**Machine Epsilon and matrix-solving methods**

**Author: Damian Bukowski**

**Task 1.**

Machine epsilon can be defined as a minimal positive machine floating-point number *g* satisfying the relation *fl(1 + g) > 1*, i.e.,



MATLAB uses double precision for any variable, which means each number will be written on 53 bits (from which one is implicit) and according to IEEE 754 standard, the errors of results from basic operations (addition, subtraction, multiplication and division) must be bound from above by machine epsilon. Said machine epsilon for floating-point numbers in 64 bits is 2-52 which is about 2.22 \* 10-16.

**Code:**

function machEps()

x = 1;

while 1 + x/2 ~= 1

x = x/2;

end

disp(x);

end

**Output:**

****

**Task 2.**

Gaussian elimination is an algorithm, where we reduce the matrix to row echelon form (using elementary row operations) in order to solve the system of linear equations. Additionally, we can add an additional step and add “pivot” element, which is the maximum modulo of the elements under the diagonal in each column. This allows us to switch rows of the diagonal value and the pivot element, which ultimately allows us to solve matrices with 0’s on the diagonal.

The solutions for n = 10 are:

|  |  |
| --- | --- |
| **Subpoint A** | **Subpoint B (\*10^11)** |
| 0.1822 | 0.0000 |
| 0.1579 | -0.0013 |
| 0.2046 | 0.0273 |
| 0.2272 | -0.2484 |
| 0.2570 | 1.1838 |
| 0.2876 | -3.2527 |
| 0.3083 | 5.3350 |
| 0.3603 | -5.1545 |
| 0.3207 | 2.7056 |
| 0.5475 | -0.5949 |

Solutions’ errors are:

|  |  |
| --- | --- |
| **Subpoint A (\*10^-15)** | **Subpoint B (\*10^-5)** |
| 0 | -0.381469726562500 |
| 0 | -0.190734863281250 |
| 0 | -0.011920928955078 |
| -0.444089209850063 | 0.026396342656110 |
| 0 | 0.000384693306377 |
| 0 | -0.000029335709556 |
| 0 | -0.000001278224748 |
| 0 | 0.000000002303713 |
| 0 | 0.000000000001388 |
| 0 | 0 |

After applying residual correction method:

|  |  |
| --- | --- |
| **Subpoint A** | **Subpoint B (\*10^23)** |
| 0.1822 | 0.000014936625536 |
| 0.1579 | -0.001286334938919 |
| 0.2046 | 0.027348597237416 |
| 0.2272 | -0.248355787783294 |
| 0.2570 | 1.183790878256740 |
| 0.2876 | -3.252735933558774 |
| 0.3083 | 5.335041250629875 |
| 0.3603 | -5.154518080518682 |
| 0.3207 | 2.705621208264839 |
| 0.5475 | -0.594931100626001 |

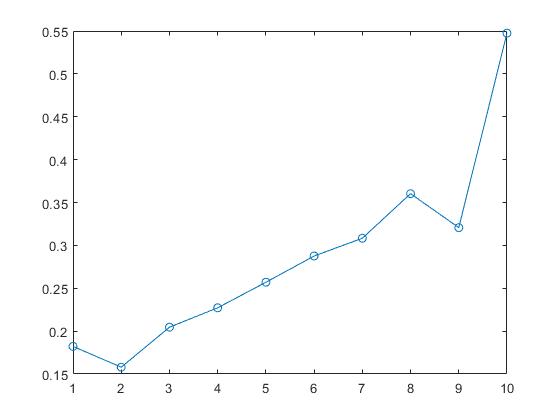


Figure 1 X – iteration, Y - solution

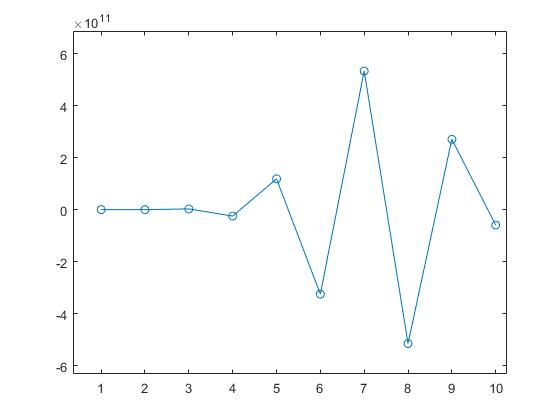


Figure 2 X - iteration, Y - solution

From foregoing tables, we can inquire that the residual correction slightly improves the solutions in sub-point A. On the other hand, in sub-point B the values “jump” from high to low, which probably means that the pivoting is not a correct method to solve the aforementioned system.

Chart

Description automatically generated

Figure 3 Norm chart for sub-point A; X – number of iterations, Y – norm

Chart, line chart

Description automatically generated

Figure 4 Norm chart for sub-point B; X – number of iterations, Y – norm

In sub-point A the norm (and simultaneously the errors) increase with the number of iterations, since it’s based on the number of operations.

