FYS3150: Computational Physics

1D Dirichlet Problem

Authors
Moonis Ali Khalid

Teacher
Morten Hjorth-Jensen

9. September, 2020

UiO : University of Oslo

Abstract

A program was written that solves a Dirichlet boundary value problem, Poisson ep, using 3 linear algebra algorithms. The problem was then solved for various grid resolutions to investigate calculation time and error scaling. Numerical error follows the $\mathcal{O}(h^2)$ prediction up to $n \simeq 10^5$. The time results show that the general and specialized Thomas algorithms are far superior to dense LU decomposition over all ranges of n, and follow a power law with exponent near 0.79, where general LU decomposition has an exponent near 2.

1 Introduction

The purpose of this project was to get f amiliar with numerous i mportant aspects of computational physics. This i ncludes, but i s not l imited to, discretization of a differential equation, dynamic memory allocation and writing programs i n c++, but most importantly to solve the Poisson eq in 1D for three different algorithms and comparing them to see which is "best". The task at hand was to solve a boundary value problem of the f orm:

$$-\frac{\partial^2 u}{\partial x^2} = f(x) \qquad u(0) = c_1 \quad u(1) = c_2 \tag{1}$$

Which involves approximating the second derivative numerically and also constructing and solving a tridiagonal Toepliz matrix. Additionally, numerical error and algorithm computation time was investigated. In principle $f(x)$ could be any continuous function on the domain in question, but for this exercise we limit ourselves to $f(x) = 100\exp(-10x)$ and to $x \in [0, 1]$. The effect of changing boundary conditions is also considered, but for the analysis itself, only $u(0) = u(1) = 0$ is considered.

2 Methodology

2.1 Discretizing the Dirichlet problem

First, we must accept the fact that a boundary value is of a continuous nature, whereas computers are not. This means we have to come up with some effective way to approximate the continuous calculus problem by means of some finite set of equations. This is all done by manipulating the series expansion of our function $u(x)$, to obtain:

$$\frac{\partial^2 u}{\partial x^2}\Big|_{x=a} = \frac{u(a-h) - 2u(a) + u(a+h)}{h^2} + \mathcal{O}(h^2) \tag{2}$$

Now we introduce a finite grid of equally spaced points, with distance h between them[1].

$$x_i = ih \qquad i = 0, 1, 2, ..., n \tag{3}$$

We define the grid so that $h = 1/n$, and the total number of grid points becomes $n + 1$, and thus when we disregard the boundary values there are $n - 1$ grid points where we wish to approximate $u(x)$. Also we introduce the notation $f_i = f(x_i)$. With this notation and definition of grid points, our numerical estimate of the second derivative becomes:

$$\frac{\partial^2 u}{\partial x^2}\Big|_{x=x_i} = \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + \mathcal{O}(h^2) \qquad i = 1, 2, ..., n-1 \tag{4}$$

Equating our approximated formula for the left hand side of Eq.(1) to the right, we arrive at:

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 f_i \qquad 1, 2, ..., n-1 \tag{5}$$

Which is equivalent to a linear algebra problem of the form $\mathbf{A}\vec{u} = \vec{f}$, where $\mathbf{A}$ is a tridiagonal Toepliz matrix, with 2 on the diagonal and -1 on the off-diagonals.

2.2 The Thomas algorithm

The Thomas algorithm is an efficient way to solve a tridiagonal matrix problem. It's comprised of two steps: A forward elimination and a backward elimination. To better illustrate this, consider the matrix defined by three sets of constants:

$$A = \begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-2} & b_{n-2} & c_{n-2} \\ & & & & a_{n-1} & b_{n-1} \end{bmatrix} \tag{6}$$

