Fys4150 Project 1

Aksel Graneng

September 11, 2019

Abstract

This paper looks at solving differential equations as sets of linear equations, and concludes with the importance of writing non-generalized code for matrix operations.

1 Introduction

In this project, we will be solving a differential equation as a set of linear equations. This will be done using algorithms of varying generalization in python.
We will look at just how row reduction can solve a differential equation, as well as the efficiency in different row reduction methods.

2 Theory

2.1 Vectorized second derivative

If we have a 1d data-set on the form:

$$V(\vec{x}) = [v_0,\ v_1,\ \cdots,\ v_{n-1},\ v_n,\ v_{n-1},\ \cdots]$$

Then we can write the second derivative of the data-set as:

$$f_n = -\frac{v_{n+1} + v_{n-1} - 2v_n}{\Delta x^2}$$

Where Δx is the change in variable we are derivating based on; usually time.

Rather than calculating the second derivatives of this data-set individually, we can instead calculate them all at the same time using linear algebra. This can be done by finding a matrix A such that

$$A\vec{V} = \vec{f}$$

Where:

$$\vec{f} = \begin{bmatrix} 2v_i - v_2 \\ -v_1 + 2v_2 - v_3 \\ \vdots \\ -v_{n-1} + 2v_n - v_{n+1} \\ -v_n + 2v_{n+1} - v_{n+2} \\ \vdots \end{bmatrix}$$

As multiplying $\vec{f}$ with $\frac{1}{\Delta x^2}$ would give us an array containing all the second derivatives. We can see that $\mathbf{A}$ must be:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 & \cdots \\ -1 & 2 & -1 & 0 & \cdots \\ 0 & -1 & 2 & -1 & \cdots \\ \vdots & 0 & -1 & 2 & -1 \end{bmatrix}$$

This matrix can then be used to put up a set of linear equations:

$$\mathbf{Av} = \mathbf{f}$$

Where $\mathbf{v}$ contains solutions to the differential equation:

$$-u''(x) = f(x)$$

3 Method

3.1 General linear equation solver

The first thing our program does is solve a set of general linear equations on the form $\mathbf{Av} = \mathbf{f}$. This is done by putting the matrix $\mathbf{M}$ on row reduced echeleon form:

$$\mathbf{M} = \begin{bmatrix} b_1 & c_1 & 0 & 0 & \cdots f_1 \\ a_2 & b_2 & c_2 & 0 & \cdots f_2 \\ 0 & a_3 & b_3 & b_3 & \cdots f_3 \\ \vdots & 0 & a_4 & b_4 & \cdots f_4 \end{bmatrix}$$

Where $\mathbf{M}$ is a $n \times (n+1)$ matrix. This is generally quite simple to solve using forward and backward substitution, but when n becomes very large (in my experience, larger than 10^5) we get problems with memory as well as computation time, as the most simple row reduction algorithm has $2(n+1)n^2$ FLOPS.
To avoid this problem, we use the fact that most of the elements in the matrix is zero. In fact, there are at most 4 elements in each row that isnt zero. By removing every element that is zero from the matrix (except one in for the first and last line), we get:

$$\mathbf{M}^* = \begin{bmatrix} b_1 & c_1 & 0 & f_1 \\ a_2 & b_2 & c_2 & f_2 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & a_n & b_n & f_n \end{bmatrix}$$

So now we have a $n \times 4$ matrix which contains all the information we need.

We now use forward substitution on $\mathbf{M}^*$ just like we would on $\mathbf{M}$, except this time we "roll" the next row every time we subtract, so that b_i is subtracted from a_{i+1} and c_i is subtracted from b_{i+1}. At the same time, we are careful to subtract f_i from f_{i+1}. This means that index notation must be used, rather than numpy.roll.
This is done in a loop that has n iterations. First it does 3 operations, one on each element in the row except for the first (which is zero), along with a division, to normalize it. Then it does 3 subtraction and multiplication operations on the next row.

Next is backwards substitution. Again we use a loop with n iterations. Here we only do 2 subtractions and multiplications, on the last and second last element of each row.
All in all, this sums up to $14n$ FLOPS.

