Moonis Ali Khalid Aleksandar Davidov

Github.com/aleksda & github.com/moonnesa

Computational physics FYS3150

PROJECT 1

**SOLVING THE ONE-DIMENTIONAL POISSON’S EQUATION**

**TRIDIAGONAL MATRIX SOLVERS:**

**GAUSSIAN ELIMINATION AND LU DECOMPOSITION**

# **1 introduction**

The main aim of this project was for us to get comfortable with vectors and matrices, and using them to solve problems numerically. In this particular project, we used the programing language C++ where we used dynamical memory allocation to solve the one-dimensional Poisson’s equation with Dirichlet boundary condition using forward and backwards substitution.  
 Later in the project, we used the Linear Algebra library “*Armadillo*” To help us solve the same project, but with the use of LU Decomposition. *Armadillo* is a matrix algebra library that works together with *LAPACK* and *BLAS* to make matrix operations simpler to code in C++.

# **2 General Theory and methods**

The Poisson’s equation is a classical equation in electromagnetism. In three dimensions, the equation is:

Where is the Nambla operator, Φ is the electrostatic potential generated by a localized charge distribution .

The Laplace operator , in three dimensions can be expressed using spherical coordinates, but in this problem, we are assuming that Φ and are spherically symmetric, thus reducing the equation only dependent on the radius, making it one-dimensional.

# Further, we can substitute Φ(r) with . This will reduce the Poisson’s equation to:

Now if we let and , then we get

**Analytical solution**

The one-dimensional Poisson’s equation will be solved with Dirichlet boundary condition by rewriting it as a set of linear equations.

The equation we will solve is:

In our case, we will set the source-term to be   
and the analytical solution is

The first step to solve the linear equation is defining the discretized approximation on *u*. We call it *vi*, with grid points being *xi = xi* from the interval *x0 = 0* to *xn+1 = 0.*   
Since *v* is an approximation of *u,* the boundary conditions are   
*vo = 0 and vn+1 = 0.*

The approximation of the second derivative gives us:

Where *h* is the step length with spacing *1/(n+1)*, and *fi = f(xi).*

Now, if we set *bi=h2fi ,* the approximation of the second derivative can be rewritten as:

This can be rewritten as a tridiagonal matrix on the form

This type of matrix with all elements equal to zero except the main diagonal and upper/lower diagonal is called a tridiagonal matrix.

**3 Algorithms**

We have used two different kind of numerical solvers to solve this tridiagonal matrix A. The first method that we use in b), c) and d) is gaussian’s elimination of a tridiagonal matrix. In b we assume that all the elements of the matrix are different and set up an algorithm that use gaussian’s elimination method to solve the equation. This algorithm is the modified to solve a matrix of identical diagonal elements in c and d. This was done in two steps which we will explain in detail further in this report. The second numerical solver that was used to solve this matrix was LU-decomposition method. All algorithms that solve this matrix are implemented in C++, plots are implemented in python. Code snippets are included and will be discussed and explained further in this section.

**3.1**  **Tridiagonal Matrix Algorithm**

Our tridiagonal system is represented by a linear equation;

with *ai = −1*, *bi = 2* and *ci = −1*, except for a1 = 0 and *cn = 0,* and **f** = h2fi. By row reducing the matrix we can see how our algorithm in b works. For example;

Where we let = b1 and = - c1/\*a2 , as new elements start to appear on the right hand side of the matrix, they are also relabeled to *.* By row reducing this matrix further:

We end up here, we have pivot elements in diagonal and the lower triangular part of the matrix is all 0’s.

As we said before gaussian method of elimination has two steps, this is step one.

***Step1 : Forward substitution***

This computation that you saw above is referred to as the “*forward substitution*”:

*βi* = bi - ai ci-1/*βi-1* , *f*˜*i* = *fi* − a*i f*˜*i-*1*/βi-1, i* ∈ {2, *n*},

We were asked to set up a general algorithm for solving the equation for a tridiagonal matrix with non identical diagonal and non diagonal elements. In C++, we implement this as shown below,

Code snippet for project1b(check link under References to my github repository with my codes)

Code snippet 1: project1b

//Forward substitution

//Boundary conditions

diag[1] = d[1];

f\_tilde[1] = b\_tilde[1];

for (int i = 2; i <=n; i++) {

diag[i] = d[i] - ((a[i]\*c[i-1])/diag[i-1]);

f\_tilde[i] = b\_tilde[i] - ((f\_tilde[i-1]\*a[i])/diag[i-1]);

}

Then comes the second step of the elimination method.

***Step2 : Backward substitution***

In this step our goal is to get all 0’s in the upper triangular part of the matrix also.As you can see below, after row reducing the matrix, (after performing the forward substitution), we need to ensure that all elements in this row reduced matrix are pivot elements,

thats why we perform step 2 that is referred to as *“backward substitution”*.