For simplicity, the 0 elements have not been drawn. Now we identify the row operations required to put this matrix in row echelon form. This is done by eliminating the a_i indexes. For a given row i, we eliminate the a_i term by subtracting the $(i-1)$ line multiplied by the correct scalar:

$$\begin{bmatrix} \ddots & \ddots & \ddots & & \\ 0 & \tilde{b}_{i-1} & c_{i-1} & & \\ & a_i & b_i & c_i & \\ & & \ddots & \ddots & \ddots \end{bmatrix} \xrightarrow{r_i - \left(\frac{a_i}{\tilde{b}_{i-1}}\right) r_{i-1}} \begin{bmatrix} \ddots & \ddots & \ddots & & \\ 0 & \tilde{b}_{i-1} & c_{i-1} & & \\ & 0 & \tilde{b}_i & c_i & \\ & & \ddots & \ddots & \ddots \end{bmatrix} \tag{7}$$

Where:

$$\tilde{b}_i = b_i - \left(\frac{a_i}{\tilde{b}_{i-1}}\right) c_{i-1} \tag{8}$$

As can be seen, the tridiagonal structure has the nice benefit of leaving c_i untouched. Of course the same row operations are always applied to the right hand side $\vec{f}$ as well [4]. Now we could define new arrays $\tilde{b}$, and $\tilde{f}$ and loop over their indices until we get the row echelon form of the tridiagonal system, or we could realize that there is no point in storing the original b_i's and f_i's and instead:

$$b_i := b_i - \left(\frac{a_i}{b_{i-1}}\right) c_{i-1} \qquad i = 2, 3, ..., n-1 \tag{9a}$$

$$f_i := f_i - \left(\frac{a_i}{b_{i-1}}\right) f_{i-1} \qquad i = 2, 3, ..., n-1 \tag{9b}$$

Here the := operator should not be thought of as an equality, but rather an assignment of value. This means that as we loop over the matrix rows, we simply assign a new value to b_i and f_i, based on their old value and the results from the previous row. This saves a decent amount of memory compared to storing duplicated matrix elements that we don't need.

For the backward substitution, the thought process is similar. By the time we're done eliminating a_i, the matrix is in row echelon form and the first unknown is identified in the last row:

$$u_{n-1} = \frac{f_{n-1}}{b_{n-1}} \tag{10}$$

Carrying on from here, the rest of the unknowns are obtained by:

$$u_i = \frac{f_i - u_{i+1}}{b_i} \qquad i = n-2, n-3, ..., 1 \tag{11}$$

Until now we have assumed nothing about the initial values of the matrix elements a_i, b_i and c_i. A simplification can be made when we know that the matrix elements are constant, that is:

$$a_i = -1, b_i = 2, c_i = -1 \forall i$$

Thus for each row, the scaling factor is the same:

$$b_i := b_i - \left(\frac{a_i}{b_{i-1}}\right) c_{i-1} = 2 - \frac{1}{b_{i-1}}$$

Now if the array is initialized with $b_1 = 2$ and loop over $i = 2, 3, ..., n-1$ one gets a series of the form:

$$b_1 = 2 \quad b_2 = \frac{3}{2} \quad b_3 = \frac{4}{3} \quad b_4 = \frac{5}{4}$$

Although this is left out here, one could prove by induction that for this special case:

$$b_i = \frac{i+1}{i} \tag{12}$$

A few words on boundary conditions

Now we're considering the Dirichlet case when the values of u_0 and u_n are known. As shown above, Eq.5 defines a linear set of equations for the unknown u_i, $i = 1, 2, ..., (n-1)$. Note however that for $i = 1$ and $i = (n-1)$, we get a special case where for computational purposes, the boundary value actually belongs on the right hand side. We will handle this by setting:

$$f_1 := f_1 + u_0 \tag{13a}$$

$$f_{n-1} := f_{n-1} + u_n \tag{13b}$$

This will be done after initializing $f_i = f(x_i)$ and before the forward elimination. Although the task at hand is to solve the special case of $u(0) = u(1) = 0$, there is no reason not to include the general Dirichlet boundary conditions in the code. The above holds true regardless of solver algorithm.

That being said, solving Eq.1 for other boundary values is not the most interesting exercise, since the null space of the -$\frac{\partial^2}{\partial x^2}$ operator is simply the linear function. So if you were to solve the boundary value problem for $u(0) = c_1$, $u(1) = c_2$ you would always get the same solution shifted by the line $c_1 + (c_2 - c_1)x$.

