

3/ Gauss Elimination with Partial Pivoting

$$a_{00}^{(2)} x_0 + a_{01}^{(2)} x_1 + \dots + a_{0n}^{(2)} x_n = b_0^{(2)}$$

$$a_{11}^{(n)} x_1 + \dots + a_{1n}^{(n)} x_n = b_1^{(n)}$$

.....

$$a_{nn}^{(n)} x_n = b_n^{(n)}$$

1. Source code

1.1 Collecting user input

Gauss.txt is to be filled with data in the following order:

1st line: number of equations

Next lines: $a[0][0]$ $a[0][1]$ $b[0]$ and so on

Problems and solutions:

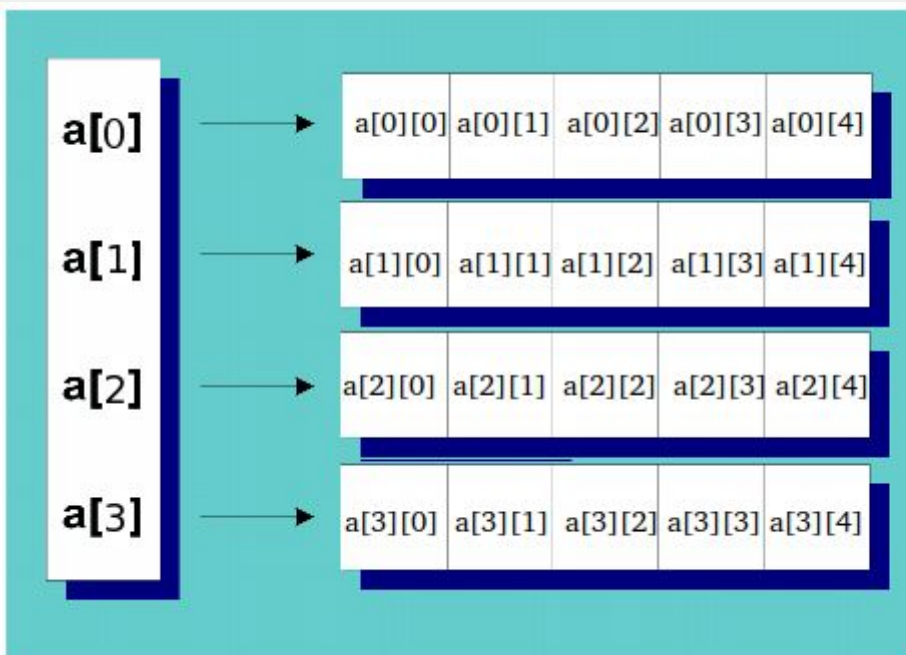
a) Using a dynamic 2D array and not getting an error.

Solution:

A dynamic 2D array is basically an array of *pointers to arrays*. You can initialize it using a loop, like this:

```
int** a = new int*[rowCount]; for(int i = 0; i < rowCount; ++i) a[i] = new int[colCount];
```

The above, for $colCount = 5$ and $rowCount = 4$, would produce the following:




```

ifstream fin("gauss.txt");
fin >> n;

a = new float*[n];
for (int i = 0; i < n; ++i)
    a[i] = new float[n+1];

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n+1; j++) {
        fin >> a[i][j];
        if (j > n-1) cout << "| ";
        cout << a[i][j] << " ";
    }
    cout << "\n";
}

```

 Microsoft Visual Studio Debug Console

```

3  2  -4  |  3
2  3   3  | 15
5  -3   1  | 14

```

Also - displays the matrix in console window.

1.2 Partial pivotisation

Comparing elements vertically and swapping them if the pivot element is smaller.

```

for ( int i=0;i<n;i++)
    for (int k=i;k<n;k++)
        if (a[i][i]<a[k][i])
            for (int j = 0; j <= n; j++) {
                double temp = a[i][j];
                a[i][j] = a[k][j];
                a[k][j] = temp;
            }

```

1.3 Gauss elimination

```

//gauss elimination
for (int i =0; i<n-1;i++) // not going to the last row
    for (int k = i + 1; k < n; k++)
    {
        double mult = a[k][i] / a[i][i];
        for (int j = 0; j <= n; j++)
            a[k][j] = a[k][j] - mult * a[i][j]; //make the elements below the pivot equal to zero.
    }

```

1.4 Back substitution.

```
for (int i = n - 1; i >= 0; i--)
{
    x[i] = a[i][n];           //x[i] is now the right side of equation in line [i]
    for (int j = i + 1; j < n; j++)
        if (j != i)           //subtract all left side numbers except the coefficient of the variable whose value is being calculated
            x[i] = x[i] - a[i][j] * x[j];
    x[i] = x[i] / a[i][i];     //divide the right side of equation by the coefficient of the variable to be calculated
}
```

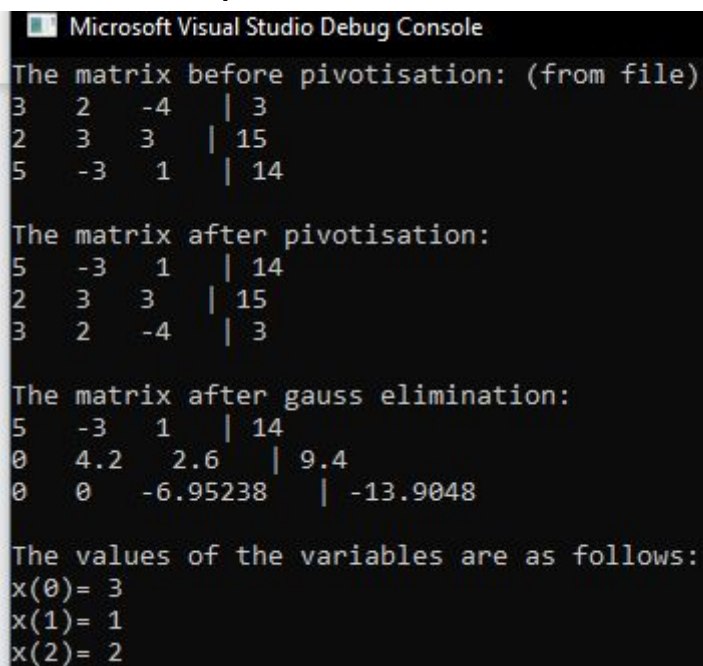
1.5 Results.

```
cout << "\nThe values of the variables are as follows:\n";
for (int i = 0; i < n; i++)
    cout << "x(" << i << ") = " << x[i] << endl;
```

2. Examples

Program displays the matrix after every operation as shown on the screen below:

Matrix 3x3 example:



```
Microsoft Visual Studio Debug Console

The matrix before pivotisation: (from file)
3   2   -4   |   3
2   3    3   |  15
5  -3    1   |  14

The matrix after pivotisation:
5  -3    1   |  14
2   3    3   |  15
3   2   -4   |   3

The matrix after gauss elimination:
5  -3    1   |  14
0  4.2  2.6   |  9.4
0   0 -6.95238 | -13.9048

The values of the variables are as follows:
x(0)= 3
x(1)= 1
x(2)= 2
```

Matrix 4x4 example:

Microsoft Visual Studio Debug Console

The matrix before pivotisation: (from file)

0	2	-4	1		2
2	3	3	3		15
5	-3	1	1		14
2	-1	0	1		12

The matrix after pivotisation:

5	-3	1	1		14
2	3	3	3		15
2	-1	0	1		12
0	2	-4	1		2

The matrix after gauss elimination:

5	-3	1	1		14
0	4.2	2.6	2.6		9.4
0	0	-0.52381	0.47619		5.95238
0	0	0	-5		-62

The values of the variables are as follows:

x(0)= -2.89091
x(1)= -5.38182
x(2)= -0.0909091
x(3)= 12.4

4/ Jacobi

Additional source: <https://www3.nd.edu/~zxu2/acms40390F12/Lec-7.3.pdf>

1. Jacobi method overview

Two assumptions made on Jacobi Method:

1. The system given by

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots a_{nn}x_n &= b_n \end{aligned}$$

has a unique solution.

2. The coefficient matrix A has no zeros on its main diagonal, namely, a_{11} , a_{22} , a_{nn} , are nonzeros

Main idea of Jacobi:

To begin, **solve**:

the 1st equation for x_1

$$x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - \cdots a_{1n}x_n)$$

The 2nd equation for x_2

$$x_2 = \frac{1}{a_{22}}(b_2 - a_{21}x_1 - a_{23}x_3 - \cdots a_{2n}x_n)$$

\vdots

$$x_n = \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - \cdots a_{n,n-1}x_{n-1})$$

...and so on

To obtain the rewritten equations.

Then, **make a initial guess of the solution**

$$x^{(0)} = (x_1^{(0)}, x_2^{(0)}, x_3^{(0)}, \dots x_n^{(0)})$$

Substitute these values(initial guess) into the right hand side of the rewritten equations to obtain *the first approximation*.

$$\tilde{x}^{(1)} = (\tilde{x}_1^{(1)}, \tilde{x}_2^{(1)}, \tilde{x}_3^{(1)}, \dots \tilde{x}_n^{(1)})$$

This accomplishes one iteration.

The second approximation is computed by **substituting the first approximation's x-values into the right hand side of the**

$$(x_1^{(2)}, x_2^{(2)}, x_3^{(2)}, \dots x_n^{(2)})$$

rewritten equations.

Then, continue with next iterations...