After it was done, the general linear equation solver was tested on:

$$u''(x) = f(x)$$

$$f(x) = 100e^{-10x}$$

Which has a closed-form solution:

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

3.2 Specific linear equation solver

The matrix we are working with in this project is very simple. All values of a, all values of b and all values of c are equal. That is to say:

$$\text{for any } i \wedge j \in [1,\ n],\ a_i = a_j,\ b_i = b_j,\ c_i = c_j$$

This means that we are practically doing the same operations n times when we are row reducing our matrix. Rather than doing this, we can limit our algorithm to only operate on our values for f, as these are unknown. We simply need to look at what the row reduction algorithm does to the matrix to see what it does to f, and then skip right to that part when calculating it numerically.

Studying the forward substitution, we see, starting with $n = 2$:

$$f_n^* = \frac{n}{n+1}(f_n - f_{n-1}^*)$$

Thus, the n'th row ends up as (excluding all the zeros):

$$[0,\ 1,\ -\frac{n}{n+1},\ f_n^*]$$

The very last row ends up as (with $n =$ size of f):

$$[0,\ 0,\ 1,\ -(f_{-1} - f_{-2}^*)\frac{n}{n+1}]$$

The first value of f (n = 1) ends up as:

$$f_1^* = \frac{1}{2}f_1$$

And then we look at backwards substitution. We can see that we subtract:

$$\frac{n-i}{n-i-1}f_{-(i-1)}^{**}$$

from f_{-i}^* to find our final value f_{-i}^{**}
The process of forward substitution ends with the last line being:

$$f_1^* = f_1^* + 2f_2^{**}$$

The specified algorithm for solving the linear equations is a lot more efficient than the general one. Assuming we have already calculated the required numbers used in finding f, the forwards and backwards substitutions each require 2 operations, resulting in a total of $4n$ FLOPS.

3.3 LU Decomposition, numpy.linalg.solve

We should be comparing our results to what we get from using a built-in LU-decomposition method. This seems to require some work in python, so instead of using numpy.linalg.lu and doing calculations, we have opted to use numpy.linalg.solve, which fully solves the set of linear equations.

4 Results

4.1 General linear equation solver

We tested the general linear equation solver for datapoints $n = 10,\ n = 100,\ n = 1000$

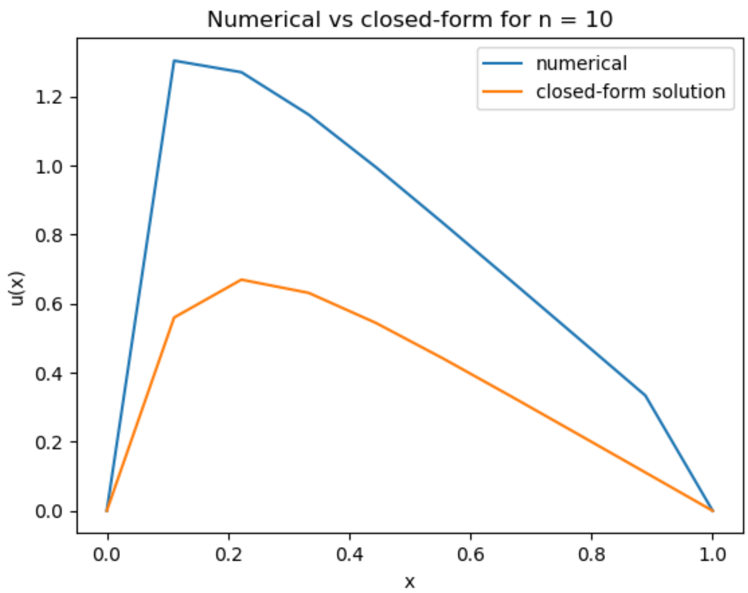

Figure 1: The numerical solution vs the closed-form solution for 10 data-points. Here we can see that they deviate massively.

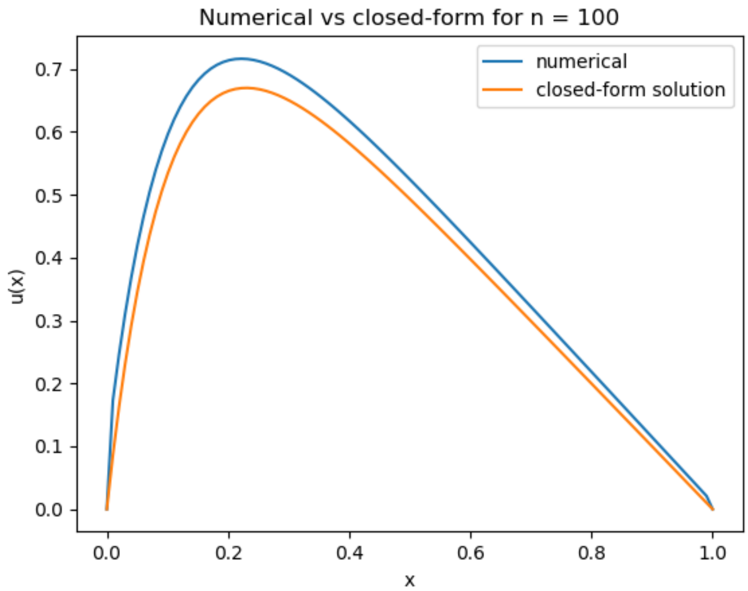

Figure 2: The numerical solution vs the closed-form solution for 100 data-points. Here we can see that the difference is decreasing.