2.3 Error Analysis

For error estimation we use the following two metrics. Max relative error:

$$\epsilon_r = \max_i \left(\left| \frac{u_i - U(x_i)}{U(x_i)} \right| \right) \tag{14}$$

Where $U(x)$ is the analytical solution to the boundary value problem.

Another way to estimate error is with the integrated square error (or L^2 error), which calculates the area squared between our calculated function and the analytical solution. The integral is approximated by a Riemann sum using our step length h:

$$\epsilon_{sq} = \int_0^1 (u(x) - U(x))^2 \, dx \simeq h \sum_i (u_i - U(x_i))^2 \tag{15}$$

2.4 The basics of LU decomposition

Another way to solve systems of linear equations is with so called lower and upper (LU) decomposition. The method relies on the fact that the initial matrix A can be split into an upper and lower triangular matrix.

$$A\vec{u} = LU\vec{u} = \vec{f} \tag{16}$$

Now define $z := U\vec{u}$ and solve, using back substitution:

$$LU\vec{u} = L\vec{z} = \vec{f} \tag{17}$$

When we have solved for $\vec{z}$, we can then solve for $\vec{u}$:

$$U\vec{u} = \vec{z} \tag{18}$$

Although a special case exists for tridiagonal matrices, when we compare our Thomas algorithm to LU decomposition, no assumptions will be made regarding the nature of A. Thus A will be treated like a dense matrix. The algorithm itself will not be described in detail here, but it is however worth mentioning that the common algorithm involves pivoting, which we thankfully need not worry about since our matrix A is positive definite [1].

3 Programming technicalities

First the basic algorithm was tested in Python as a proof of concept. Afterwords a more efficient C++ program was written, which handles the matrix operations and generates output files. All source code is avalible on Github [3]. The output is then analyzed and plotted using Python with Numpy and Matplotlib. Since the program will be run for various parameters, a Config class was written that reads the following type of input files:

```
input.cfg:
\hline
#This is our input file. Please avoid unnecessary whitespace
#Grid resolution
n=50
#No. of itterations for time averaging
loops=1
#Solver, 0 for general toepliz, 1 for constant toepliz, 2 for LU
solver=0
#Dirichlet boundary Conditions
bc_left=0
bc_right=0
#Choose output, 1-> everything, 0-> only timing and error
long_output=1
#Output file name
out_filename=out.dat
```

Simply put the program performs the following operations:

1. Read input file.
2. Allocate arrays dynamically.
3. Select algorithm.
4. For no. of loops:
    (a) Initialize arrays.
    (b) Solve matrix problem, time how long it takes.
5. Take time average.
6. Calculate errors.
7. Write desired output to file

The majority of the programs `main` function is quite easily readable and will not be discussed in detail here. It starts by reading the input filename as a command line input, loading the settings and dynamically allocating five arrays: Three for the matrix elements, one for the right hand side and one for the result to be stored in. All manipulation of the arrays are handled by short functions. The initialization is done by:

```
void init_arrs( int N, double *A, double *B, double *C,
                double *RHS, double *RESULT,double BC[2])
{

        double h = 1/((double)N);
        double h2 = h*h;
        for (int i=0;i<N-1;i++){
                double x = (i+1)*h;
                RHS[i] = h2*100*exp(-10*x);
        }
        for (int i=0;i<N-1;i++){
                A[i]= -1.0;
                B[i]= 2.0;
                C[i]= -1.0;
        }
        //Apply BCs
        RHS[0] -= A[0]*BC[0];
        RHS[N-2] -= C[N-1]*BC[1];
        RESULT[0] = BC[0];
        RESULT[N] = BC[1];
}
```

This is quite self explanatory. Note that we initialize the first and last element of the `RESULT` array with the left and right boundary conditions. When we later carry out the Thomas algorithm we don't touch the first and last elements.