*vi* = (*f*˜*i* − *civi*+1)*/βi , where i* ∈ {*n* − 1*,n* − 2*,...,*1}

Note*: vn = f*˜*n*  That’s why we use *vi*+1 instead of *f*˜*i+1*

Code snippet 2: project1b

//Backward substitution

//Boundary condition

v[n] = f\_tilde[n]/diag[n];

for (int i = n-1; i >=1; i--){

v[i] = (f\_tilde[i]- c[i]\*v[i+1])/diag[i];

}

**3.3**  **FLOPS: Gaussian’s Elimination method**

In order to make an algorithm fast, compact and efficient we need to think about FLOPS, the number Floating Point Operations, the algorithm requires. For the forward substitution method there are 3 FLOPS in each iteration and n-1 iteration(because my for loop begins from 2 to n which means n-1 elements), that means my code requires 3(n-1) FLOPS for the entire loop. My Backward substitution requires 3FLOPS per iteration, which means 3(n-1) FLOPS for the whole loop. This means that my tridiagonal matrix solver requires in total:

6(n-1)FLOPS

Now in the rest of the project i am told to work with the tridiagonal matrix with know and identical diagonal and non diagonal elements. i.e ( a = c = -1, d = 2). Which makes the solving algorithm much faster and more compact. And my forward and backward substitution code looks like this:

Code snippet 3: project1c-d

// Forward substitution

f\_tilde[1] = b[1];

diag[1] = d[1];

for (int i = 2; i <=n; i++) {

diag[i] = d[i]-(1.0/diag[i-1]);

f\_tilde[i] = b[i] + f\_tilde[i-1]/diag[i-1];

}

// Backward substitution

v[n] = f\_tilde[n]/diag[n];

for (int i = n-1; i >=1; i--){

v[i] = (f\_tilde[i]+ v[i+1])/diag[i];

}

# This algorithm only requires 5(n-1) FLOPS, 2(n-1)FLOPS for Forward- and 3(n-1)FLOPS for Backward-substitution.

**3.4**  **LU-Decomposition:**

LU decomposition forms the backbone of other algorithms in linear algebra, such as the solution of linear equations given by

a11x1+a12x2+a13x3 +a14x4 = w1

a21x1+a22x2+a23x3 +a24x4 = w2

a31x1+a32x2+a33x3 +a34x4 = w3

a41x1+a42x2+a43x3 +a44x4 = w4

The algorithm for obtaining L and U is actually quite simple. We start always with the first column. In our simple (4×4) case we obtain then the following equations for the first column

a11 = u11

a21 = l21u11

a31 = l31u11

a41 = l41u11,

which determine the elements u11, l21, l31 and l41 in **L** and **U**. Writing out the equations for the second column we get

a12 = u12

a22 = l21u12+u22

a32 = l31u12+l32u22

a42 = l41u12+l42u22.

Here the unknowns are u12, u22, l32 and l42 which can all be evaluated by means of the results from the first column and the elements of **A**. Note an important feature. When going from the first to the second column we do not need any further information from the matrix elements ai1. This is a general property throughout the whole algorithm. Thus the memory locations for the matrix **A** can be used to store the calculated matrix elements of **L** and **U**. This saves memory.

We can generalize this procedure into three equations

i < j : li1u1 j+li2u2 j+···+liiui j = ai j

i = j : li1u1 j+li2u2 j+···+liiuj j = ai j

i > j : li1u1 j+li2u2 j+···+lijuj j = ai j

More about LU-decomposition can be read in Lecture Notes by Morten Hjorth-Jensen. (Link in references)

In this project as said before we used armadillo to LU-decompose our tridiagonal matrix. We used a method called *solve()* , that is included in our armadillo library to find the approximated solution of our poisson’s equation.

With an LU decomposed matrix, the number of floating point operations for solving a set of linear equations scales as O(n2).

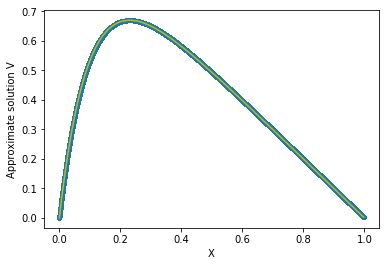
# **4 Results**

Table Comparing The time it took to solve the poisons equation using gaussian method vs LU Decomposition:

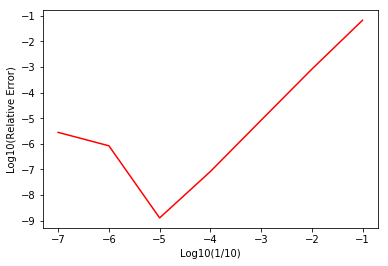
|  |  |  |
| --- | --- | --- |
| n | GEM[s] | LU[s] |
| 10 | 0.000000 | 0.001000 |
| 102 | 0.000000 | 0.001000 |
| 103 | 0.000000 | 0.005000 |
| 104 | 0.000000 | 0.444000 |
| 105 | 0.003000 | n/a |
| 106 | 0.021000 | n/a |
| 107 | 0.211000 | n/a |

We found out our Gaussian Elimination Method is much faster then LU-decomposition, and LU-decomposition cant solve a matrix of size higher the 104. That’s why our table says n/a.

For n = 107 : Exact vs v



Log10 of Relativ error vs Log10 of 1/10i where i = 1,….,7



# **5 Conclusion**

In conclusion we can say that Gaussian Elimination method is much faster and more precise then LU-decomposition. While GEM uses 5-6(n-1) flops LU-decomposition uses O(n2) flops. We also saw that the error in the numerical solution decreases for more grid points but after 105 grid points it increases again due to small values. So its best to limit our matrix to 105 dimensions.

# **References:**

1. Hjorth-Jensen, M. (2015). *Computational Physics - Lecture Notes 2015*. University of Oslo
2. <https://github.com/gregwinther/FYS3150-ComputationalPhysics>