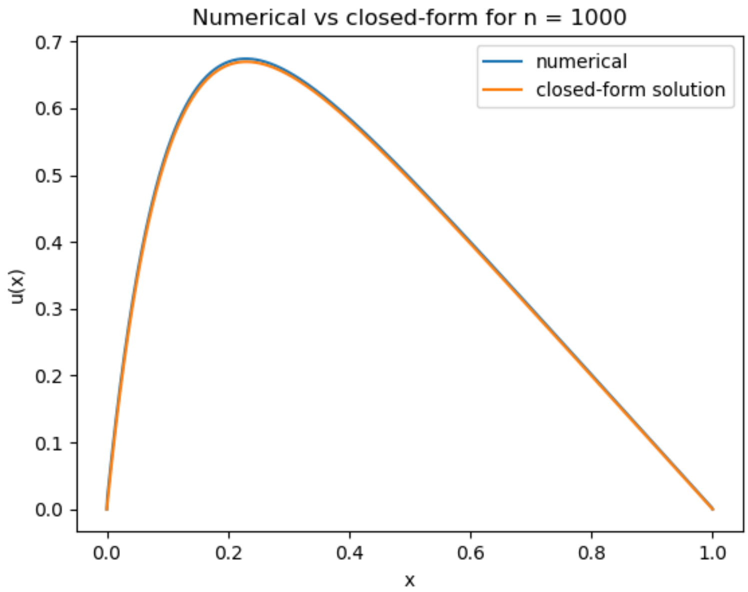

Figure 3: The numerical solution vs the closed-form solution for 100 data-points. Here we can see that the difference has been even more decreased, and it is becoming hard to tell them apart.

4.2 Computation time test

The general linear equation solver method took 6.4 seconds to compute 10^6 data points.
The specific linear equation solver method took 1.0 seconds to compute 10^6 data points.

4.3 Relative error

Using the specific linear equation solver method for data points ranging from 10 to 10^7, we got these relative errors:

n	10^1	10^2	10^3	10^4	10^5	10^6	10^7
$\epsilon_{\max}$	10^{-1}	$5.3 \cdot 10^{-3}$	$4.9 \cdot 10^{-4}$	$4.8 \cdot 10^{-5}$	$4.8 \cdot 10^{-6}$	$4.8 \cdot 10^{-7}$	$4.8 \cdot 10{-}8$

Table 1: The maximal relative error from the specific linear equation solver. ϵ is $\log_{10}(|\frac{v-u}{u}|)$ where v is the calculated value and u is the closed-form solution.

4.4 Our method vs numpy.linalg.solve

Solving the linear equation systems using numpy.linalg.solve and our method for different amounts of data points results in:

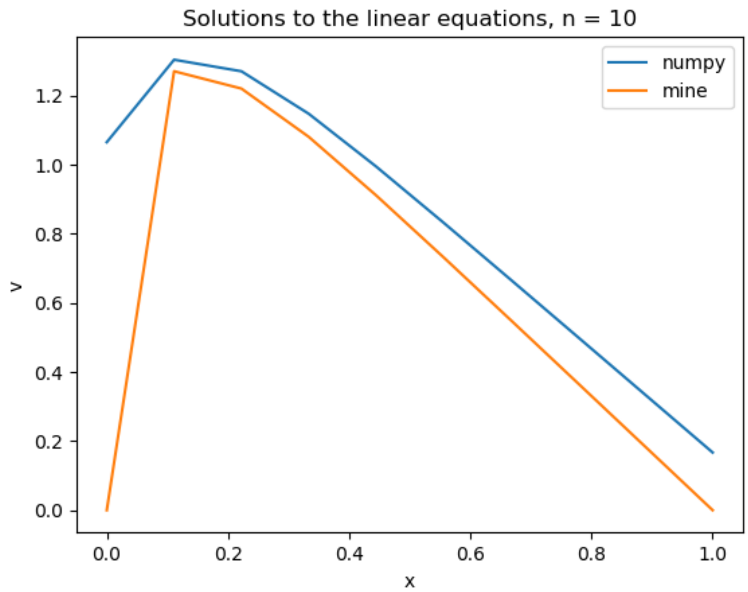

Figure 4: Numpy solution vs my solution. They already seem similair.