The sequence of approximations:

$$x^{(k)} = (x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, \dots x_n^{(k)})^t, \quad k = 1, 2, 3, \dots$$

Real example:

$$\begin{aligned} 5x_1 - 2x_2 + 3x_3 &= -1 \\ -3x_1 + 9x_2 + x_3 &= 2 \\ 2x_1 - x_2 - 7x_3 &= 3 \end{aligned}$$

Rewritten equations:

$$\begin{aligned} x_1 &= \frac{-1}{5} + \frac{2}{5}x_2 - \frac{3}{5}x_3 \\ x_2 &= \frac{2}{9} + \frac{3}{9}x_1 - \frac{1}{9}x_3 \\ x_3 &= -\frac{3}{7} + \frac{2}{7}x_1 - \frac{1}{7}x_2 \end{aligned}$$

Make a initial guess:

the initial guess $x_1 = 0, x_2 = 0, x_3 = 0$

The first approximation:

$$\begin{aligned} x_1^{(1)} &= \frac{-1}{5} + \frac{2}{5}(0) - \frac{3}{5}(0) = -0.200 \\ x_2^{(1)} &= \frac{2}{9} + \frac{3}{9}(0) - \frac{1}{9}(0) = 0.222 \\ x_3^{(1)} &= -\frac{3}{7} + \frac{2}{7}(0) - \frac{1}{7}(0) = -0.429 \end{aligned}$$

Continue iteration for $k=2,3,\dots$

n	$k = 0$	$k = 1$	$k = 2$	$k = 3$
$x_1^{(k)}$	0.000	-0.200	0.146	0.192
$x_2^{(k)}$	0.000	0.222	0.203	0.328
$x_3^{(k)}$	0.000	-0.429	-0.517	-0.416

To sum up - unambiguous method notation:

For each $k \geq 1$, generate the components $x_i^{(k)}$ of $\mathbf{x}^{(k)}$ from $\mathbf{x}^{(k-1)}$ by

$$x_i^{(k)} = \frac{1}{a_{ii}} \left[\sum_{\substack{j=1, \\ j \neq i}}^n (-a_{ij} x_j^{(k-1)}) + b_i \right], \quad \text{for } i = 1, 2, \dots, n$$

Matrix Form:

$n \times n$ size matrix:

$$A\mathbf{x} = \mathbf{b} \text{ with } A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \text{ for } \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

We split A into

$$A = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix} - \begin{bmatrix} 0 & \dots & 0 & 0 \\ -a_{21} & \dots & 0 & 0 \\ \vdots & & \ddots & \vdots \\ -a_{n1} & \dots & -a_{n,n-1} & 0 \end{bmatrix} - \begin{bmatrix} 0 & -a_{12} & \dots & -a_{1n} \\ 0 & 0 & & \vdots \\ \vdots & \vdots & \ddots & -a_{n-1,n} \\ 0 & 0 & \dots & 0 \end{bmatrix} = D - L - U$$

$A\mathbf{x} = \mathbf{b}$ is transformed into $(D - L - U)\mathbf{x} = \mathbf{b}$

$$D\mathbf{x} = (L + U)\mathbf{x} + \mathbf{b}$$

Assume D^{-1} exists and $D^{-1} = \begin{bmatrix} \frac{1}{a_{11}} & 0 & \dots & 0 \\ 0 & \frac{1}{a_{22}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{a_{nn}} \end{bmatrix}$

Then

$$\mathbf{x} = D^{-1}(L + U)\mathbf{x} + D^{-1}\mathbf{b}$$

The matrix form of Jacobi iterative method is

$$\mathbf{x}^{(k)} = D^{-1}(L + U)\mathbf{x}^{(k-1)} + D^{-1}\mathbf{b} \quad k = 1, 2, 3, \dots$$

Define $T = D^{-1}(L + U)$ and $\mathbf{c} = D^{-1}\mathbf{b}$, Jacobi iteration method can also be written as $\mathbf{x}^{(k)} = T\mathbf{x}^{(k-1)} + \mathbf{c} \quad k = 1, 2, 3, \dots$

2. Source code

2.1 Collecting user input

jacobi.txt is to be filled with data in the following order:

1st line: number of equations

Next lines: $a[0][0] \ a[0][1] \ \dots \ b[0]$ and so on

The last line: number of iterations

So, we read the matrix from file:

$$Ax = b \text{ with } A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \text{ and } b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \text{ for } x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

Firstly we have to check if the matrix is diagonally dominant.

```
double diagonal;
double others;
int test1=0;
int test2=0;

for (int i = 0; i < n; i++) {
    others = 0;
    for (int j = 0; j < n; j++) {
        if (i == j) diagonal = a[i][j];
        else others += a[i][j];
    }
    if (abs(diagonal) >= abs(others)) test1++;
    if (diagonal > others) test2++;
}

if (test1 == n && test2 > 0) {
    cout << "Matrix is diagonally dominant." << endl;
}
else {
    cout << "Matrix is not diagonally dominant. Use another matrix." << endl;
    return;
}
```

2. Split A into D,L,U

We split A into

$$A = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix} - \begin{bmatrix} 0 & \dots & 0 & 0 \\ -a_{21} & \dots & 0 & 0 \\ \vdots & & \ddots & \vdots \\ -a_{n1} & \dots & -a_{n,n-1} & 0 \end{bmatrix} - \begin{bmatrix} 0 & -a_{12} & \dots & -a_{1n} \\ 0 & 0 & & \vdots \\ \vdots & \vdots & \ddots & -a_{n-1,n} \\ 0 & 0 & \dots & 0 \end{bmatrix} = D - L - U$$

```

d = new double*[n];
d_inv = new double*[n];
l = new double*[n];
u = new double*[n];
m = new double*[n];

for (int i = 0; i < n; ++i) {
d[i] = new double[n];
d_inv[i] = new double[n];
l[i] = new double[n];
u[i] = new double[n];
m[i] = new double[n];
}

for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
d[i][j] = 0.0;
d_inv[i][j] = 0.0;
l[i][j] = 0.0;
u[i][j] = 0.0;
}
}

```

```

//D and D_inv
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
d[i][i] = a[i][i];
d_inv[i][i] = 1 / a[i][i];
}
}

//L
for (int i = 1; i < n; i++) {
for (int j = 0; j < i; j++) {
if (a[i][j] != 0) l[i][j] = a[i][j];
}
}

//U
int temp = 0;
for (int i = 0; i < n ; i++) {
for (int j = n-1; j > temp ; j--) {
if (a[i][j] != 0.0) u[i][j] = a[i][j];
}
temp++;
}

```

Displaying matrices:


```

cout << "The matrix D\n";
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (j > n - 1) cout << "| ";
        cout << d[i][j] << " ";
    }
    cout << "\n";
}
cout << "The matrix L\n";
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (j > n - 1) cout << "| ";
        cout << l[i][j] << " ";
    }
    cout << "\n";
}
cout << "The matrix U\n";
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (j > n - 1) cout << "| ";
        cout << u[i][j] << " ";
    }
    cout << "\n";
}

```

```

//Matrix D^(-1)
cout << "The matrix D^(-1)\n";
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {

```

Function to calculate $x[i]$

```

for (int i = 1; i < ilimit+1; i++) {

    x[i] = sumVECTOR(multiplySQplusVECTOR(multiplySQ(multiplySQ(d_inv, m,n), sumSQ(l, u, n), n), x[i-1], n), multiplySQplusVECTOR(d_inv, b, n), n);
}

```

The matrices operations are handled by MatrixOperations.h

Displaying the results:

```

for (int i = 0; i < ilimit; i++) {
    cout << "Iteration no " << i + 1 << endl << "Solution: ";
    for (int j = 0; j < n; j++) {
        cout << x[i][j] << " ";
    }
    cout << endl << endl;
}

```

Output for the file from website:

```
The matrix from file
5  1  1  1  1 | 10
1  12  1  0  1 | 15
0  1  32  1  0 | 34
1  1  0  4  0 | 6
1  0  1  0  3 | 5
Matrix is diagonally dominant.
```

```
The matrix D
5  0  0  0  0
0  12  0  0  0
0  0  32  0  0
0  0  0  4  0
0  0  0  0  3
```

```
The matrix L
0  0  0  0  0
1  0  0  0  0
0  1  0  0  0
1  1  0  0  0
1  0  1  0  0
```

```
The matrix U
0  1  1  1  1
0  0  1  0  1
0  0  0  1  0
0  0  0  0  0
0  0  0  0  0
```

```
The matrix D^(-1)
0.166667  0  0  0  0
0  0.0833333  0  0  0
0  0  0.03125  0  0
0  0  0  0.25  0
0  0  0  0  0.333333
```

Iteration no 1
Solution: 0 0 0 0 0

Iteration no 2
Solution: 1.66667 1.25 1.0625 1.5 1.66667

Iteration no 3
Solution: 0.753472 0.883681 0.976563 0.770833 0.756944

Iteration no 4
Solution: 1.102 1.04275 1.0108 1.09071 1.08999

Iteration no 5
Solution: 0.960959 0.983102 0.995829 0.963813 0.962402

Iteration no 6
Solution: 1.01581 1.00673 1.00166 1.01398 1.0144

Iteration no 7
Solution: 0.99387 0.997344 0.999353 0.994364 0.994177

Iteration no 8
Solution: 1.00246 1.00105 1.00026 1.0022 1.00226

Iteration no 9
Solution: 0.999039 0.999585 0.999899 0.999122 0.999094

Iteration no 10
Solution: 1.00038 1.00016 1.00004 1.00034 1.00035

Iteration no 11
Solution: 0.99985 0.999935 0.999984 0.999863 0.999859

Iteration no 12
Solution: 1.00006 1.00003 1.00001 1.00005 1.00006

Iteration no 13
Solution: 0.999976 0.99999 0.999998 0.999979 0.999978

Iteration no 14
Solution: 1.00001 1 1 1.00001 1.00001

Iteration no 15
Solution: 0.999996 0.999998 1 0.999997 0.999997

```
Iteration no 16
Solution: 1 1 1 1 1

Iteration no 17
Solution: 0.999999 1 1 0.999999 0.999999

Iteration no 18
Solution: 1 1 1 1 1

Iteration no 19
Solution: 1 1 1 1 1

Iteration no 20
Solution: 1 1 1 1 1

Iteration no 21
Solution: 1 1 1 1 1

Iteration no 22
Solution: 1 1 1 1 1

Iteration no 23
Solution: 1 1 1 1 1

Iteration no 24
Solution: 1 1 1 1 1

Iteration no 25
Solution: 1 1 1 1 1

Iteration no 26
Solution: 1 1 1 1 1

Iteration no 27
Solution: 1 1 1 1 1

Iteration no 28
```

..and so on until it hits the last iteration.

5/ Bisection method

1. Bisection method

Bisection method is a root-finding method that applies to any continuous functions for which we know two values with opposite signs.

The method consists of repeatedly bisecting the interval defined by these values and then selecting the subinterval in which the function changes sign, and therefore must contain a root.

