Department of Physics, University of Oslo, Norway

Abstract

In this project we solve the one-dimensional Poisson equation with Dirichlet boundary conditions by rewriting it as a set of linear equations. Our best algorithm runs as $4n$ FLOPS with n being the dimension of the matrix. The physical limitation of computers can cause computational errors when using too small values and memory issues when dealing with too large amounts of data.

Project 1

Are Nikolai Sigersvold

September 10, 2019

1 Introduction

This project is all about solving the one-dimensional Poisson equation with Dirichlet boundary. We start of by rewriting it as a set of linear equations and then use a general and specific algorithm to solve it.
The aim of this project is to get familiar with various vector and matrix operations, from dynamic memory allocation to the usage of programs in the library package of the course.
The library used for this project for handling of arrays in C++ is the Armadillo library.
The link to the github domain containing the source codes for project1 is here:
https://github.com/FastFirefly/Project1_arensi

2 Theory, algorithms and methods

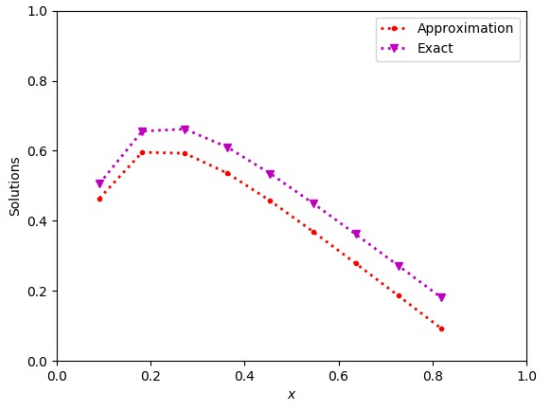
The algorithms that were used in this project were a general algorithm for solving a set of linear equations in a matrix of different elements and one for solving a set of linear equations in a matrix of specific elements. For the specific elements we also used the library armadillo's solve function to test how the LU decomposition could be used to find the approximated values.

For the general algorithms, we used the iterative Jacobi method that checks for an acceptable difference between the approximated values, named **ans** in the code, and the exact result, named **u** in the code. It begins this process by creating a vector of n random values, named **ansOld** in the code,

and another with n zero values, named **ansNew**. The algorithm then uses a while statement to check if certain conditions are met, and then calculate the difference between the answer, **b**, and the solution for the linear equation when our matrix, **A**, is multiplied by the random vector, **ansOld**, and assign the new values as vector **ansNew**. After that it would In our case, the while loop conditions would check if the difference between vector **ansNew** and **ansOld** to see if the difference has reached the minimum acceptable threshold, identified as **e** in the code. It also makes sure that the loop happens at least 10 iterations, just to be safe.
For example: If we use a matrix of random values, the matrix would be multiplied by a random vector and looped over until the difference between the new and previous calculated vectors.
For the specific algorithm, we use a bit less taxing algorithm. As the diagonal and non-diagonal values are known, this results in less expensive computations compared to the general algorithm. As for the method for using this algorithm, we simply use the known values to first calculate the forward substitution and then the backward substitution to find the approximated values.
For the LU decomposition we simply just use the provided solve() function from the armadillo library to find the approximated values. To use this function we provide the matrix, **A** and the vector, **b**, and assign the answer to the vector, **ans**. ans = solve(A, b);

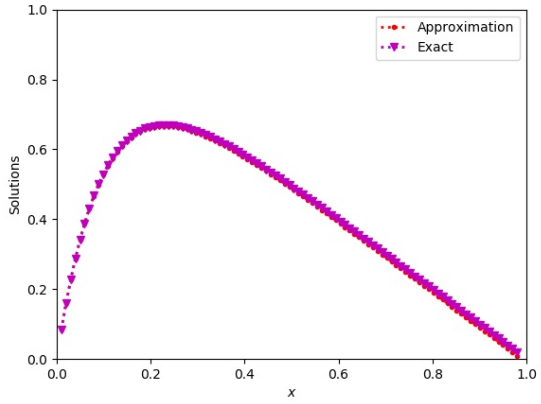
3 Results and discussions

According to the lecture notes for Linear Algebra Methods, the amount of floating point operations used to calculate the approximation for the general algorithm is $\frac{2n^3}{3} + n^2$.
To reproduce the following results, compile and execute the code, 1b.cpp.
Example:
Compile with: g++ -O2 -std=c++0x 1b.cpp -o gaussian -larmadillo
Run with: ./gaussian output_b 3
Where the given 3 will result find the solution n=10, n=100, and n=1000.
Result for the general algorithm for n=10.