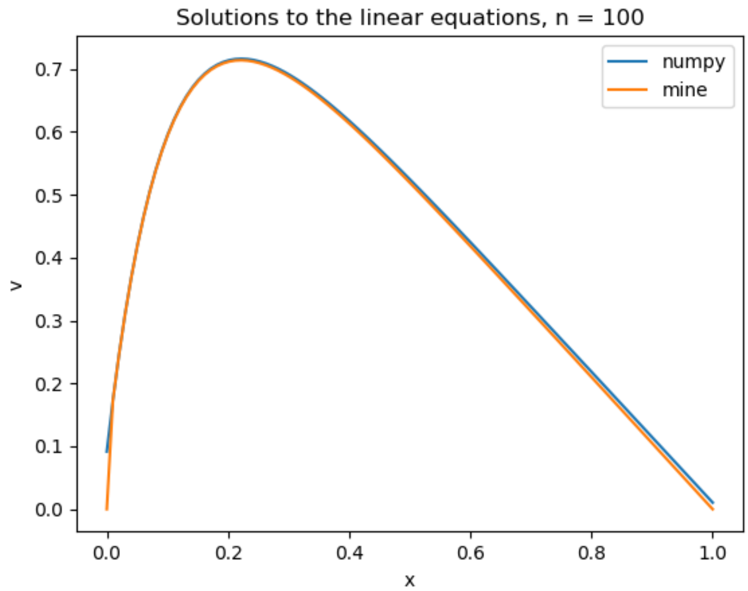

Figure 5: Numpy solution vs my solution. It is becoming harder to tell them apart.

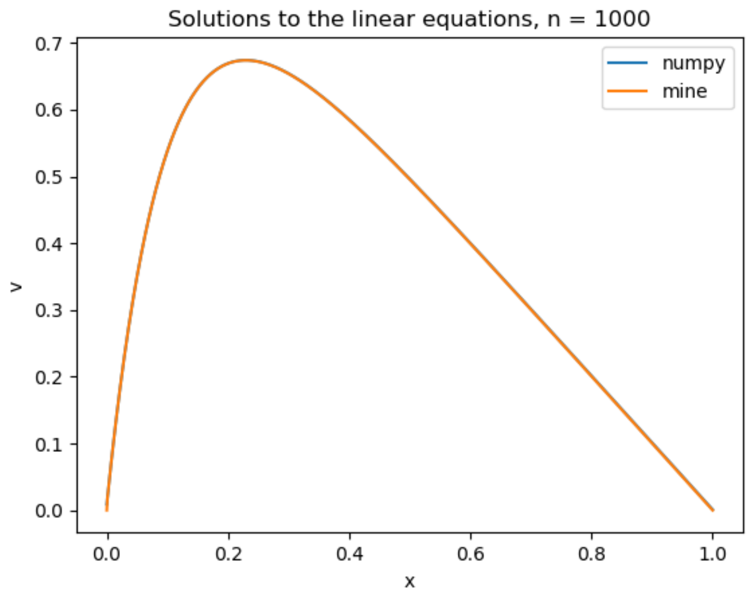

Figure 6: Numpy solution vs my solution. It is now impossible to tell them apart.

The elapsed time was:

Method	$n = 100$	$n = 1000$
numpy	0.001 s	0.015 s
mine	0.00097 s	0.010 s

Table 2: Computation time for the 2 methods.

5 Discussion

5.1 Row reduction methods.

The easiest, and least efficient, way to implement row reduction seems to be operating directly on a whole matrix using matrix notation. For this project, that was definitely not viable as we did not have enough memory for matrices larger than $10^4 \times 10^4$.
Removing all the zeros and operating on a $n \times 4$ matrix was also quite simple, and required a lot less computation time and memory. It is a shame work was not started on that immediately, as the memory problem was not realised until using $10^5 \times 10^5$ matrices was attempted and the first implementation took some time.

