getrows() < 4 && other
    .getcols() < 4) {
            this->MatMult(other,out);
        }
        else {
            Matrix atl = this->Left(this->getcols()/2).Top(this->getrows()/2);
            Matrix atr = this->Right(this->getcols()/2).Top(this->getrows()/2);
            Matrix abl = this->Left(this->getcols()/2).Bot(this->getrows()/2);
            Matrix abr = this->Right(this->getcols()/2).Bot(this->getrows()/2);

            Matrix btl = other.Left(other.getcols()/2).Top(other.getrows()/2);
            Matrix btr = other.Right(other.getcols()/2).Top(other.getrows()/2);
            Matrix bbl = other.Left(other.getcols()/2).Bot(other.getrows()/2);
            Matrix bbr = other.Right(other.getcols()/2).Bot(other.getrows()/2);

            Matrix otl = out.Left(out.getcols()/2).Top(out.getrows()/2);
            Matrix otr = out.Right(out.getcols()/2).Top(out.getrows()/2);
            Matrix obl = out.Left(out.getcols()/2).Bot(out.getrows()/2);
            Matrix obr = out.Right(out.getcols()/2).Bot(out.getrows()/2);

            atl.RecursiveMatMult(btl,otl);
            atr.RecursiveMatMult(bbl,otl);

            atl.RecursiveMatMult(btr,otr);
            atr.RecursiveMatMult(bbr,otr);

            abl.RecursiveMatMult(btl,obl);
            abr.RecursiveMatMult(bbl,obl);

            abl.RecursiveMatMult(btr,obr);
            abr.RecursiveMatMult(bbr,obr);
        }
    }

    //for testing purposes
    void print() {
        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                cout << this->at(i,j) << " ";
            }
            cout << endl;
        }
        cout << endl;
    }

    void printdata() {
        for(int i = 0; i < lda*n; i++) {
            cout << data[i] << " ";
        }
        cout << endl << endl;
    }

};



int main() {

    vector<double> LDA = {64, 512, 1024, 2048,8192 };// 8192 seems to be the limit
    vector<int> lda = {16,32,64, 512, 1024, 2048, 3072, 3584,4096};
    double y_max = 50; // radius of the pipe, [m]
    double x = 100.0; // horizontal location on the pipe, [m]
    double z = 50.0; // 3D location on the pipe, [m]
    double g = 9.81; // gravity, [kg/m*s^2]
    double mu = 1e-4; // dynamic viscosity of water, [kg  m 1   s 1 ]

    vector<double> data1 = perturbedShear_calculator( LDA[4], y_max, x, z, mu);// Shear Matrix, water.
    vector<double> data2 = surfaceRoughnes_calculator( LDA[4], y_max, x, z, mu);// Shear Matrix, water.
    for (auto jj : lda){
        vector<double> time_BaseMult;// amount of time for base multiplication
        vector<double> time_RecursiveMult; // amount of time for recursive application
        cout << "+++++++++++++++ Testing for " << jj << "by"<< jj << "matrix +++++++++" << endl;
        for(int ii=0; ii<5; ii++){
            // Test base multiplication
            //cout << "Computing Matrix Product with Base Multiplication Function. Result is: " << endl;
                Matrix m1(jj,8192,jj,data1.data());
                Matrix m2(jj,8192,jj,data2.data());
                //Matrix m3(jj,8192,jj,data1.data());
            auto start = high_resolution_clock::now();    // time product function
                m1.MatMult(m1,m2);
                //m2.print();
            auto stop = high_resolution_clock::now();
            auto duration = duration_cast<microseconds>(stop - start);
            cout << "Time taken by Base Multiplication Function: "
            << duration.count() << " microseconds" << endl;
                time_BaseMult.push_back(duration.count());

            // Test recursive multiplication
            //cout << "Computing Matrix Product with Recursive Multiplication
    Function. Result is: " << endl;
                Matrix mr1(jj,8192,jj,data1.data());
                Matrix mr2(jj,8192,jj,data2.data());
                //Matrix mr3(jj,8192,jj,data1.data());
            auto startRecursive = high_resolution_clock::now();    // time product
    function
                mr1.RecursiveMatMult(mr1,mr2);
                //mr2.print();
            auto stopRecursive = high_resolution_clock::now();
            auto durationRecursive = duration_cast<microseconds>(stopRecursive -
    startRecursive);
            cout << "Time taken by Recursive Multiplication function: "
            << durationRecursive.count() << " microseconds" << endl;
                time_RecursiveMult.push_back(durationRecursive.count());
        }

        cout << "+++++++++++++++++++ Tested "<< jj  << " dimension" <<
                "+++++++++++++++++++++++++++++++" << endl;
            cout << "Average Time taken by Base Multiplication Function: "
            << average(time_BaseMult) << " microseconds" << endl;

            cout << "Average Time taken by Recursive Multiplication Function: "
            << average(time_RecursiveMult) << " microseconds" << endl;