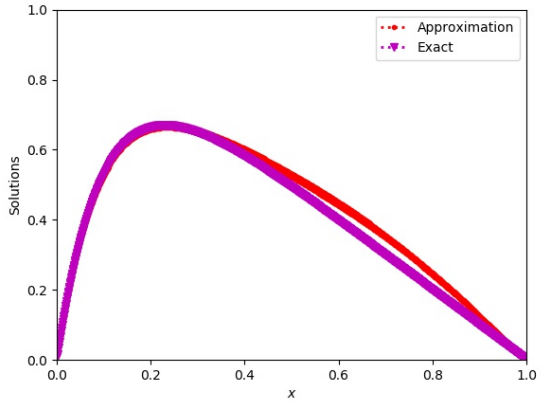
The approximation for n=10 shows to give a similar slope to the exact value, however the approximated values were incorrect with a seemingly constant difference.

Result for the general algorithm for n=100



The approximation for n=100 shows a slope that follow the exact values quite precisely, compared to the n=10 slope.

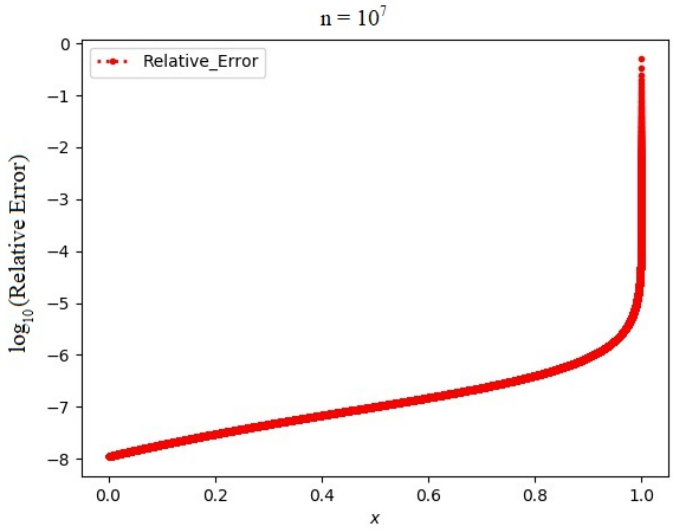
Result for the general algorithm for n=1000



The approximation for n=1000 shows to follow the exact values up until x=0.3 to x=0.9, where the approximation has exceeded the exact value.

For the specialized algorithm, the amount of floating point operations were 4n, where n is the dimension of the matrix. 2n for the forward substitution and 2n for the backward substitution.
To reproduce the following results, compile and execute the code, 1c.cpp.
Example:Compile with: g++ -O2 -std=c++0x 1c.cpp -o simple -larmadillo
Run with: ./simple output_c 6
Where the 6 will result in finding n=10, n=100, n=1000, n=10^5, and n=10^6.

The CPU time for the specialized algorithm for n=10^6 was 11.4375 seconds. The CPU time for the general algorithm could not be computed as the size of the matrix was too extensive, and therefor used too much memory for the applied computer to handle.
To calculate the relative error, we will use the spesialized algorithm.
Relative error for $n = 10^7$



The relative error for the algorithm used to calculate the specified matrix. The formula used to calculate the relative error was: $\epsilon = log_{10}(|\frac{v_i - u_i}{u_i}|)$. The large relative error might be a result of a lack of precision from the computer as the values used to find the approximations are minuscule, requiring the machine to compute with values that have roughly 10^7 steps between the 0 to 1. This might lead to small error overall becoming large ones because the errors are relatively sizeable compared to the used values.

Algorithms	n=10 Time/s	n=100 Time/s	n=1000 Time/s	n=10^4 Time/s
Specific algorithm	0.000000 s	0.000000 s	0.015625 s	0.156250 s
LU algorithm	0.046875 s	0.062500 s	0.625000 s	261.078125 s

The table above shows the average of five time measurements for the spesialized algorithm and the LU decomposition algorithm. As shown, when reaching around $n = 10^4$ the time used for the LU decomposition increases

dramatically. This is mostly due to the large increase in matrix elements that the LU algorithm has to process, which the spesialized algorithm do not require to do as it has been designed to only use elements along the diagonal. The measurement for time for the different algorithms was the clock() function from the C++ time library.
According to the lecture notes for Linear Algebra Methods, The LU decomposition uses $2/3n^3$ floating point operations. We could not run the LU decomposition for a matrix of dimension $n = 10^5$ as we ran out of memory before it could complete the algorithm.

4 Conclusions

As shown from the results and data recorded above, a computer can handle large amounts of variables and compute with enormous amounts data, but they are not always 100% precise in their calculations. Even though the precision of computers is astonishing, they are still affected by physical limitations, such as memory storage or processing power. In works that requires near perfect computations, it is important to remember that these limiting factors can lead to less than ideal results and could in specific circumstances cause small, but critical errors.

5 Compiler and Hardware

The compiler used to compile this program was the GNU Compiler Collection, or g++, and was executed in an Ubuntu 16.04 LTS terminal with a Intel(R) core(TM) i3-7100U CPU @ 2.40GHz processor with 4GB of RAM.

References

Lecture slides by M. Hjorth-Jensen http://compphysics.github.io/ComputationalPhysics/doc/pub/linalg/html/linalg.html