Project 1
FYS4150

Sara Jønvik

8. september 2014

Abstract

In this project we were to write an algorithm to compute the second derivative of a function and compare our results to both the numerical solution and the built-in derivation routine (armadillo in my case). We were also to compare the effectiveness, time-wise, of our code compared to the built-in one. This would hopefully show that it is often more effective to make out own algorithm for time consuming tasks, taylor made for that specific problem, than to rely upon more general algorithms provided by, for example, armadillo.

`https://github.com/saracj/Project-1---FYS4150`

Introduction

We are to show that we can rewrite the given equation as a linear set of equations on the form $A\vec{v} = \vec{b}$ where $A \in \mathbb{R}^{n\times n}$ and $v \in \mathbb{R}^n$. Given:

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{pmatrix}, \ \vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{pmatrix}$$

we get

$$A\vec{v} = \begin{pmatrix} 2 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{n-1} \\ v_n \end{pmatrix}$$

$$= \begin{pmatrix} 2v_1 - v_2 + 0 + 0 + 0 \cdots + 0 \\ -v_1 + 2v_2 - v_3 + 0 + 0 \cdots + 0 \\ 0 - v_2 + 2v_3 - v_4 + 0 \cdots + 0 \\ 0 + 0 + \cdots - v_{n-1} + 2v_n \end{pmatrix}$$

$$= \begin{pmatrix} 2v_1 - v_2 \\ -v_1 + 2v_2 - v_3 \\ -v_2 + 2v_3 - v_4 \\ \vdots \\ -v_{n-2} + 2v_{n-1} - v_n \\ -v_{n-1} + 2v_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix} = \begin{pmatrix} h^2 f_1 \\ h^2 f_2 \\ h^2 f_3 \\ \vdots \\ h^2 f_{n-1} \\ h^2 f_n \end{pmatrix}$$

Which becomes, as we can see, a set of linear equations of the same form as the equation given in the problem text.

Algorithm

Given a tridiagonal matrix we can use a form of gaussian elimination in order to solve for the elements in $\vec{v}$. Performing the matrix operations as above will give us a set of equations on the form:

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i$$

assuming that $a_1 = 0$ and $c_n = 0$ we get three "equations":

$$\begin{aligned} b_1 u_1 + c_1 u_1 &= f_1 && ; i = 1 \\ a_i u_{i-1} + b_i u_i + c_i u_{i+1} &= f_i && ; i = 2, \ldots, n-1 \\ a_n u_n + b_n u_n &= f_n && ; i = n \end{aligned}$$

Using gaussian elimination we are supposed to eliminate all the u_i from the equations for index j with $j < i$. We see from the general equation above that for the first three steps this means:

$$
\begin{aligned}
f_1 - \frac{a_1}{b_0} f_0 &= a_1 u_0 + b_1 u_1 + c_1 u_2 - \left(a_1 u_0 + \frac{a_1 c_0}{b_0} u_1 \right) \\
&= \left(b_1 - \frac{a_1 c_0}{b_0} \right) u_1 + c_1 u_2 \\
&= b_1' u_1 + c_1 u_2 = f_1' \\
f_2 - \frac{a_2}{b_1'} f_1' &= a_2 u_1 + b_2 u_2 + c_2 u_3 - \left(a_2 u_1 + \frac{a_2 c_1}{b_1'} u_2 \right) \\
&= \left(b_1 - \frac{a_2 c_1}{b_1'} \right) u_2 + c_2 u_3 \\
&= b_2' u_2 + c_2 u_3 = f_2' \\
f_3 - \frac{a_3}{b_2'} f_2' &= a_3 u_2 + b_3 u_3 + c_3 u_4 - \left(a_3 u_2 + \frac{a_3 c_2}{b_2'} u_3 \right) \\
&= \left(b_3 - \frac{a_3 c_2}{b_2'} \right) u_3 + c_3 u_4 \\
&= b_3' u_3 + c_3 u_4 = f_3'
\end{aligned}
$$

We see from the above equations that we can obtain recursive formula for the coefficients b_i' and f_i':

$$
f_i' = b_i' u_i + c - i u_{i+1} = f_i - \frac{a_i}{b_{i-1}'} f_{i-1}' \tag{1}
$$

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

which gives us

$$
u_i = \frac{f_i' - c_i u_{i+1}}{b_i'}
$$

The two coefficients above can easily be found by a for-loop since $a_1 = 0$ they only depend on each other at previous indices and. Given $c_n = 0$ we see that we have an equation for u_n which does not depend on u_{n+1}, and from

these equations we can find u_i as a function of u_{i+1} which we can backwards substitute from $i = n$ to $i = 1$.

This algorithm will consist of approximately $8n$ Floating point operations not including the for-loop that caclulates the given doble derivative $h^2 f(x)$ and the analytical solution $u(x)$. The number of flops for a normal gaussian elimination is around $2n^3$ and the LU-decomposition around $\frac{2}{3}n^3$, which sugests that our algorithm should be significantly faster than the two general methods

Implementation

Given the equations for $u(x)$ and $f(x) = u''(x)$ the simple two step algorithm above can solve the resulting set of equations with the following code:

```
h = 1./(n+1); \\Step length
B[1] = b[1];
for(i=0; i<n+2; i++){
    x = i*h;
    //Analytical solution:
    u[i] = 1-(1-exp(-10))*x - exp(-10*x);
    f[i] = h*h*100.*exp(-10*x);
}
F[1] = f[1];
for(i=2; i<n+1; i++){
    factor = (a[i]/B[i-1]);
    B[i] = b[i] - factor*c[i-1];
    F[i] = f[i] - factor*F[i-1];
}
v[n] = F[n]/B[n];
v[n+1] = 0.0;
for(i=n-1; i>=1; i--){
    v[i] = (F[i] - c[i]*v[i+1])/B[i];
}
```

