FYS3150 Computational Physics - Project 1

Nicholas Karlsen

A look on two similar algorithms for solving an inhomogeneous second-order differential equation with a known analytical solution and comparing their efficiency to a standard LU-decomposition method. Found that the algorithms allow for greater precision and faster run-times.

INTRODUCTION

When assuming spherical symmetry, the Poisson equation, eqn. 1

$$\nabla^2 \mathbf{\Phi} = -4\pi\rho(\mathbf{r}) \tag{1}$$

can be reduced to the 1-dimensional second order inhomogeneous differential equation, eqn. 2, by assuming spherical symmetry and scaling the problem.

$$\frac{d^2u(x)}{dx^2} = f(x) \tag{2}$$

A numerical solver for this equation not only allows us to solve the Poisson equation, but several other systems as well. In this report i have looked at one such solver, in both its general and specialized form and compared its speed to a solver which employs the LU-decomposition method in scipy and found that the solver was faster, but more importantly, allowed for a solution closer to what is allowed by the machine precision.

THEORY, ALGORITHMS AND METHODS

We have second order inhomogeneous differential equation, Eqn. 3

$$\frac{d^2u(x)}{dx^2} = f(x) \tag{3}$$

With boundary conditions $\ddot{u}(x) = f(x)$, $u(0) = u(1) = 0$ and $x \in (0, 1)$.

If we let $f(x) = 100e^{-10x}$, then it can be shown analytically that the differential equation has a solution Eqn. 4

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{4}$$

This problem can also be solved numerically by discretization. By the use of taylor expansions, it can be shown that there is a discrete, iterative solution in the form Eqn. 5 [1]

$$-\frac{v_{i+1} + v_{i-1} + 2v_i}{h^2} = f_i \tag{5}$$

where $h = 1/(n-1)$ is the step size for n points and V_i is the discretized form of $u(x)$.

This problem can be written in the form

$$A\vec{v} = \vec{b} \tag{6}$$

Where A $\in R^{n\times n}$ is a tridiagonal matrix with diagonal elements 2 and -1 (Eqn. 7) and $\vec{b} = h^2(f_0, \ldots, f_{n-1})$.

$$A = \begin{bmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & 0 \\ 0 & -1 & 2 & -1 & \ldots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & -1 \\ \ldots & \ldots & \ldots & \ldots & -1 & 2 \end{bmatrix} \tag{7}$$

By matrix multiplication, when we multiply the i-th row of A by $\vec{v}$, we get

$$\begin{aligned} -1v_{i-1} + 2v_i - 1v_{i+1} &= -\left(v_{i+1} + v_{i-1} - 2v_i\right) \\ &= h^2 f_i \end{aligned} \tag{8}$$

Which is equivalent to the discretized differential equation Eqn. 5, showing that the differential equation can be solved as a linear algebra problem.

A can be written more generally as

$$A = \begin{bmatrix} b_0 & c_0 & 0 & \ldots & \ldots & 0 \\ a_1 & b_1 & c_1 & 0 & \ldots & 0 \\ 0 & a_2 & b_2 & c_2 & \ldots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & c_{n-2} \\ \ldots & \ldots & \ldots & \ldots & a_{n-1} & b_{n-1} \end{bmatrix} \tag{9}$$

By gauss elimination, we get the following set of equations for forward and backward substitutions respectively, which can be used to solve the system.

$$\begin{aligned} \mathrm{v}_{n-2} \\ \mathrm{v}_i &= \frac{\tilde{f}_i + v_{i+1}}{\tilde{b}_i}, \quad i \in \{n-2, n-3, \ldots, 1\} \end{aligned}$$

Or written in pseudocode/python;

```
    # Forward substitution
    for i in xrange(2, n - 1):
        b[i] = b[i] - ((a[i] * c[i - 1]) / b[i - 1])
```

```
        f[i] = f[i] - ((a[i] * f[i - 1]) / b[i -
    1])
    u[-2] = f[-2] / b[-2]
    # Backward substitution
    for i in xrange(n - 2, 0, -1):
        u[i] = (f[i] - c[i] * u[i + 1]) / b[i]
```

If we asume $\mathcal{O}(n)$ is such that $\pm$ numbers of order 0 become insignificant, then $9n$ FLOPS is required to compute this algorithm.

For the special case, eqn. 7, this algorithm can be further simplified by abusing the fact that the diagonal matrix elements a_i, c_i are identical for all i. Resulting in the following sets of equations, 10, for the substitutions.

$$\tilde{b}_i = b_i - \frac{1}{\tilde{b}_{i-1}}, \quad \tilde{f}_i = f_i + \frac{\tilde{f}_{i-1}}{\tilde{b}_{i-1}}, \quad i \in \{2, 3, \ldots, n-2\}$$
$$v_i = \frac{\tilde{f}_i + v_{i+1}}{\tilde{b}_i}, \quad i \in \{n-2, n-3, \ldots, 1\} \tag{10}$$

This simplification results in $\approx 6n$ flops required to solve the problem, 66% less flops than the general algorithm.

RESULTS AND DISCUSSIONS

The implementation of the code discussed in the previous section can be found on my github: https://github.com/nicholaskarlsen/FYS3150. The code is in its entirety written in python, both due to how quick it is to implement and (mostly) because i am not yet comfortable enough with a more efficient language yet to use anything else. In order to offset some of the shortcomings of python i added some level of optimization using the JIT flag provided in the numba package.