// Wykonanie

Pobieranie danych wraz z precyzją od użytkownika oraz komunikat gdy jest źle dobrany interwał tak że nie możliwe jest znalezienie miejsca zerowego

```
void f1()
{
    double a, b;
    double precision;

    cout << "f1(x) = cos(x^3 - 3x)\n";

    cout << "Insert interval start:";
    cin >> a;
    cout << "Insert interval end: ";
    cin >> b;
    cout << "Insert precision: (for example 0.01): ";
    cin >> precision;

    double y1 = cos(a * a * a - 3 * a);
    double y2 = cos(b * b * b - 3 * b);

    if (y1*y2 >= 0) {
        cout << " Wrong interval" << endl;
        return;
    }
}
```



```

double x = (a + b) / 2;
double y = cos(x * x * x - 3 * x);

cout << "f1(" << a << ") = " << y1 << endl;
cout << "f1(" << b << ") = " << y2 << endl;
cout << "x = " << x << endl;
cout << "f1(" << x << ") = " << y << endl;

int i = 1;
while ((b - a) >= precision)
{
    if (y > 0)
        a = (a + b) / 2;

    else
        b = (a + b) / 2;

    cout << "\nIteration:" << i++ << ": interval: <" << a << ", " << b << ">, interval length: " << b - a << endl << endl;

    x = (a + b) / 2;
    y = cos(x * x * x - 3 * x);
    y1 = cos(a * a * a - 3 * a);
    y2 = cos(b * b * b - 3 * b);

    cout << "f1(" << a << ") = " << y1 << endl;
    cout << "f1(" << b << ") = " << y2 << endl;
    cout << "x = " << x << endl;
    cout << "f1(" << x << ") = " << y << endl;
}

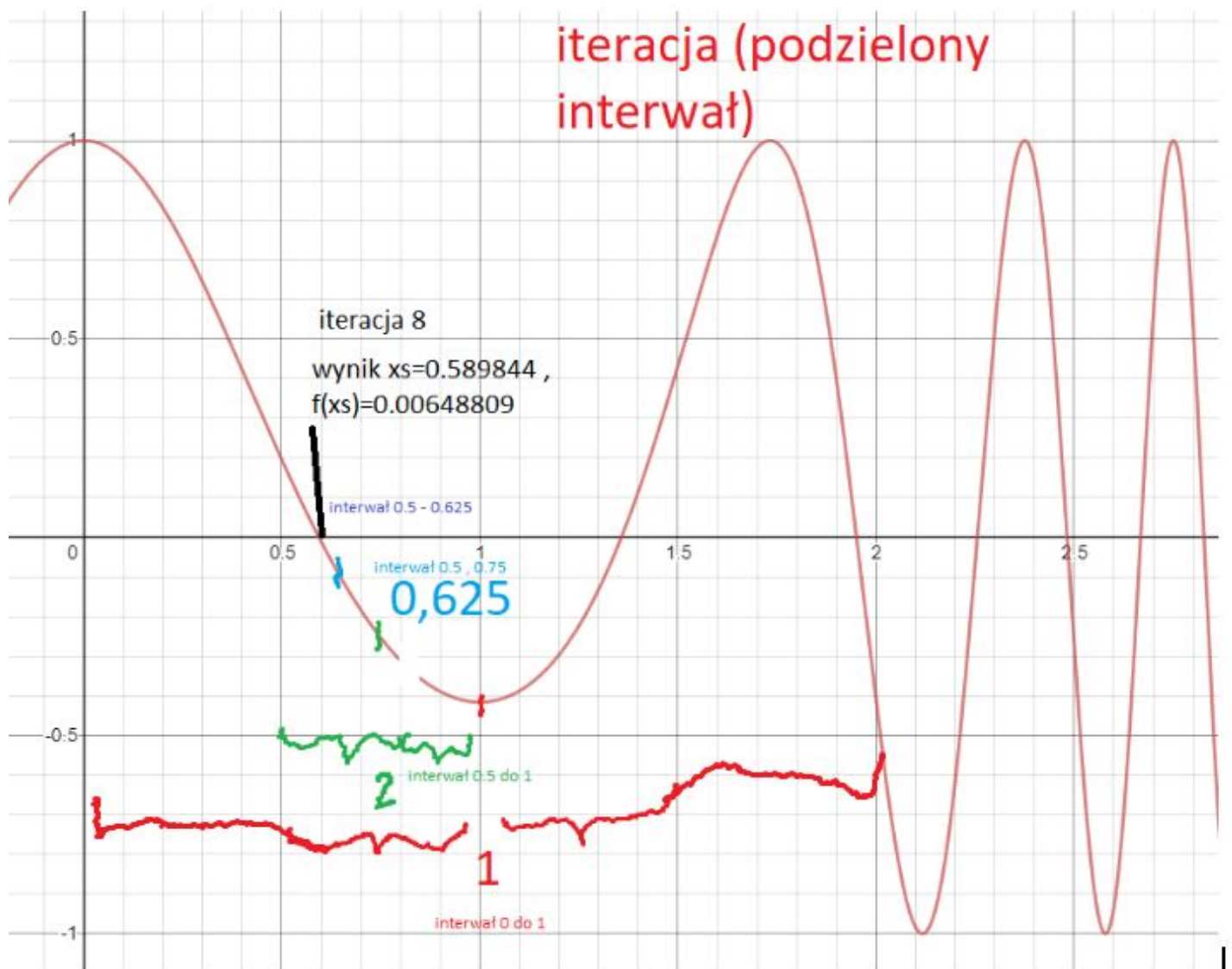
cout << "\n\nValue of f1 with declared precision "<<precision<<": f1(" << x << ") = " << y << endl;
}

```

////

Wykonanie dla funkcji $\cos(x^3 - 3x)$

iteracja (podzielony interwał)



```
f1(x) = cos(x^3 - 3x)
Insert interval start:0
Insert interval end: 2
Insert precision: (for example 0.01): 0.01
f1(0) = 1
f1(2) = -0.416147
x = 1
f1(1) = -0.416147
```

Iteration:1: interval: $\langle 0,1 \rangle$, interval length: 1

```
f1(0) = 1
f1(1) = -0.416147
x = 0.5
f1(0.5) = 0.194548
```

Iteration:2: interval: $\langle 0.5,1 \rangle$, interval length: 0.5

```
f1(0.5) = 0.194548
f1(1) = -0.416147
x = 0.75
f1(0.75) = -0.254498
```

Iteration:3: interval: $\langle 0.5,0.75 \rangle$, interval length: 0.25

```
f1(0.5) = 0.194548
f1(0.75) = -0.254498
x = 0.625
f1(0.625) = -0.0600269
```

Iteration:4: interval: $\langle 0.5,0.625 \rangle$, interval length: 0.125

```
f1(0.5) = 0.194548
f1(0.625) = -0.0600269
x = 0.5625
f1(0.5625) = 0.0612365
```

Iteration:5: interval: $\langle 0.5625,0.625 \rangle$, interval length: 0.0625

```
f1(0.5625) = 0.0612365
f1(0.625) = -0.0600269
x = 0.59375
f1(0.59375) = -0.0011336
```

Iteration:6: interval: $\langle 0.5625,0.59375 \rangle$, interval length: 0.03125

```
f1(0.5625) = 0.0612365
f1(0.59375) = -0.0011336
x = 0.578125
f1(0.578125) = 0.0296428
```

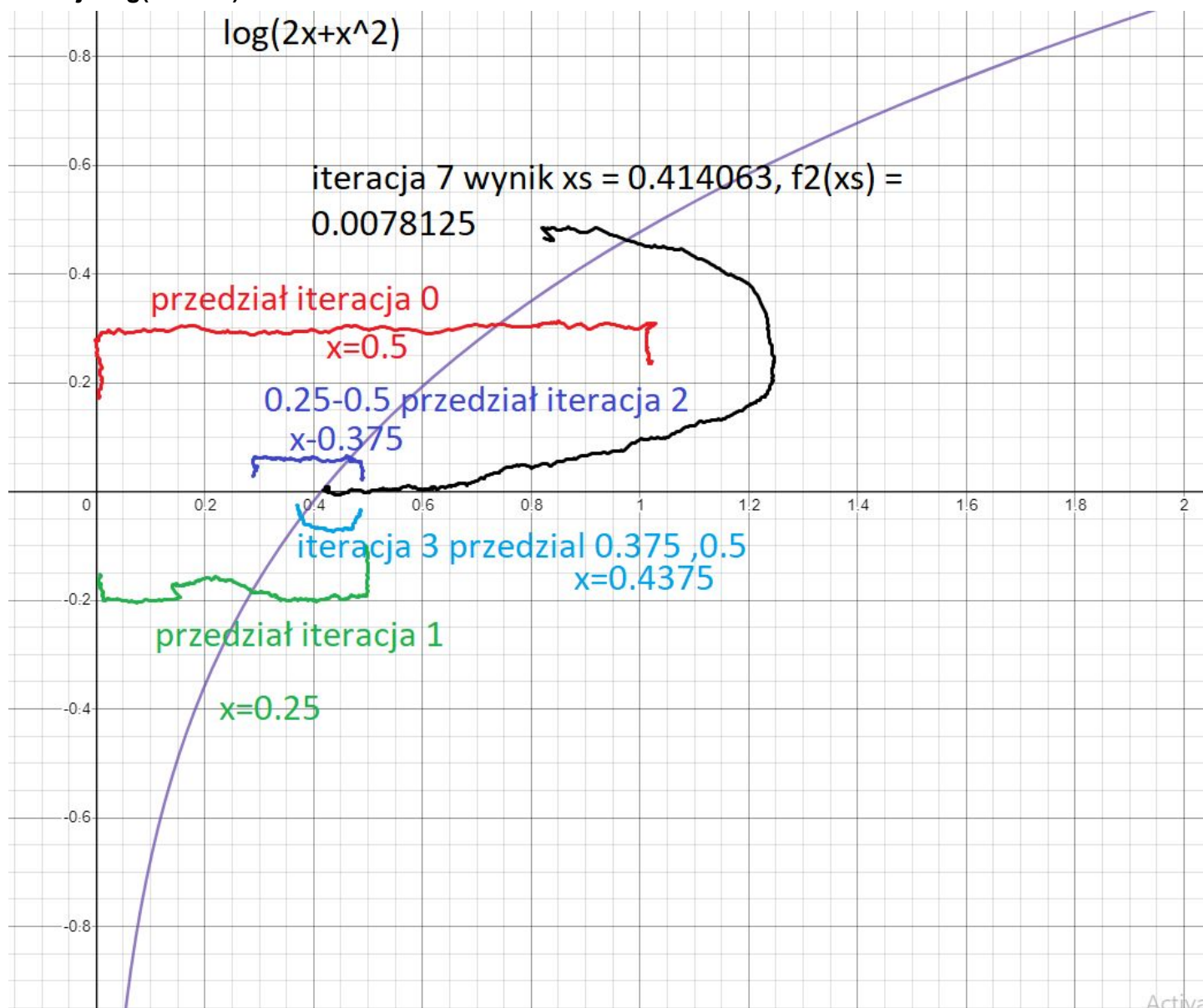
Iteration:7: interval: $\langle 0.578125,0.59375 \rangle$, interval length: 0.015625

```
f1(0.578125) = 0.0296428
f1(0.59375) = -0.0011336
x = 0.585938
f1(0.585938) = 0.014149
```

Iteration:8: interval: $\langle 0.585938,0.59375 \rangle$, interval length: 0.0078125

```
f1(0.585938) = 0.014149
f1(0.59375) = -0.0011336
x = 0.589844
f1(0.589844) = 0.0064809
```