**Task 3**

In this task we are supposed to write a program that will solve a given matrix using Gauss-Seidel method and Jacobi method. In case on Jacobi’s method, first we decompose the main matrix into 3 submatrices: sub-diagonal (L), and diagonal (D) and one with the values above the diagonal (U). We assume that D is non-singular. Then, we can proceed to solve the problem with the following formula:

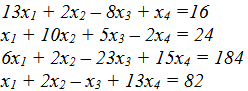


Gauss-Seidel method is similar, we still assume D is non-singular, except this time a single iteration of the method is written in a form:



Overall, the difference between Gauss-Seidel method and Jacobi method is that Jacobi method can consists of *n* independent scalar equations, meaning they can be solved in parallel, totally or partially. Gauss-Seidel on the other hand, has to be performed in sequence, because each subsequent equation depends on the previous one. The Gauss-Seidel method, in general, is faster than Jacobi’s method.

When applied to the system:



The program yields following results:

X1 = -3.12097342

X2 = 6.19173871

X3 = -4.87108549

X4 = 5.22049312

It took Gauss-Seidel method 20 iterations with the norm of the residuum being 9.952845103873589.

Jacobi method managed to solve the system in 21 iterations with the norm 9.952845103860810.

From this we can deduce that the Gauss-Seidel method is in fact faster but yields a less accurate result.

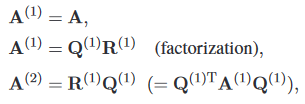
Solving the equation from task 2 using Jacobi method gives the following results:

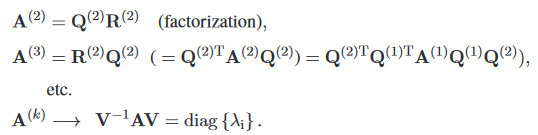
|  |
| --- |
| **Subpoint A (49 iterations)** |
| 0.18217762 |
| 0.15792275 |
| 0.20457346 |
| 0.22721351 |
| 0.25698264 |
| 0.28759291 |
| 0.30834041 |
| 0.36030077 |
| 0.32068209 |

For sub-point B, program using Jacobi method increases the solutions to infinity, hence this method (as well as Gauss-Seidel) cannot be used to calculate the matrix from sub-point B.

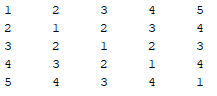
**Task 4**

In this task we are supposed to write a program that will find eigenvalues for any symmetric 5x5 matrix using QR method with and without shifts. This method consists of two main elements: QR factorization and the algorithm itself. The QR factorization leaves us with two matrices orthogonal (Q) and upper triangular (R). Having these two we can proceed to finding the eigenvalues. The algorithm goes as follows (without shifts):





For initial matrix:



Algorithm without shifts takes 18 iteration’s and returns eigenvalues:

14.086258192781935

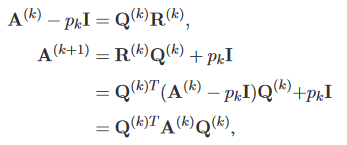
-4.554928944217567

-2.887218071512920

-1.088860757176658

-0.555250419874788

A single iteration of the algorithm with shifts:



Final matrices:

13.6000 2.0113 1.0011 1.6820 0.1432

2.0113 -0.5300 -0.2054 0.0614 0.0126

1.0011 -0.2054 -1.1973 -1.0495 -0.0718

1.6820 0.0614 -1.0495 -3.9670 -0.1506

0.1432 0.0126 -0.0718 -0.1506 -2.9057

14.0803 0.2410 0.1016 0.1715 0.0000

0.2410 -0.7343 -0.3015 -0.1323 -0.0000

0.1016 -0.3015 -1.2386 -1.0283 -0.0001

0.1715 -0.1323 -1.0283 -4.2201 -0.0002

0.0000 -0.0000 -0.0001 -0.0002 -2.8872

14.0862 0.0290 0.0102 0.0170 0.0000

0.0290 -0.6791 -0.2585 -0.1045 -0.0000

0.0102 -0.2585 -1.2349 -0.9323 0.0000

0.0170 -0.1045 -0.9323 -4.2850 -0.0000

0.0000 -0.0000 -0.0000 -0.0000 -2.8872

14.0863 0.0058 0.0010 0.0000

0.0058 -0.6476 -0.2018 -0.0002

0.0010 -0.2018 -0.9965 -0.0022

0.0000 -0.0002 -0.0022 -4.5549

14.0863 0.0012 0.0002 0.0000

0.0012 -0.6277 -0.1828 -0.0000

0.0002 -0.1828 -1.0164 -0.0000

0.0000 -0.0000 -0.0000 -4.5549

14.0863 0.0000 0.0000

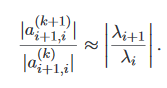
0.0000 -0.5553 -0.0000

0.0000 -0.0000 -1.0889

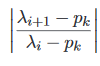
14.0863 -0.0000

0.0000 -0.5553

This version of the algorithm takes 2 iterations to return the same eigenvalues for the same matrix.

The reason for such discrepancy in the amount of iterations is the fact that if the eigenvalues are similar, then the ratio of convergence will be very slow. It calculated from the following equation: 

Where *a* denotes an off-diagonal element and λ­i is the eigenvalue for certain iteration *i*. When we apply the shift *pk*, the convergence ratio will change to:



Where *pk* is an estimate of λi+1. The way to chose it is to take the lower right corner 2x2 matrix and take either dn (last value on the diagonal) or a closer to dn eigen value from said matrix as the shift. Then the procedure is repeated until dk = λn. Next, the procedure can be repeated for the (n-1)x(n-1) matrix after dropping the last row and column corresponding to the obtained eigenvalue.

**Sources**

* “Numerical Methods” by Piotr Tatjewski