We have her defined v_i as the numerical and u_i as the analytical solution. First we store the unchanged f_i values in an array as well as the analytical solution u. We then compute the coefficients b'_i and f'_i for $i = 2$ to $i = n$, even though our arrays are $n+2$ elements long. We do this in order to cover the boundary values given in the assignment ($v_0 = 0$ and $v_{n+1} = 0$) which can't be covered by the algorithm ($v_n = f'_n/b'_n \neq 0$) since it only covers the internal points of the matrix $i = 1, 2..., n$.

The rest of the code calculates the relative error and picks out the maximum value for all values of n (and prints it to screen), declares and defines variables as well as writing the relevant data to .dat files in order to plot them using python. Since this is a rathaer small program I chose to run through the

values of n with a for-loop. Even with $n = 10000$ this does not take too long, but had the program been bigger and longer I would most likely have chosen to read in the n-values from the command line instead.

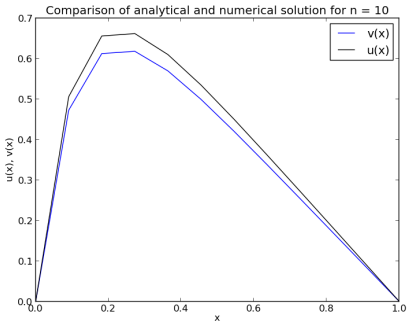
I chose to generate the plots with python's matplotlib.pyplot.

Relative error:

$$\epsilon = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right)$$

Results

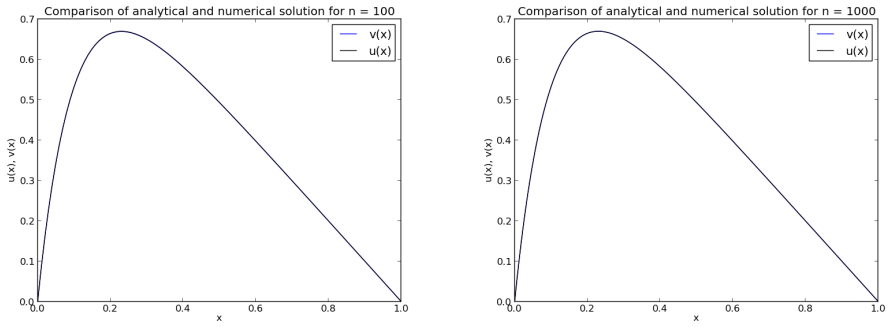
Running the script for $n = 10$ shows that the correlation between the analytical and numerical solution is quite good. This correlation increases with n and with $n = 100$ and $n = 1000$ the two lines overlap each other completely, as seen below.



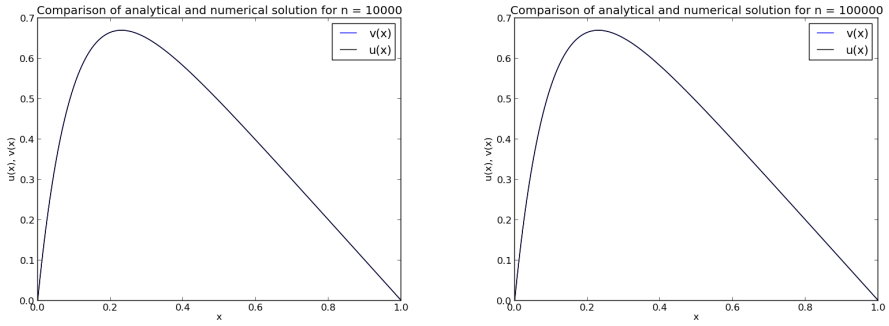
Plot of u and v for $n = 10$

For larger n, $n = 10000$ and $n = 100000$ we see that the lines still overlap, and the relative error is decreasing with increasing n, almost by two powers of 10 for each power of 10 we increase n. The algorithm becomes more accurate with decreasing step length.

n	Maximum value of ϵ
10	-1.1797
100	-3.08804
1000	-5.08005
10000	-7.07929
100000	-8.84297



Figur 1: Plot over u and v for $n = 100$ and $n = 1000$



Figur 2: Plot over u and v for $n = 10000$ and $n = 100000$

Comparisons and comments

For $n = 10$ and $n = 100$ the time measured both for my algorithm and the armadillo LU-decomposition is so small the program prints zero even with TIME defined as a double. For $n = 1000$ and $n = 10000$ my algorithm still measure 0 seconds, while the armadillo's LU-decompostion time has increased to 0.82 seconds at $n = 1000$ and takes several minutes at $n = 10000$. If I try to run the LU-decomp for $\left(10^5 \times 10^5\right)$-matrix i run out of memory, effectively demonstrating that I can not run the LU-dexomp for the $\left(10^5 \times 10^5\right)$-matrix.

Table of computation time:

n	My algorithm	Armadillo's LU-decomp.
10	0	0
100	0	0
1000	0	0.82
10000	0	778.84
100000	0	*unknown*

As we were expecting to see, it is much more effective computation-time-wise, to make a program for the specific problem we are to solve instead of always using the same general method. The time difference we observe for larger n were much greater than I had expected, but demonstrates the importance of keeping the number of floating point operations in mind when constructing an algorithm