Funkcja $\log(x^2+2x)$



```

f1(x) = log(x^2 + 2x)
Insert interval start:0
Insert interval end: 1
Insert precision: (for example 0.01): 0.01
f1(0) = -inf
f1(1) = 1.09861
x = 0.5
f1(0.5) = 0.223144

Iteration:1: interval: <0,0.5>, interval length: 0.5

f1(0) = -inf
f1(0.5) = 0.223144
x = 0.25
f1(0.25) = -0.575364

Iteration:2: interval: <0.25,0.5>, interval length: 0.25

f1(0.25) = -0.575364
f1(0.5) = 0.223144
x = 0.375
f1(0.375) = -0.115832

Iteration:3: interval: <0.375,0.5>, interval length: 0.125

f1(0.375) = -0.115832
f1(0.5) = 0.223144
x = 0.4375
f1(0.4375) = 0.0642944

Iteration:4: interval: <0.375,0.4375>, interval length: 0.0625

f1(0.375) = -0.115832
f1(0.4375) = 0.0642944
x = 0.40625
f1(0.40625) = -0.022717

Iteration:5: interval: <0.40625,0.4375>, interval length: 0.03125

f1(0.40625) = -0.022717
f1(0.4375) = 0.0642944
x = 0.421875
f1(0.421875) = 0.0214958

Iteration:6: interval: <0.40625,0.421875>, interval length: 0.015625

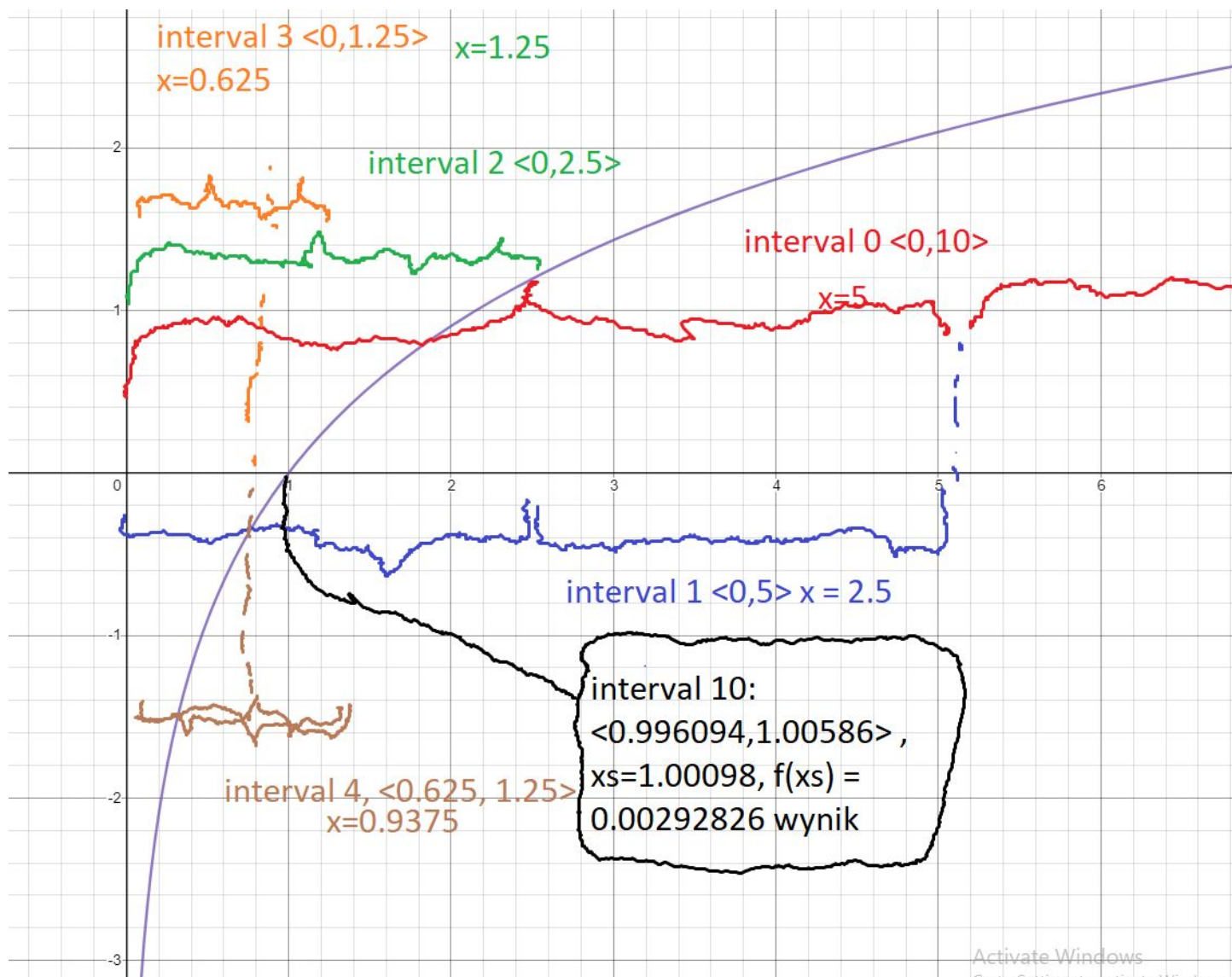
f1(0.40625) = -0.022717
f1(0.421875) = 0.0214958
x = 0.414063
f1(0.414063) = -0.000427337

Iteration:7: interval: <0.414063,0.421875>, interval length: 0.0078125

f1(0.414063) = -0.000427337
f1(0.421875) = 0.0214958
x = 0.417969
f1(0.417969) = 0.0105792

```

Wykonanie dla funkcji $\log(x^3)$



C:\Users\grzegorz\Source\Repos\NumericalMethods2\NumericalMethods3\Debug\NumericalMethods3.exe

$f_3(x) = \log(x^2 + 2x)$

Insert interval start: 0

Insert interval end: 10

Insert precision: (for example 0.01): 0.01

$f_1(0) = -\text{inf}$

$f_1(10) = 6.90776$

$x = 5$

$f_1(5) = 4.82831$

Iteration:1: interval: $\langle 0, 5 \rangle$, interval length: 5

$f_1(0) = -\text{inf}$

$f_1(5) = 4.82831$

$x = 2.5$

$f_1(2.5) = 2.74887$

Iteration:2: interval: $\langle 0, 2.5 \rangle$, interval length: 2.5

$f_1(0) = -\text{inf}$

$f_1(2.5) = 2.74887$

$x = 1.25$

$f_1(1.25) = 0.669431$

Iteration:3: interval: $\langle 0, 1.25 \rangle$, interval length: 1.25

$f_1(0) = -\text{inf}$

$f_1(1.25) = 0.669431$

$x = 0.625$

$f_1(0.625) = -1.41001$

Iteration:4: interval: $\langle 0.625, 1.25 \rangle$, interval length: 0.625

$f_1(0.625) = -1.41001$

$f_1(1.25) = 0.669431$

$x = 0.9375$

$f_1(0.9375) = -0.193616$

Iteration:5: interval: $\langle 0.9375, 1.25 \rangle$, interval length: 0.3125

$f_1(0.9375) = -0.193616$

$f_1(1.25) = 0.669431$

$x = 1.09375$

$f_1(1.09375) = 0.268836$

```

Iteration:6: interval: <0.9375,1.09375>, interval length: 0.15625

f1(0.9375) = -0.193616
f1(1.09375) = 0.268836
x = 1.01563
f1(1.01563) = 0.0465126

Iteration:7: interval: <0.9375,1.01563>, interval length: 0.078125

f1(0.9375) = -0.193616
f1(1.01563) = 0.0465126
x = 0.976563
f1(0.976563) = -0.0711496

Iteration:8: interval: <0.976563,1.01563>, interval length: 0.0390625

f1(0.976563) = -0.0711496
f1(1.01563) = 0.0465126
x = 0.996094
f1(0.996094) = -0.0117417

Iteration:9: interval: <0.996094,1.01563>, interval length: 0.0195313

f1(0.996094) = -0.0117417
f1(1.01563) = 0.0465126
x = 1.00586
f1(1.00586) = 0.0175268

Iteration:10: interval: <0.996094,1.00586>, interval length: 0.00976563

f1(0.996094) = -0.0117417
f1(1.00586) = 0.0175268
x = 1.00098
f1(1.00098) = 0.00292826

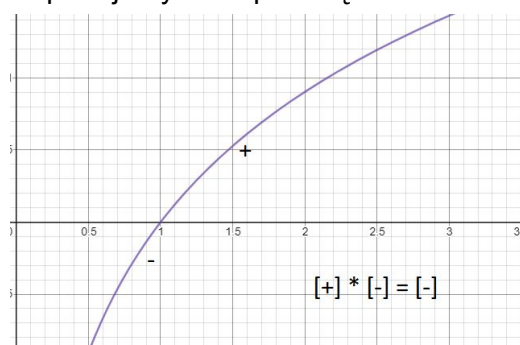
Value of f1 with declared precision 0.01: f1(1.00098) = 0.00292826
Press any key to continue . . .

```

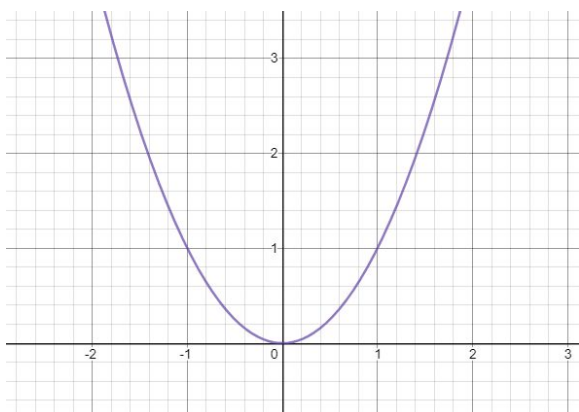
1. Czy podany powyżej warunek istnienia rozwiązania jest warunkiem koniecznym i wystarczającym?

$$f(a) \cdot f(b) < 0 .$$

Warunek ten dla znalezienia konkretnego dowolnego rozwiązania jest wystarczający tylko w przypadku funkcji która przechodzi przez część ujemną i dodatnią wykresu kartezjańskiego, ponieważ gdy wartość funkcji dla jednego argumentu jest dodatnia a drugiej jest ujemna to siłą rzeczy musi funkcja ciągła, czyli taką jaką rozpatrujemy musi przeciąć zero i mieć miejsce zerowe.