The main Thomas algorithm is performed by:

```
void do_thomas( int N, double *A, double *B, double *C,
                double *RHS, double *RESULT)
{
        double scalar;
        //Forw. elim.
        for(int i=1;i<N-1;i++){
                scalar = A[i]/B[i-1];
                B[i] -= scalar*C[i-1];
                RHS[i] -= scalar*RHS[i-1];
        }
        //Back. sub.
        RESULT[N-1]=RHS[N-2]/B[N-2];
        for(int i=N-3;i>-1;i--){
                RESULT[i+1]=(RHS[i]-C[i]*RESULT[i+2])/B[i];
        }
}
```

Note that when performing the Thomas algorithm, we don't need to initialize an extra array for the manipulated matrix elements, such as $\tilde{b}$. Instead we can simply overwrite the elements, since we don't care about their initial values, but rather just solving the system with less memory usage so we can potentially handle larger systems. Also another variation of this function (`init_arrs_special`) was written for the special case of constant elements. This function is similar except it ignores the `A` and `C` arrays, and initializes the `B` array using Eq.12, thus removing the need for altering the diagonal during forward eliminations.

During the special case, we can assume that all A and C elements are -1. The algorithm is still the same but we reduce the number of floating point operations by performing:

```
void do_thomas_special(int N, double *B, double *RHS, double *RESULT){
        //Forw. elim.
        for(int i=1;i<N-1;i++){
                RHS[i] += RHS[i-1]/B[i-1];
        }
        //Back. sub.
        RESULT[N-1]=RHS[N-2]/B[N-2];
        for(int i=N-3;i>-1;i--){
                RESULT[i+1]=(RHS[i]+RESULT[i+2])/B[i];
        }
}
```

Comparing the two we notice that for the general case, the forward elimination performs 5 FLOPS per line: First one division for calculating the factor which the line is scaled by, and then a division and subtraction for both the diagonal and the right hand side. During the backwards substitution it performs a multiplication, a subtraction and a division, thus 3 FLOPS. So in total, the `do_thomas` function performs 8 floating point operations per line in the matrix.

By comparison, the `do_thomas_special` algorithm performs performs 2 FLOPS per line during forward elimination, and 3 FLOPS during the backward substitution, yielding a total of 5 floating point operations per line in the matrix. However, solving a dense matrix equation using LU decomposition on the other hand requires a leading term of $\frac{2}{3}n^2$ FLOPS. Thus one would expect the two variatons of the Thomas algorithm to scale much better with increasing system size n.

The programming aspects of implementing LU decomposition will not be shown in detail here. The Armadillo library was used along with its Mat and Vec classes to construct the system in a very similar way, and it has built in function to perform LU decomposition and back substitution as per Eq.16-18 [2].

To evaluate errors, the following function is used:

```
double* get_error(int N,double *calc){
        //Enough to acces calc[1]->calc[N] since calc[0]=calc[N+1] are exact due to BCs.
        double h = 1/((double)N);
        double x;
        double U_analytical;
        double temp_rel_err;
        double max_rel_err=0; //avoid special case for first value. anything is >=0
        double err_sq_sum = 0;
        double *output;
        output = new double[2];
    for (int i=1;i<N;i++){
                x=i*h;
                U_analytical = 1-(1-exp(-10))*x-exp(-10*x);
                //Relative Error
                temp_rel_err = abs((U_analytical-calc[i])/U_analytical);
                if (temp_rel_err > max_rel_err){
                        max_rel_err = temp_rel_err;
                }
                //Error Squared Integral
                err_sq_sum += pow((U_analytical-calc[i]),2)*h;
        }
        //Take the log of both metrics and return
        output[0] = log10(max_rel_err);
        output[1] = log10(err_sq_sum);
        return output;
}
```

The function reads the array of calculated u_i values. It then loops over the indices and calculates two different error estimates. The max relative error is found by storing a max value and reassigning it once you find a point with higher relevant error. The integrated square error is found by a sum as per Eq.15. It then returns a pointer to an array of doubles containing the two error estimates. Note that the first and last element of the result array are neglected, since we previously have fed them with the known boundary values. Estimating an error here is pointless, and causes a division by zero error in the case of $u(0) = u(1) = 0$.

For a more general program (with other $f(x)$), one might want to add an exception to the relative error to deal with interior points where $u(x_i) \simeq 0$, or simply use the L^2 metric, which makes no assumptions other than the fact that the functions are square integrable.