Making the specified linear equation solver was by far the biggest task this project. The immediate reaction was to find a way to omit any calculations that did not directly apply to $\vec{f}$. The implementation of this eluded success for quite a while, however. This resulted in the implementation of the method "slow_specific_linear_equation_solver", which follows the same principle as the method "linear_equation_solver". Later, implementation of the original idea was attempted once again, and succeded.
The method "fast_specific_linear_equation_solver" seems to be around 5 times faster than the general one. This seems inconsistent with our calculation of FLOPS. This might be because we counted division to be 1 FLOP, while it is actually more costly than that.

5.2 Relative Error

From Table 1, it looks like solving the differential equation using row reduction is a first order method, causing the error to be reduced by a factor of 10 when the amount of data-points is increased by a factor of 10.

5.3 LU decomposition

We can see that the time elapsed inceases faster for the numpy method than for our method. This is no surprise, as the numpy method operates on the whole matrix. The numpy method assumedly has $\frac{2}{3}n^3$ FLOPS. It is impossible to use it on a matrix as large as $10^5 \times 10^5$ as our computer lacks the memory.

6 Conclusion

From these experiments, we see the importance of making less generalized code when operating on very large matrices, as this can cut down on both memory usage and computation time.

7 Appendix

This is the code used in the project.

```python
import numpy as np
import matplotlib.pyplot as plt
from time import time


class Project_solver():

    def __init__(self, f, a = -1, b = 2, c = -1, M = True):
        self.n = len(f)
        self.h = 1/(self.n + 1)
        self.f = f*self.h**2
        self.a = a
        self.b = b
        self.c = c
        if M == True:
            self.builder()
        else:
            self.M = M
    def builder(self):
        """
        Builds a matrix M
        Used in the linear transformation
        Mv = f
        """
        M = np.zeros((self.n, 4))
        M[0, :] = np.array([self.b, self.c, 0, self.f[0]])
        for i in range(1, self.n-1):
            M[i, :] = np.array([self.a, self.b, self.c, self.f[i]])
        M[-1, :] = np.array([0, self.a, self.b, self.f[-1]])
        self.M = M
        self.M[:, :-1] = self.M[:, :-1]


    def linear_equation_solver(self):
        """
        Solves Mv = f for v
        using forwards and backwards substitution
        """
        self.M[0, :] = self.M[0, :]/self.M[0, 0]
        self.M[1, :] = self.M[1, :] - self.M[0, :]*self.M[1, 0]
        for i in range(1, self.n-2):
            self.M[i, 1:] = self.M[i, 1:]/self.M[i, 1]

            multi_comp = self.M[i+1, 0]
            for j in range(2):
                self.M[i+1, j] = self.M[i+1, j] - self.M[i, j+1]*multi_comp
            self.M[i+1, -1] = self.M[i+1, -1] - self.M[i, -1]*multi_comp
        self.M[-2, :] = self.M[-2, :]/self.M[-2, 1]
        self.M[-1, :] = self.M[-1, :] - self.M[-2, :]*self.M[-1, 1]
        self.M[-1, :] = self.M[-1, :]/self.M[-1, -2]


        self.M[-2, :] = self.M[-2, :] - self.M[-1, :]*self.M[-2, -2]