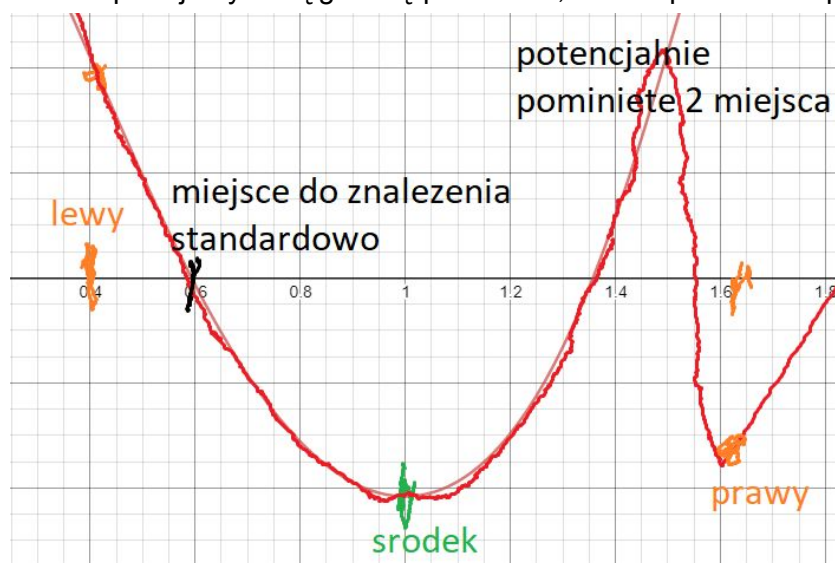


Jeśli mamy do czynienia z funkcją, która jedynie “haczy” o wartość 0, to metoda bisekcji nie znajdzie takiego rozwiązania, więc warunek ten nie jest konieczny i wystarczający aby znaleźć miejsce zerowe.



2. Podany algorytm znajduje zawsze tylko jedno z rozwiązań. Jak można zmodyfikować algorytm, by znalazł więcej rozwiązań, o ile oczywiście one istnieją.

Jeśli rozpatrujemy lewą granicę przedziału, środek przedziału i prawą granicę przedziału, to



W sytuacji jak powyżej pominiemy dwa miejsca po prawej stronie ponieważ prawy koniec przedziału był ujemny więc nie wiedzieliśmy co się dzieje pomiędzy środkiem a prawym końcem przedziału.

Można by było np. Zmodyfikować program tak by wziął wyrykowo kilka punktów z prawego przedziału i sprawdził czy są mniejsze niż zero, jeśli by znalazł jakiś punkt większy niż zero wiedziałby, że istnieją conajmniej dwa nieznane miejsca zerowe i wtedy rozpatrywać przedział między środkiem, a znalezionym punktem o przeciwnym znaku, i między tym punktem a prawym końcem przedziału.

6/ Metoda Newtona(stycznych) i siecznych. /pl/

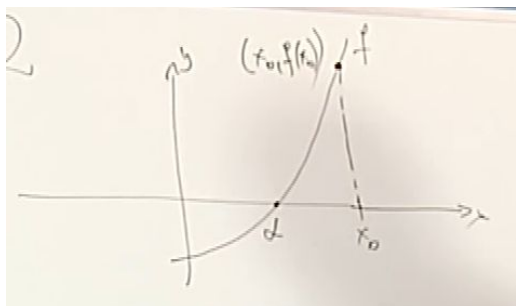
1. Metoda Newtona (stycznych)

Matematyka:

Szukamy rozwiązania równania $f(x) = 0$. Oznaczamy miejsce zerowe funkcji f jako α .

Założmy że funkcja ma pochodną $f'(x)$, $f''(x)$, z tym że $f'(x)$ jest różna od zera w otoczeniu punktu α .

Wybieramy w otoczeniu α jeden punkt x_0 .



Mamy punkt $(x_0, f(x_0))$. Rysujemy styczną w tym punkcie do funkcji f .

Równanie stycznej: $y - f(x_0) = f'(x_0)(x - x_0)$

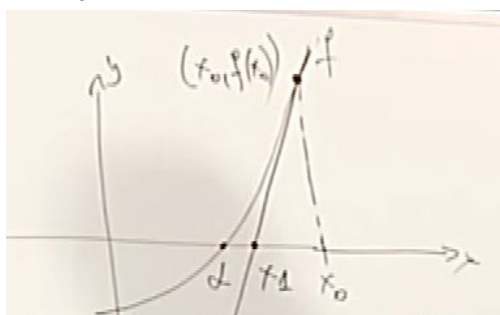
Punkt w którym styczna ta przecina oś OX nazywamy x_1 i liczymy. Wyznaczamy więc miejsce zerowe tej stycznej. Do powyższego równania więc podstawiamy $x=x_1$, $y=0$ i wychodzi wzorek na x_1 .

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

Twierdzenie Newtona:

Jeżeli funkcja f i f' na odcinku $[\alpha, x_0]$ są tego samego znaku to $\alpha < x_1 < x_0$

W naszym przypadku gdy f jest dodatnią funkcją jest "wypukła::rosnąca" czyli druga pochodna jest dodatnia po prawej stronie punktu α .



I faktycznie x_1 przybliży nas do α .

2 Iteracja:

Teraz należy wyznaczyć kolejną styczną, w $(x_1, f(x_1))$, równanie będzie miało $y - f(x_1) = f'(x_1)(x - x_1)$

Wyznaczyć x_2 czyli pkt przecięcia z osią x , czyli znowu podstawić $x=x_2$, $y=0$ do równania stycznej powyżej.

$$x_2 = x_1 - f(x_1) / f'(x_1)$$

Z twierdzenia Newtona..... $\alpha < x_2 < x_1 < x_0$

Kolejne iteracje:

$$x[n+1] = x[n] - f(x[n]) / f'(x[n])$$

Z twierdzenia Newtona wynika, że jest to ciąg malejący o ogólnej zasadzie: $\alpha < \dots x[n] < x_2 < x_1 < x_0$

$x[n]$ jest ograniczony z dołu przez alfa, więc jest to ciąg zbieżny $x[n] \rightarrow (n \rightarrow \infty) \rightarrow g$

Nakładając funkcję f : $f(x[n]) \rightarrow (n \rightarrow \infty) \rightarrow f(g)$

Funkcja ma pochodną, więc jest ciągła, więc dąży do $f(g)$

Zapis na odwrot: $f(g) = \lim_{n \rightarrow \infty} f(x[n]) =$

Przekształcenie 1 wzoru $= \lim_{n \rightarrow \infty} (x[n] - x[n+1]) * f'(x[n])$
 $\rightarrow g \quad \rightarrow g \quad \rightarrow f'(g)$
 $\rightarrow 0 \quad \rightarrow f'(g)$

Więc $f(g) = 0$

Wniosek: $f(g) = 0$, dowód, że g jest m. zerowym funkcji f

Czyli $g = \alpha$

(rozwiązanie α jest liczbą g , a liczba g jest granicą ciągu $x[n]$ - ciągu miejsc zerowych stycznych)(tak więc granicą tego ciągu jest rozwiązanie... możemy liczyć tak długo jak dokładny wynik chcemy mieć)

Algorytm:

Założenia:

1. W przedziale $[a,b]$ funkcja posiada pierwiastek, $f(a) * f(b) < 0$
2. Funkcja $f(x)$ jest ciągła na przedziale $[a,b]$ i posiada pochodną
3. Pierwsza i druga pochodna f ma stały znak na przedziale $[a,b]$.

Kolejne kroki:

1. Wybrać punkt startowy (zazwyczaj a, b 0 lub 1).
2. Wyprowadzić styczną w tym punkcie do funkcji $f(x)$. Współrzędna x gdzie styczna przecina oś OX stanowi pierwsze przybliżenie pierwiastka funkcji.
3. Jeżeli przybliżenie nie jest zadowalające powtarzać krok 2.

Metoda siecznych do znajdowania przybliżeń pochodnej:

$$f'(x) \sim (f(x[i+1]) - f(x[i])) / (x[i+1] - x[i])$$

To można podstawić do wzoru na kolejne przybliżenia.

$$x[i+1] = x[i] - f(x[i]) / f'(x)$$

W tym przypadku jednak w tej sytuacji potrzebujemy przybliżenia nie tyle z poprzedniego kroku, co z dwóch poprzednich. Powinny być one bliskie sobie np. 0 i 0.01.

Zadanie do wykonania:

- Funkcja $f(x)$ zaimplementowana jako osobna funkcja, tak aby łatwo ją było zmienić
- Pochodna f' liczona ręcznie i zaimplementowana jako osobna funkcja

Implementacja w cpp, ponieważ tam mam wszystkie inne programy:

Równanie oraz pochodna jako osobne funkcje:

```

double f(double x)
{
    return cos(x);
}

double fd(double x)
{
    return -sin(x);
}

```

Funkcja do znajdowania pierwiastka

```

void findRoot(double x0, double precision)
{
    cout << "f = cos(x)" << endl;

    double x = x0;
    cout << "Starting point x= " << x << ", precision= " << precision << endl;
    double y = f(x);
    int i = 0;

    while (abs(y) > precision) {
        x = x - y / fd(x);
        y = f(x);
        cout << ++i << ". x= " << x << endl;
    }
    cout << fixed << "Func value:" << y << endl;
}

```

$\cos(x^2+2x)$, punkt startowy 1.8, dokładność 0.001

Wykonanie programu:

```

f = cos(x)
Starting point x= 1.8, precision= 0.001

y= 0.848943
1. x= 2.08685, y= -0.624643
2. x= 1.95729, y= 0.108217
3. x= 1.97569, y= -0.000765674
Func value:-0.000766
Press any key to continue . . .

```

Do narysowania wykresu użyłem osobnego skryptu w MATLAB:

```

func = @(x) cos(x^2+2*x)
fd = @(x) -2*(x+1)*sin(x*(x+2));

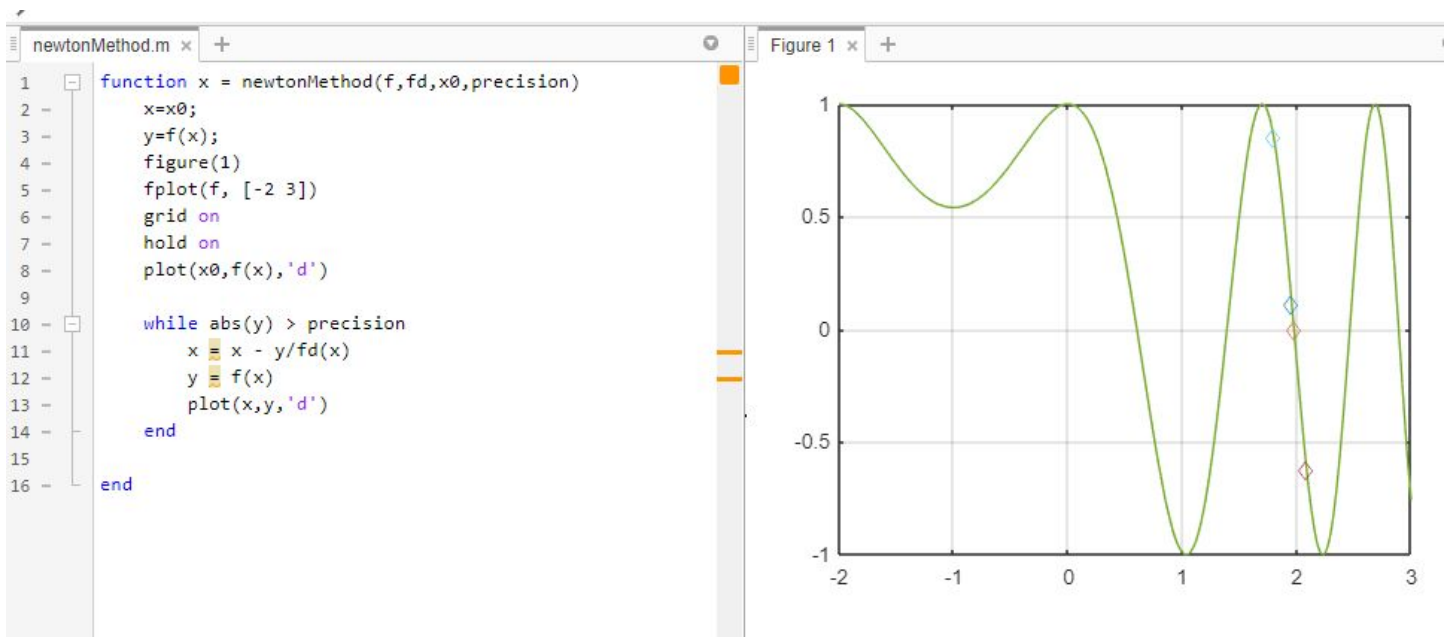
function x = newtonMethod(f,fd,x0,precision)
    x=x0;
    y=f(x);
    figure(1)
    fplot(f, [-2 3])
    grid on
    hold on

    plot(x0,f(x),'d')

    while abs(y) > precision
        x = x - y/fd(x)
        y = f(x)
        plot(x,y,'d')
    end

end