4 Results

First, the general algorithm was used to solve the boundary value problem Eq.1 for $u(0) = u(1) = 0$ and $f(x) = 100 \exp(-10x)$. The results were plotted for grid sizes n of values 10, 100 and 1000 and compared with the analytical solution on Fig.1.

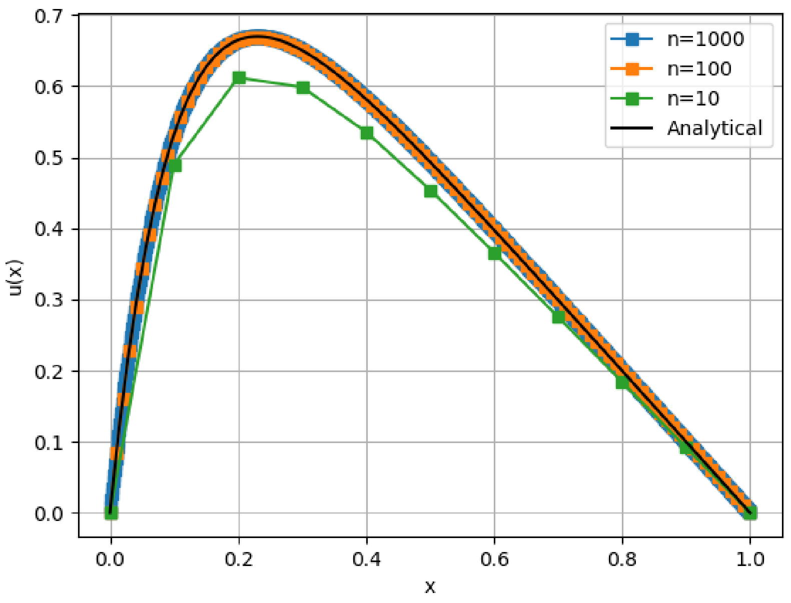

Figure 1: Calculated $u(x)$ for three values of n compared to the analytical solution of Eq.1. Upon inspection the results are already very good by $n = 100$.

Another key aspect of this exercise was to get more familiar with the concept of numerical error. To investigate this, the program was run for various n ranging from 10 to 10^8. Although powers of 10 are aesthetically pleasing, some intermediate values were added where the behaviour seemed interesting. The error values are shown in Tb.1 and plotted on Fig.2. From a discrete mathematics point of view, the numerical error should scale linearly with $h^2 = n^{-2}$, according to Eq.2. Add that with the fact that for smaller h we introduce roundoff errors, one might expect the linear relationship to break down for sufficiently small h or large n. Already at $n \simeq 5 \cdot 10^5$ we see the expected increase in numerical error. We see that for smaller values of n, the numerical error is in quite good agreement with that predicted by Eq.4. That has a leading error term of $\mathcal{O}(h^2)$, so it's reasonable that the slope on a loglog plot becomes close to 2 for relative error (Eq.14) and 4 for L^2 error (since it has dimensions error squared) (Eq.15). This is illustrated on Fig.2.

Table 1: Two numerical error metrics (Eq.14-15) for various grid sizes.

n	$\log_{10}(\epsilon_{sq})$	$\log_{10}(\epsilon_r)$
10^1	-2.894	-1.101
10^2	-6.851	-3.079
10^3	-10.85	-5.079
10^4	-14.85	-7.079
10^5	-18.44	-8.843
$5 \cdot 10^5$	-14.22	-6.448
10^6	-13.06	-6.076
$5 \cdot 10^6$	-12.71	-5.882
10^7	-12.78	-5.525
$5 \cdot 10^7$	-6.732	-2.447
10^8	-5.031	-1.470

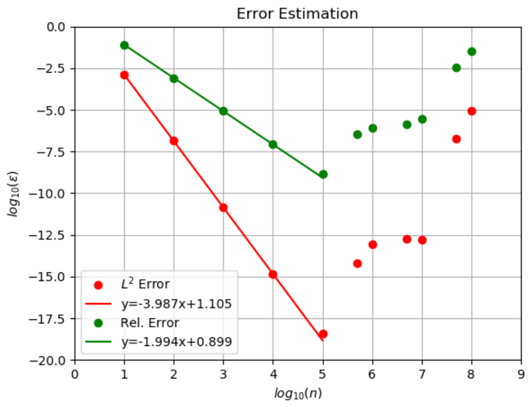

Figure 2: Numerical error estimation for various n. Already at $n \simeq 5 \cdot 10^5$ we start losing precision.