```

```
        self.M[0, -1] = 0
        self.thingyjing = self.M[:, -1]
        for i in range(1, self.n-1):
            multi_comp = self.M[-(i+1), -2]
            self.M[-(i+1), -2] = \
            self.M[-(i+1), -2] - self.M[-i, -3]*multi_comp
            self.M[-(i+1), -1] = self.M[-(i+1), -1] - self.M[-i, -1]*multi_comp

        self.M[0, :] = self.M[0, :] - self.M[1, :]*self.M[0, 1]
        self.v = self.M[:, -1]
        self.M[-1, -1] = 0
        self.M[0, -1] = 0
        self.v = self.M[:, -1]


    def slow_specific_linear_equation_solver(self):
        """
        Solves Mv = f for v
        using the fact that the diagonal elements in M are all equal.
        More generalized.
        """

        self.M[0, :] = self.M[0, :]/self.M[0, 0]
        self.M[1, :] = self.M[1, :] - self.M[0, :]*self.M[1, 0]

        for i in range(1, self.n-2):
            self.M[i, 2:] = self.M[i, 2:]/self.M[i, 1]
            self.M[i+1, 1] = self.M[i+1, 1] + self.M[i, 2]
            self.M[i+1, -1] = self.M[i+1, -1] + self.M[i, -1]

        self.M[-2, :] = self.M[-2, :]/self.M[-2, 1]
        self.M[-1, :] = self.M[-1, :] - self.M[-2, :]
        self.M[-1, 2:] = self.M[-1, 2:]/self.M[-1, -2]
        self.M[0, -1] = 0
        self.M[-1, -1] = 0
        for i in range(1, self.n-1):
            self.M[-(i+1), -1] = self.M[-(i+1), -1] - \
            self.M[-i, -1]*self.M[-(i+1), -2]

        self.M[-1, -1] = 0


    def relative_error_finder(self):
        """
        finds the relative error epsilon
        for the specific linear equation solver
        epsilon = log10((v - u)/u)
        """
        i = np.arange(1, self.n+1)
        numbers = (i)/(i+1)
        self.fast_specific_linear_equation_solver(numbers)
        x = np.linspace(0, 1, self.n)
        u = (1 - (1 - np.exp(-10))*x - np.exp(-10*x))
        v = self.v

        self.epsilon = np.log10(np.abs((v[1:-1] - u[1:-1])/u[1:-1]))


    def fast_specific_linear_equation_solver(self, numbers):
        """
```

```
        Solves Mv = f for v
        for the matrix
        [[2, -1, ...,          f]
         [-1, 2, -1, ...,      f]
          [..., -1, 2, -1, ..., f]
          [...0, 2, -1,         f]]
        using a list of numbers containing what to do with f.
        """

        v = self.f
        v[0] = v[0]*numbers[0]
        for i in range(1, self.n-1):
            v[i] = (v[i] + v[i-1])*numbers[i]
        v[-1] = 0
        for i in range(1, self.n-1):
            v[-(i+1)] = v[-(i+1)] + v[-(i)]*numbers[-(i+1)]
        v[0] = 0
        self.v = v
if __name__ == "__main__":
    def trueplot(x):
        return(1 - (1 - np.exp(-10))*x - np.exp(-10*x))

    def linear_equation_tester():
        n_list = [int(1e1), int(1e2), int(1e3)]
        for n in n_list:
            x = np.linspace(0, 1, n)
            f = 100*np.exp(-10*x)
            objekt = Project_solver(f)
            objekt.linear_equation_solver()
            plt.plot(x, objekt.v)
            plt.plot(x, trueplot(x))
            plt.legend(["numerical", "closed-form solution"])
            plt.title("Numerical vs closed-form for n = %g" % n)
            plt.xlabel("x")
            plt.ylabel("u(x)")
            plt.show()

    def computation_time_test():
        n = int(1e6)
        i = np.arange(1, n+1)
        numbers = i/(i+1)

        x = np.linspace(0, 1, n)
        f = 100*np.exp(-10*x)

        objekt = Project_solver(f)

        start = time()
        objekt.linear_equation_solver()
        stop1 = time()
        objekt.fast_specific_linear_equation_solver(numbers)
        stop2 = time()

        print("The first method took %g seconds" % (stop1 - start))
        print("The second method took %g seconds" % (stop2 - stop1))
        print((stop1 - start)/(stop2 - stop1))





    def relative_error_tester():
```

```
    ns = [10, 100, 1000, 10000, 100000, 1000000, 10000000]
    error_max = []
    for n in ns:
        x = np.linspace(0, 1, n)
        f = 100*np.exp(-10*x)

        objekt = Project_solver(f)
        objekt.relative_error_finder()
        error_max.append(np.max(objekt.epsilon))
    print(error_max)


def LU_tester():
    ns = [10, 100, 1000]
    a = -1
    b = 2
    c = -1

    for n in ns:
        x = np.linspace(0, 1, n)
        f = 100*np.exp(-10*x)

        M = np.zeros((n, n))
        M[0, 0] = b
        M[0, 1] = c
        for i in range(1, n-1):
            M[i, i-1] = a
            M[i, i] = b
            M[i, i+1] = c
        M[-1, -2] = a
        M[-1, -1] = b

        h = 1/(n+1)
        start = time()
        v_np = np.linalg.solve(M, f*h**2)
        stop1 = time()
        objekt = Project_solver(f)
        objekt.slow_specific_linear_equation_solver()
        stop2 = time()
        v_mine = objekt.M[:, -1]

        plt.plot(x, v_np)
        plt.plot(x, v_mine)
        plt.legend(["numpy", "mine"])
        plt.xlabel("x")
        plt.ylabel("v")
        plt.title("Solutions to the linear equations, n = %g" % n)
        plt.show()
        print(stop1 - start)
        print(stop2 - stop1)
LU_tester()
#relative_error_tester()
#computation_time_test()
```