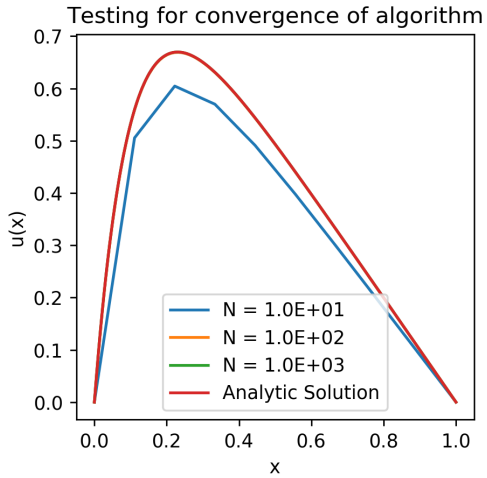

FIG. 1. A plot of the numeric solution for different N and the analytic solution.

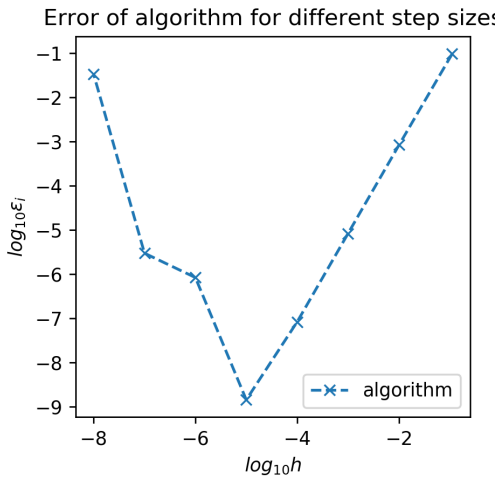

FIG. 2. How the error evolves for smaller step sizes when comparing the general algorithm to the analytic solution

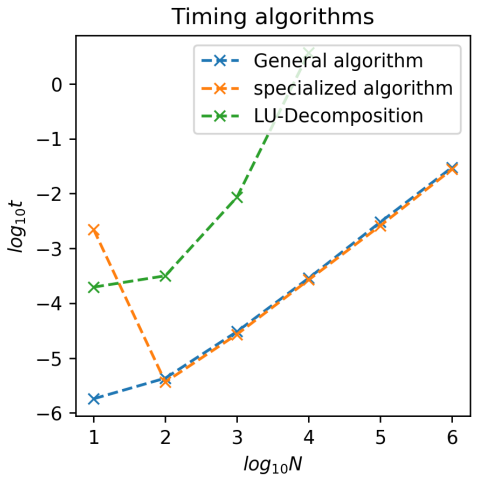

FIG. 3. The average execution time of 10 function calls on my computer

As seen in Figures 1, 2, the algorithm converges towards the analytic solution with a relative error, ϵ such that $\log_{10}\epsilon \propto \log_{10} h$ for $\log_{10} h \gtrsim 5$ which is where loss of precision due to the finite representation of numbers in computers becomes relevant. As such, it is ideal to chose $h \approx 10^{-5}$ when precision is the main priority, but never $h \lesssim 10^{-5}$ as the extra computation time is simply wasted.

In Figures 3, 4 we see how the execution time of the different algorithms compare for different N. This data does not reflect the expected 66% performance increase predicted based on the number of FLOPS. Therefore I suspect there may be something wrong with either the way I'm timing the functions, or the numpy calls prior to the backward/forward loops may be the dominating

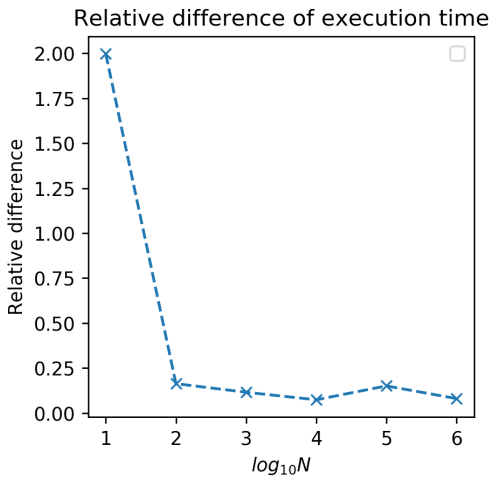

FIG. 4. Percentage difference between the execution times of the general and specialized algorithms on my computer

factor in the execution time. Due to the way i wrote gausselim.general(), i was somewhat limited in the way i could time my function without causing errors, and i did not have the time to explore different options / restructure my code.

Comparing them to the execution time for the scipy LU-decomposition solver however shows that the custom custom algorithms perform very well. First and foremost, the LU-decomposition method is much more limited by memory. On my computer, with 16GiB of memory, the largest exponent of 10 sized matrix that could be solved without running out of memory was 10^4. The execution time also rises grows quicker than for the custom algorithms.

CONCLUSIONS

In conclusion, it has become apparent that there is a lot of time and memory to be saved by remodeling a problem to a more efficient form rather than blindly using a standard solver, thus enabling us to solve problems that may have otherwise not been feasible given our resources. Further, one must be critical when lowering the step sizes of numerical solvers such that machine precision does not become a dominant term in its error.

[1] M. Hjorth-Jensen, Computational Physics - Lecture Notes 2015, (2015).