William Daniels AMS 326 homework three writeup

April 17, 2023

All of my source files can be found in this github repositiory:
https://github.com/William-J-Daniels/DanielsAms326.git

A detailed README on how the repositiory is used is there.

Problem One

Description

The problem is to solve a linear system using three methods: Gaussian elimination, Jacobi iteration, the Gauss-Seidel method, and successive over-relaxation (SOR).

Algorithms

Gaussian elimination

Gaussian elimination finds the solution to a system of linear equations by applying elementary matrix operatrions, which do not change the solution of the matrix, until the coefficient matrix is upper-triangular. Back substitution is then used to solve for the coefficients.

My implementation is found in the header file `LinearAlgebra/include/LinearSolver.h`. It is threaded with open message passing (OMP), and also uses partial pivoting to reduce roundoff error.

Jacobi iteration

This is an iterative method to find the solution to a linear system. One decomposes the matrix M into its diagonal and remaining components, $D+R$. We can then apply the formula $x^{(k+1)} = D^{-1}\left(b - R\,x^{(k)}\right)$, where $x^{(n)}$ is the solution vector after n iteration and b is the right hand side vector, successively until we converge to a soltion.

My impementation of this method is also found in `LinearAlgebra/include/LinearSolver.h`. In the header `LinearAlgebra/include/Matrix.h`, I define classes and overloaded operators for the aritmatic used in the Jacobi iteration implementation. Among them include my matrix class, my diagonal matrix class (which saves on space and computation compared to using the matrix class to represent D^{-1}, especailly when finding the inverse), and overloads involving those two classes and `std::vector` for the aritmatic.

Gauss-Seidel method

This is another iterative method. Instead of using the definition in the notes, I used the element-wise definition of the solution vector found on Wikipedia: https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method#Element-based_formula. This is because finding the inverse of non-diagonal matrices is difficult to code and expensive to run. The element wise formula is

$$x_i^{(k+1)} = \frac{1}{a_{(ii)}}\left(b_i - \sum_{j=i}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}\right), i = 1, 2, \ldots, n$$

Again, my implementation is in `LinearAlgebra/include/LinearSolver.h`.

Successive over-relaxation (SOR)

Our final algorithm is another iterative linear system solver. It is actually a variation on the Gauss-Seidel method designed to improve the convergence rate. Again, I used an element-wise equation from Wikipedia (https://en.wikipedia.org/wiki/Successive_over-relaxation#Formulation). The formulation is

$$x_i^{(k+1)} = (1-\omega) x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij} x_j^{(k)}\right), i = 1, 2, \ldots, n$$

Once more, my implementation is in the header `LinearAlgebra/include/LinearSolver.h`.

Results

Each method obtianed the same solution vector:

$$\left(0.6875, 0.875, 0.6875, 0.875, 1.125, 0.875, 0.6875, 0.875, 0.6875\right)^T$$

Gaussain elimination uses on the order of $n\left(n\left(n-1\right)\left(n-2\right)\ldots\left(1\right)\right)+n!=n!\left(n+1\right)$ for forward elimination and back substitution. With $n=9$, this produces 3.6 TFLOPs.

Jacobi iteration uses on the order of n^2+2n when sparse matrix multiplication for the diagonal matrix is used FLOPS per iteration. For $n=9$, this is 99 FLOPs. Since I used 512 iterations, my progroam used about 51 KFLOPs.

My Gauss-Seidel implementation uses about 11 FLOPs per iteration. With 512 iterations, we use 5.6 KFLOPs.

Finally, my SOR program uses around 13 FLOPs per iteraion, which sounds like a regression from Gauss-Seidel, but is not since this method, whith an appropriate relaxation factor, will converge faster than Gauss-Seidel. For 512 iterations, we use 6.7 KFLOPs.

Problem Two

This problem asks us to find the maximum area of overlap for a heart and a quadrifolium.

Algorithms

I choose to appriximate the area of the overlap using Monte-Carlo simple sampling.

I first generate a table of values for the heart in cartesian coordinates with a translational transformation applied to it. I then use the “poor person’s generator” from the beginning of the course to generate random samples in a box of known area around the heart. The samples are reduced to those which lay inside the heart. The ratio of the total number of samples generated to the number of samples insider the heart is used to approximate the area of the heart. These samples are then converted from a cartesian coordinate system to a polar one and sorted according to theta to make subsequent analysis easier.

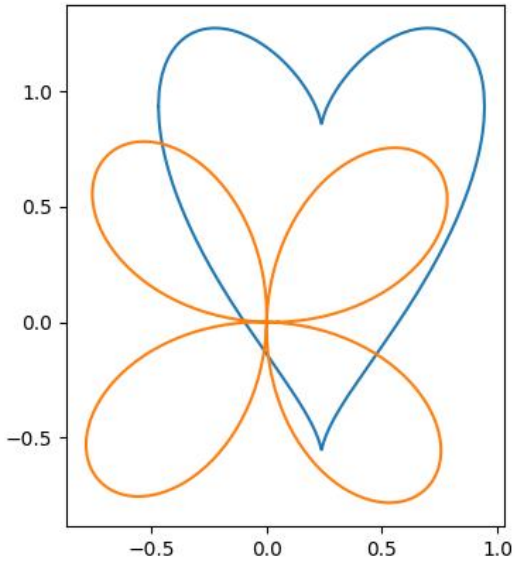
I then generate a table of values for the quadrifolium in polar coordinates with a rotational transformation applied to it. This table is compared with the samples from the heart and only samples which are inside the quadrifolium, that is, samples for which the radial component is less than the quadrifolium’s linearly interpolated radial component, are kept. The ratio of the samples inside of the heart to the samples inside of both the heart and the quadrifolium are used to estimate the overlap.

The translational transformation is applied to the heart because it easier to use cartesian coordinates with the heart and that transformation, and the roational transformation is applied to the quadrifolium because it is easier to use polar coordinates with the quadrifolium and that transformation.

To find the maximum overlap, a three-dimensional grid of theta, x, and y is constructed and iterated over to find the overlap at each point. The grid point with the greatest overlap is selected.

Results

There exists some error in my implementation. For the transformation $(\theta, x, y)=(1, 2, 3)$, my program found a maximum overlap of 0.393398. The program took 1 hour and 45 minutes to run, even with threading using OMP and all complier optimizations enabled. The result of the transformations is given.



Problem Three

Description

This problem has us compare the numerical solution to an implicit ordinary differential equation using two methods and two numbers of subdivisions on the domain $x \in (0,1)$. I choose to use the Euler and Euler-Cromer methods because they were the simpliest to code up.

Algorithms

Euler

The Euler method for approximating solutions to ODEs uses the slope at the current postion to update the state. In equation, that's $y^{(n+1)} = y^{(n)} + y'^{(n)} \Delta x$.

Euler-Cromer

This method is similar to the Euler method, but instead of using the current slope to update the state, it uses the slope at the next position. For this reason, it is also called the implicit Euler method. In equation, $y^{(n+1)} = y^{(n)} + y'^{(n+1)} \Delta x$.

Results

With so many subdivsions (10k and 100k) on such a smal interval (zero to one), it is not suprising that the methods agree very well. All lines are plotted, they overlap each other.