        }
    return 0;
}
```

A.5 ”input_forKnownVelocityProfile.h”

```
#include <iostream>
#include <vector>
#include <random>
using namespace std;

// shear sub-routines
double calculate_exx(double x, double y, double z, double mu) {
    return (2*x*z*z*z);
}

double calculate_eyy(double x, double y, double z, double mu) {
    return (3*y*y + 1);
}

double calculate_ezz(double x, double y, double z, double mu) {
    return 0;
}
```

```

double calculate_exy(double x, double y, double z, double mu) {
    return (2*y - 1)/(4*mu);
}

double calculate_ezx(double x, double y, double z, double mu) {
    return (3*x*x*z*z)/2 + 1/2;
}

double calculate_ezy(double x, double y, double z, double mu) {
    return (3*y*y)/2;
}

vector<double> surfaceRoughnes_calculator(double LDA, double y_max, double x, double
    z, double mu){
    /* Compute the scaling matrix that accounts for boundary layer (BL) interactions
    at the wall
     *  These numbers will be very small since the BL are usually ~4 orders of
    magnitude smaller
     *  than the wall. Normally we would have to use numerical or analytical
    equations (e.g Blasius Equations)
     *  to get values for the wall. But for our application, we will just fill a
    matrix with random numbers
     *  between 0 and 1e-5
     */
    vector<double> input;
    double step = y_max/((LDA*LDA)-1); // denominator should always be total number
    of elements divided 1 - LDA

    // initialize random number for perturbations in the flow
    std::random_device rd;
    std::default_random_engine eng(rd());
    std::uniform_real_distribution<double> distr(0, 1e-5);
    for(double y = 0; y < y_max; y += step ){
            double pert =  distr(eng);
            input.push_back(pert);
    };
    return input;
};

vector<double> perturbedShear_calculator(double LDA, double y_max, double x, double z
    , double mu) {
    /* Compute shear forces and other perturbation forces that affect shear
     * Parameters for testing function
    double LDA = 8; // number of rows or columns in a matrix
    double y_max = 50; // radius of the pipe, [m]
    double x = 100.0;
    double z = 50.0;
    double mu = 1e-4;
    */
    vector<double> input;
    double step = y_max/((LDA*LDA)-1); // denominator should always be total number
    of elements divided 1 - LDA
    int counter = 0;

    // initialize random number for perturbations in the flow
    std::random_device rd;
    std::default_random_engine eng(rd());
    std::uniform_real_distribution<double> distr(0, 1e-5);

    for(double y = 0; y < y_max; y += step ){
        // cout << (y) << endl;
        //input.push_back(ii);
        // insert shear matrix
        if (counter == 0){
            input.push_back(calculate_exx(x, y, z, mu));
            counter++;
        }else if (counter == 1){
            input.push_back(calculate_exy(x, y, z, mu));
            counter++;
        }else if (counter == 2){
            input.push_back(calculate_ezx(x, y, z, mu));
            counter ++;
        }else if(counter == (LDA)){
            input.push_back(calculate_exy(x, y, z, mu));
            counter ++;
        }else if (counter == (LDA)+1){
            input.push_back(calculate_eyy(x, y, z, mu));
            counter ++;
        }else if (counter == (LDA)+2){
            input.push_back(calculate_ezy(x, y, z, mu));
            counter ++;
        }else if (counter == (LDA*2)){
            input.push_back(calculate_ezx(x, y, z, mu));
            counter ++;
        }else if (counter == (LDA*2)+1){
            input.push_back(calculate_ezy(x, y, z, mu));
            counter ++;
        }else if (counter == (LDA*2)+2){
            input.push_back(calculate_ezz(x, y, z, mu));
            counter ++;
        }else{
            double pert =  distr(eng);
            input.push_back(pert);
            counter ++;
        }
    }
return input;
}
```

A.6 ”postProcessing.h”

```
/*
 * Header file with simple subroutines to compute mean, standard deviation
 * couldn't find a C++ stats library that made sense:(
 */

// source code courtesy of stack overflow
#include <iostream>
#include <vector>
#include <numeric>

double average(std:: vector<double> v){
    if(v.empty()){
        return 0;
    }
    double sum = 0;
        for(int ii = 0; ii<v.size(); ii++){
            sum = sum + v[ii];
    }
    double mean = sum / v.size();

    return mean;
}
/*
double standard_dev(std::vector<double> const& v){
    double sum = std::accumulate(v.begin(), v.end(), 0.0);
    double mean = sum / v.size();

    double sq_sum = std::inner_product(v.begin(), v.end(), v.begin(), 0.0);
    double stdev = std::sqrt(sq_sum / v.size() - mean * mean);
    return stdev;
}
*/
```