```



Wynik dokładny:

```

COMMAND WINDOW

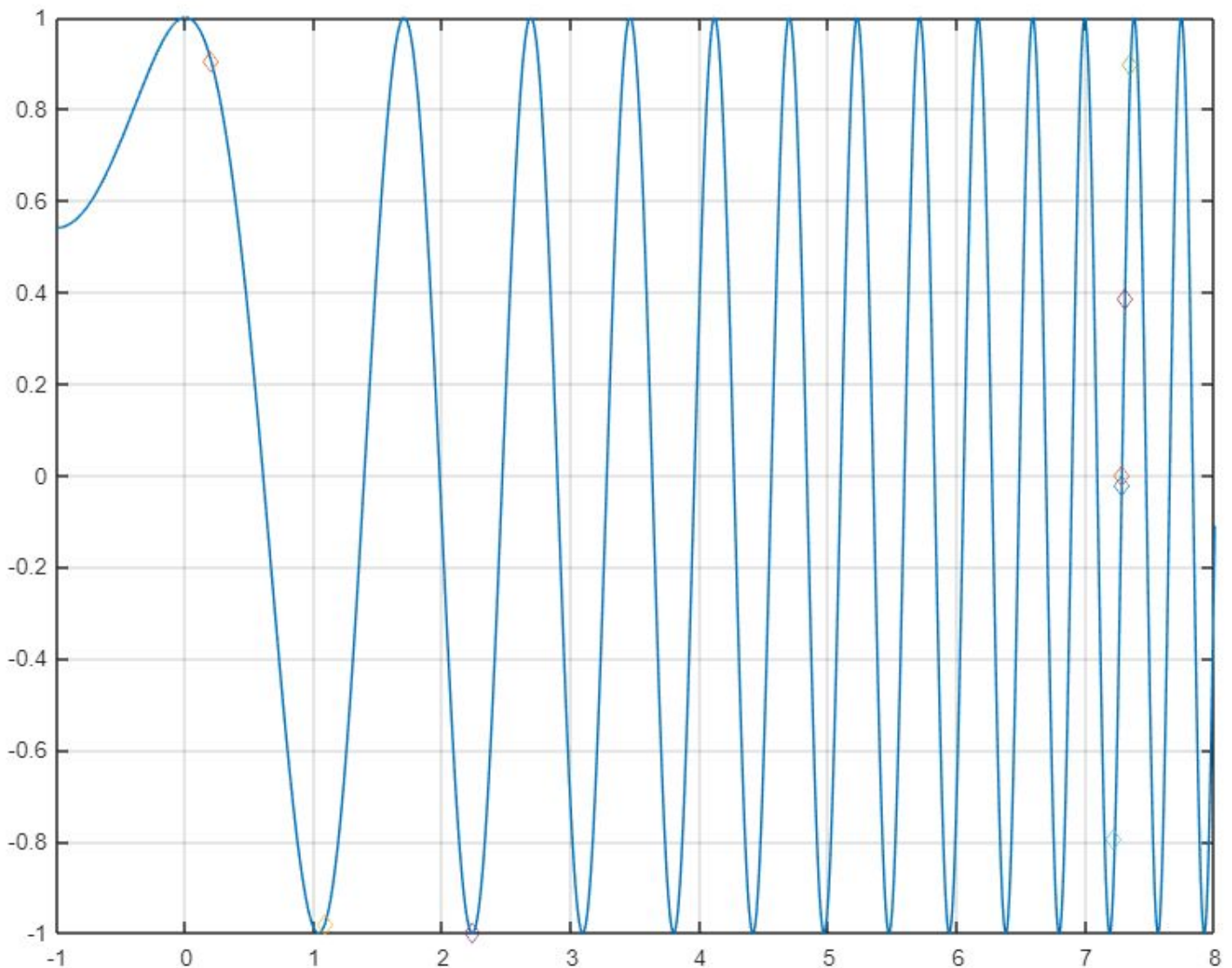
New to MATLAB? See resources for Getting Started.

ans =

    1.9757

```

Ważny jest dobór punktu startowego. Dla powyższej funkcji $\cos(x^2+2x)$ po wybraniu punktu $x=0.2$ styczne układają się tak, że przeskakujemy z $x \sim 1$ na $x \sim 2$, by znaleźć się na $x \sim 7$ z ostatecznym wynikiem 7 iteracji $x=7.2791$.



Punkty to po kolei :

$x=1.08$

$x=2.23$

$x=7.34$

$x=7.22$

$x=7.30$

$x=7.27$

$x=7.27$

Ostatecznie znaleziony pierwiastek:

$y = 4.9733e-06$, $x = 7.2791$

Funkcja z metodą siecznych oraz wykres:

```

function x = newton_secantsMethod(f,x0,x1,precision)
    x=x1;
    y=f(x1);
    figure
    fplot(f, [-1 3])
    grid on
    hold on
    plot(x0,f(x0),'d')
    plot(x1,f(x1),'d')

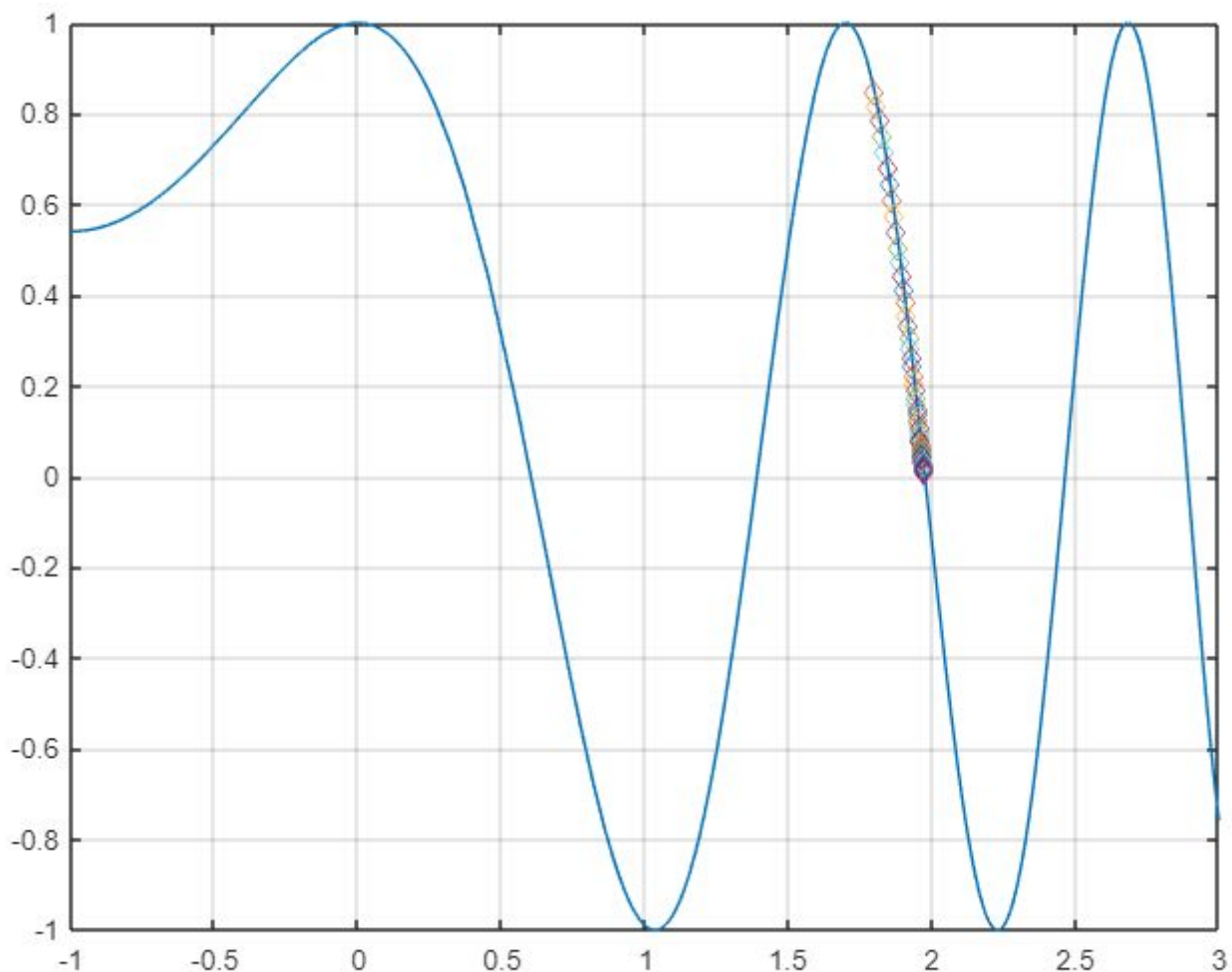
    i = 0;

    while abs(y) > precision
        temp = x;
        i = i+1
        x = x - y/ (f(x)-f(x0)/(x-x0))
        x0 = temp;
        y = f(x)
        plot(x,y,'d')
    end

end

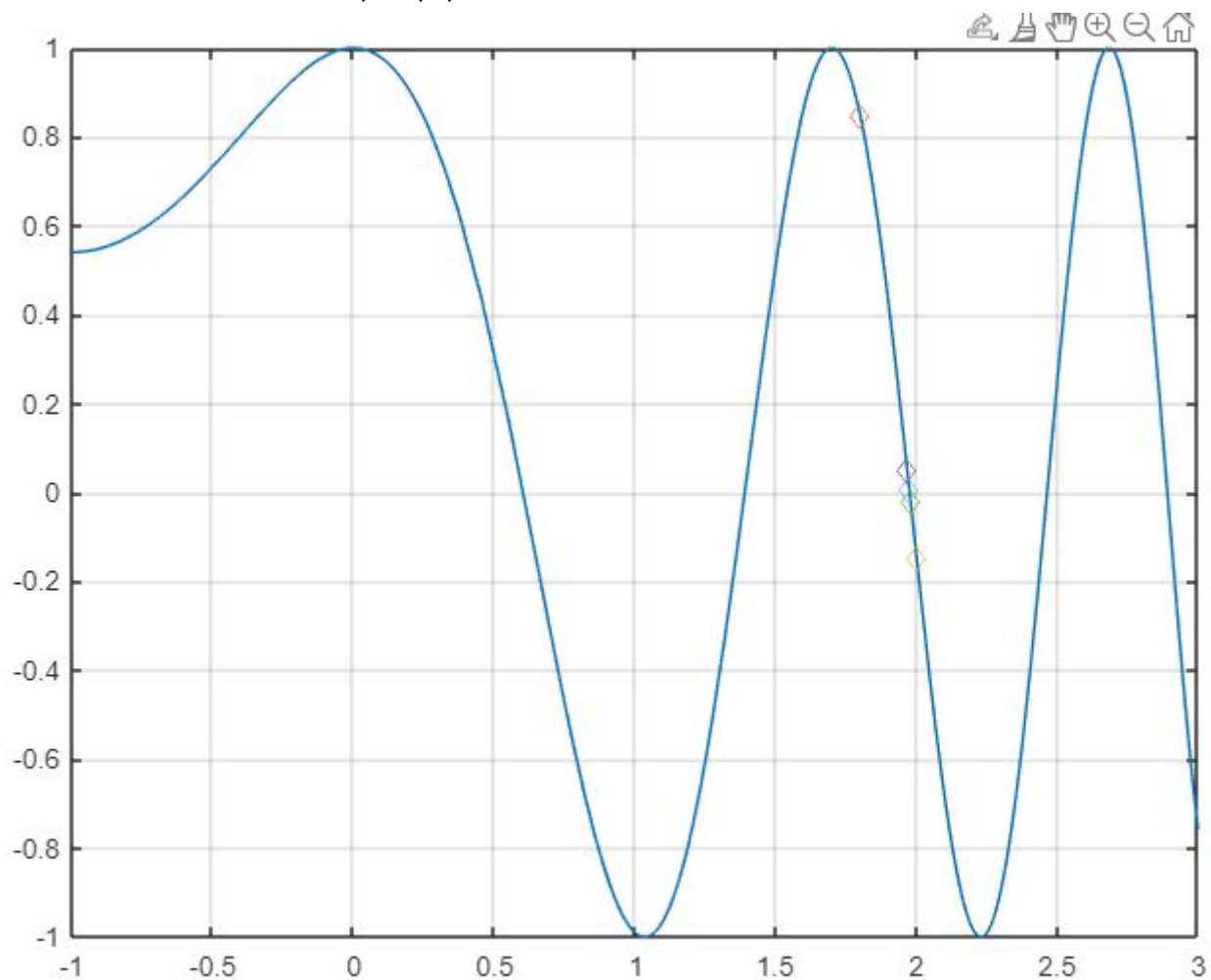
```