The final aspect of choosing an appropriate numerical method we consider is calculation time. As mentioned earlier, the Thomas algorithm is linear in number of FLOPS for a given matrix size n. This is true both for the case of a general tridiagonal matrix and the case of constant elements, but the prefactor is different. However solving the linear system by means of LU decomposition requires a leading term of $\frac{2}{3}n^2$ FLOPS. To evaluate the run times, the algorithm is timed for multiple runs and an average time is taken. Note that only the algorithm itself is timed, and not initialization, output writing etc. The results of timing is shown on Fig.3.

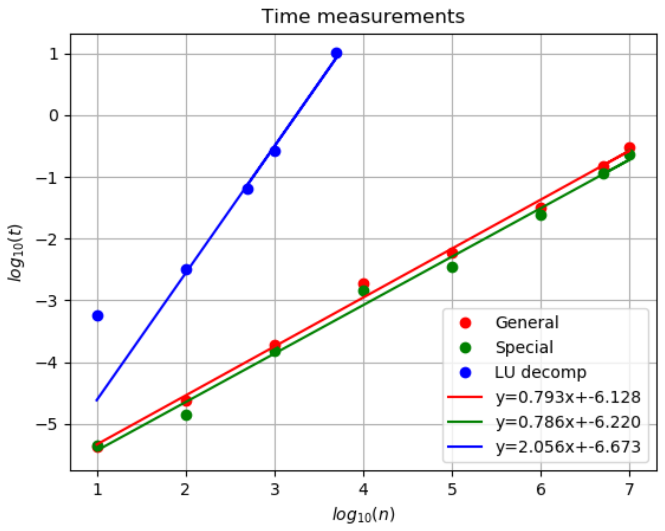

Figure 3: Average time measurements, averaged over 50 runs, for various n and the three algorithms considered.

Timing measurements are very simple to perform, since our input file accepts both an algorithm flag and a time-loops flag. When timing we can also disable the long output and only recieve time and error data, allowing us to go to bigger n without being slowed down by the operating system writing absurdly large output files. The results clearly shows that LU decomposition scales much worse. Also, surprisingly the exponent for the general and constant Thomas algorithm are seemingly closer to 0.79 than 1, despite the fact that the amount of FLOPS is expected to be linear in n. The reason for this is not known. One naive guess could be that the memory operations outweigh the FLOPS when it comes to time usage, and that memory reads and writes scale in some different way.

5 Discussion

A working program was written that solves Eq.1 for $f(x) = 100\exp(-10x)$. The output was compared to the analytical solution, to confirm that the program was in fact functioning correctly. Although not done here, other unit tests which could have been useful to gain more confidence in the program could for example be exporting the diagonal elements b_i and comparing to the analytical expression Eq.12. Also one could have implemented a method to vary $f(x)$ and for example solve for "obvious" cases, such as $f = 0$, or simple polynomials. The error and timing results agree very well with predictions, apart from the fact that our general and specialized Thomas algorithm calculation times seem to scale with an exponent near 0.79, which is noticably different from the predicted 1.0.

Further goals for the next project include structuring the program better, into more smaller .cpp and .h files, to make the code more readable. Also the Config class can be repurposed for future projects, and perhabs a similar Output class can be implemented for cases when we might want to vary the type of output being written to a file.

References

[1] Brian Bradie. *A Friendly Introduction to Numerical Analysis*. Pearson, 2007. ISBN: 8131709426.

[2] Dr. Conrad Sanderson, Ph.D, Dr. Ryan Curtin, Ph.D. *Armadillo: C++ l ibrary f or l inear algebra scientific computing*. URL: `http://arma.sourceforge.net/`.

[3] *FYS3150 Github Repository*. URL: https://https://github.com/moonnesa/FYS3150-2020.

[4] Morten Hjorth-Jensen. *Computational Physics Lecture Notes*. 2015.