Wykres dla wywołania: `newton_secantsMethod(func,1.8,1.81,0.01)` znajduje rozwiązanie w 57 iteracji. Robi stosunkowo małe kroki, ponieważ zadana jest mała różnica między punktami startowymi



Gdy zadana różnica między punktami startowymi jest większa tj. np. wywołanie dla $x_0 = 1.8$ oraz $x_1=2.0$

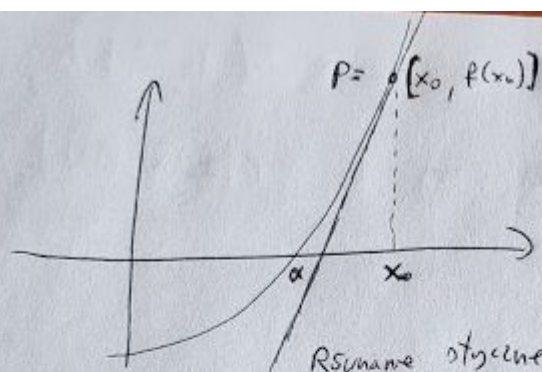
`newton_secantsMethod(func,1.8,2,0.01)`



Wywołanie to znajduje rozwiązanie w 3 iteracji:

$x=1.9669$	$y=0.0517$
$x=1.9788$	$y=-0.0191$
$x=1.9744$	$y=0.0070$

Obliczenia; identyczne jak w 1 podpunkcie sprawozdania, na kartce: (warunek 25pkt)



Równanie stycznej do funkcji f
w punkcie x_0 :

$$y - f(x_0) = f'(x_0)(x - x_0)$$

Styczna przecina oś w punkcie x_1

Podstawiam $x_1 = x$, $y = 0$ do wzoru

$$x_1 = x_0 - f(x_0) / f'(x_0) \quad \rightarrow \text{analogicznie}$$

$$x_2 = x_1 - f(x_1) / f'(x_1)$$

$$\boxed{\alpha < x_1 < x_0}$$

Stąd:

$$x[n+1] = x[n] - f(x[n]) / f'(x[n])$$

A7/ Newton-Raphson

1. Metoda Newtona-Raphsona

Metoda rozwiązywania układu równań nieliniowych Newtona-Raphsona jest odpowiednikiem metody stycznych. Różnica polega na tym, że tym razem działamy na zmiennych macierzowych, a zamiast pochodnej liczymy jacobian macierzy.

Wersja podstawowa zadania:

```
//newton-raphson method
double eq1(double x1, double x2) {
    return x1 * x1*x1 + 2*x2*x2;
}

double eq2(double x1, double x2) {
    return 4 * x1 + sin(x2);
}

//jacobian j[i][j]
double j11(double x1, double x2) {
    return 3 * x1 *x1;
}

double j12(double x1, double x2) {
    return 4 * x2;
}

double j21(double x1, double x2) {
    return 4;
}

double j22(double x1, double x2) {
    return cos(x2);
}
```

```
//(it -number of iterations
void findRootNR(int it)
{
```

```

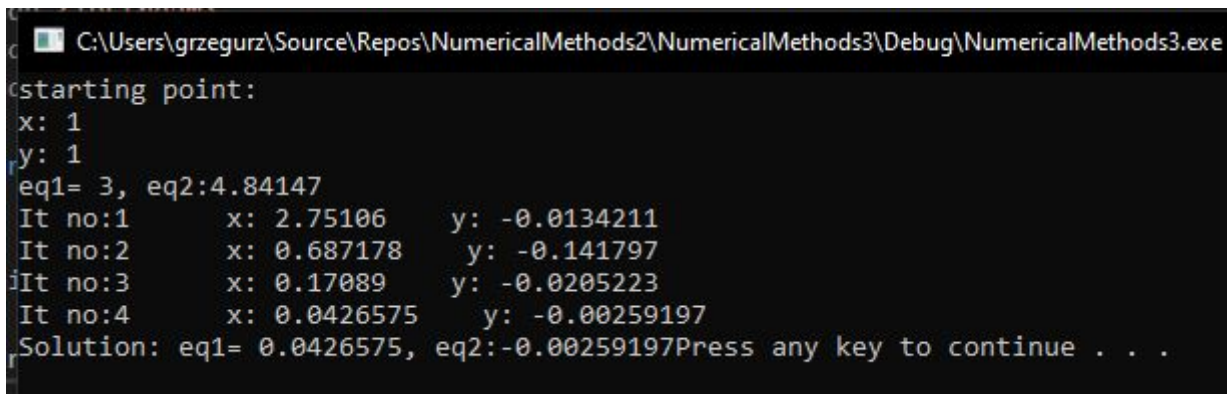
double *X = new double[2];
cout << "starting point:" << endl;
cout << "x: ";
cin >> *X;
cout << "y: ";
cin >> *(X + 1);

cout << "eq1= " << eq1(*X, *(X + 1)) << ", eq2:" << eq2(*X, *(X + 1))<<endl;
double x1, x2;
for (int i = 0; i < it; i++) {
    x1 = *X;
    x2 = *(X + 1);
    double odw= 1 / (j11(x1,x2)*j22(x1,x2) - j12(x1,x2)*j21(x1,x2)); //odwrotnosc macierzy
    *X = x1 - odw* (eq1(x1, x2)*j22(x1,x2) - eq2(x1, x2)*j12(x1,x2));
    *(X + 1) = x2 - odw* (eq2(x1, x2)*j11(x1,x2) - eq1(x1, x2)*j21(x1,x2));

    cout << "It no:" << i + 1 << "    x: " << eq1(*X, *(X + 1)) << "    y: " << eq2(*X, *(X + 1)) << endl;
}

cout <<"Solution: eq1= " << eq1(*X, *(X + 1)) << ", eq2:" << eq2(*X, *(X + 1));
}

```



```

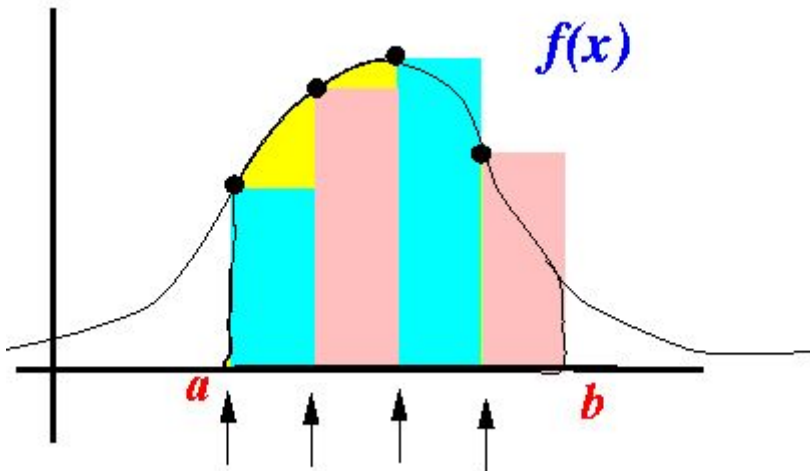
C:\Users\grzegorz\Source\Repos\NumericalMethods2\NumericalMethods3\Debug\NumericalMethods3.exe
starting point:
x: 1
y: 1
eq1= 3, eq2:4.84147
It no:1      x: 2.75106      y: -0.0134211
It no:2      x: 0.687178     y: -0.141797
It no:3      x: 0.17089      y: -0.0205223
It no:4      x: 0.0426575    y: -0.00259197
Solution: eq1= 0.0426575, eq2:-0.00259197Press any key to continue . . .

```

A8/ Rectangular and Trapezoidal Methods of approximation of definite integral.

1. The rectangle method

The rectangle method (also called the midpoint rule) is the simplest method in Mathematics used to compute an approximation of a definite integral.



$$\text{width} = \frac{b - a}{n}$$

$$\text{height} = f(\text{start_of_the_subinterval})$$

Algorithm:

```
double rectangle_area = 0;
double width = (b - a) / n;

for (int i = 0; i < n; i++) {
    rectangle_area += width * func1(a + i * width);
}
```

Output:

```
Func 1: x^3 + x^2+ x
Integration between 2 and 4, subintervals no: 7
Rectangle area=74.9388
Press any key to continue . . .
```

```
Func 2: x^4 + x^3 + x^2+ x
Integration between 2 and 4, subintervals no: 7
Rectangle area=240.576
Press any key to continue . . .
```



```

Func 3: cos(x)
Integration between 6 and 7, subintervals no: 9
Rectangle area=0.946898

Press any key to continue . . .

```

2. Trapezoidal rule

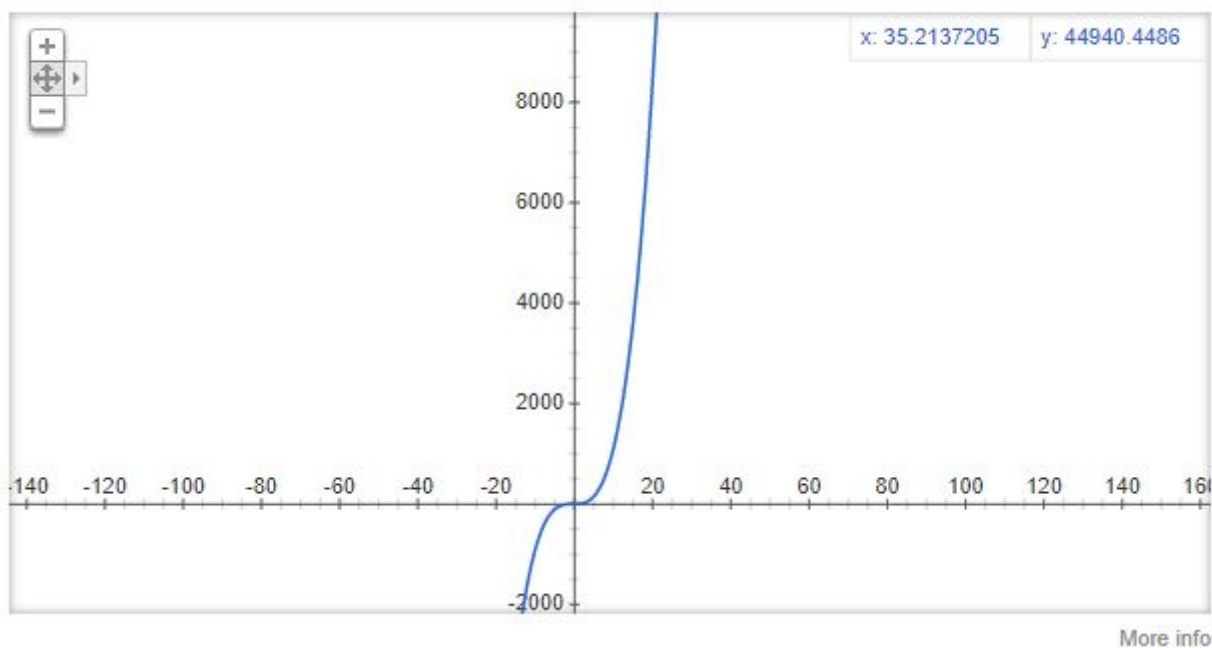
```

trapezoid_area += width * ( func3(a + i * width) + func3(a + i * width+width) ) / 2;

```

Big difference in approximation precision when we calculate function as below:

Graph for $x^3 + x^2 + x$



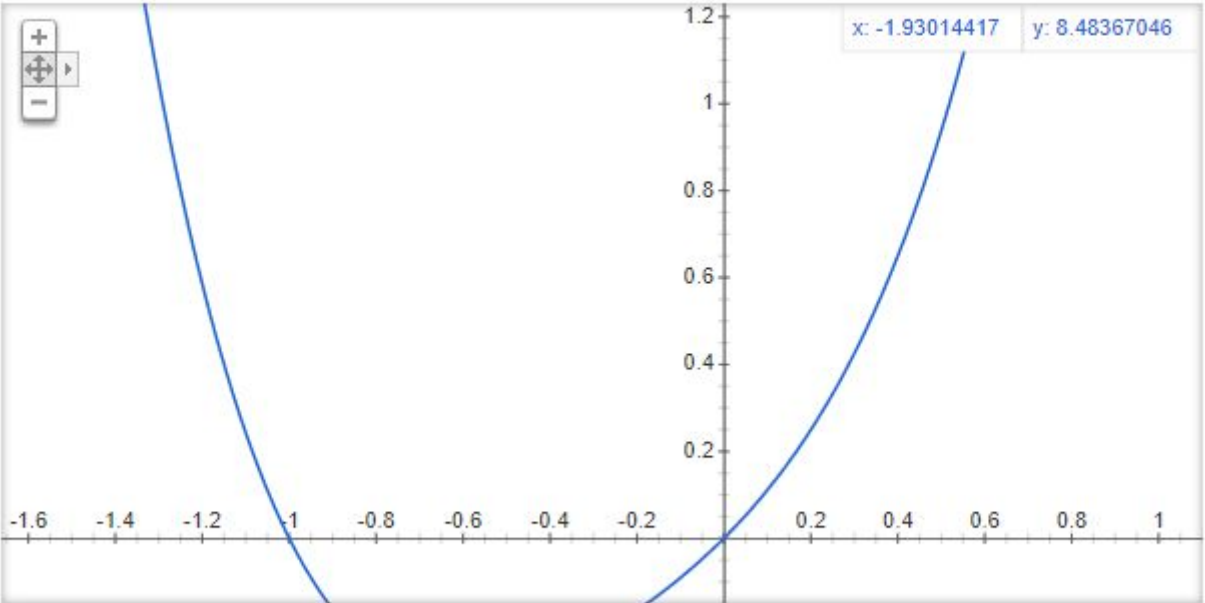
```

Func 1: x^3 + x^2 + x
Integration between 10 and 20, subintervals no: 5
Rectangular rule area=32980
Trapezoidal rule area=40290
Press any key to continue . . .

```

Rectangular function do its job with much less precision when we have to face a very rapidly growing function.

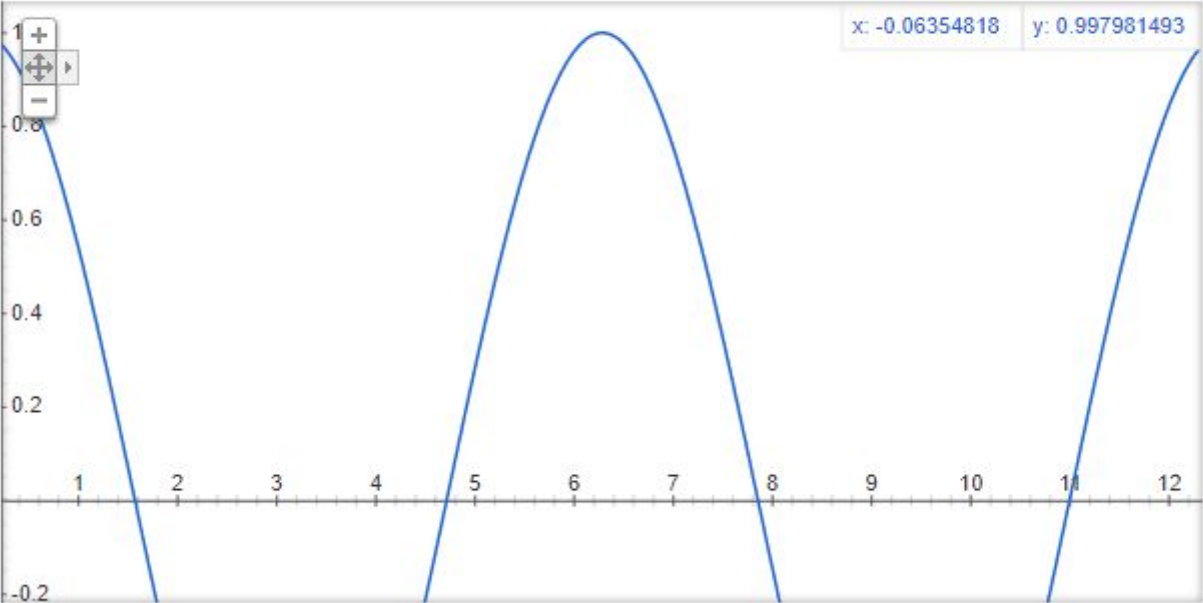
Graph for $x^4+x^3+x^2+x$



More info

```
Func 2: x^4 + x^3 + x^2+ x
Integration between 0.2 and 0.4, subintervals no: 7
Rectangular rule area=0.0810033
Trapezoidal rule area=0.0867176
Press any key to continue . . .
```

Graph for $\cos(x)$

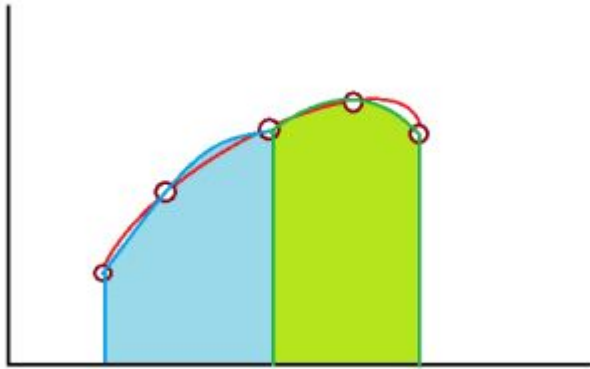


More info

```
Func 3: cos(x)
Integration between 5 and 7, subintervals no: 10
Rectangular rule area=1.5635
Trapezoidal rule area=1.61052
Press any key to continue . . .
```

A9/ Simpson, Monte Carlo, Gauss Methods of approximation of definite integral.

1. Simpson Method - using quadratic functions.



$$\int_a^b f(x)dx \approx \frac{(b-a)}{6} \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right)$$

Basic syntax, without creating more subintervals:

```
simpson_area = (b - a) / 6 * (func1(a) + 4 * func1((a + b) / 2) + func1(b));
```

Extended syntax, Riemann sum of subintervals:

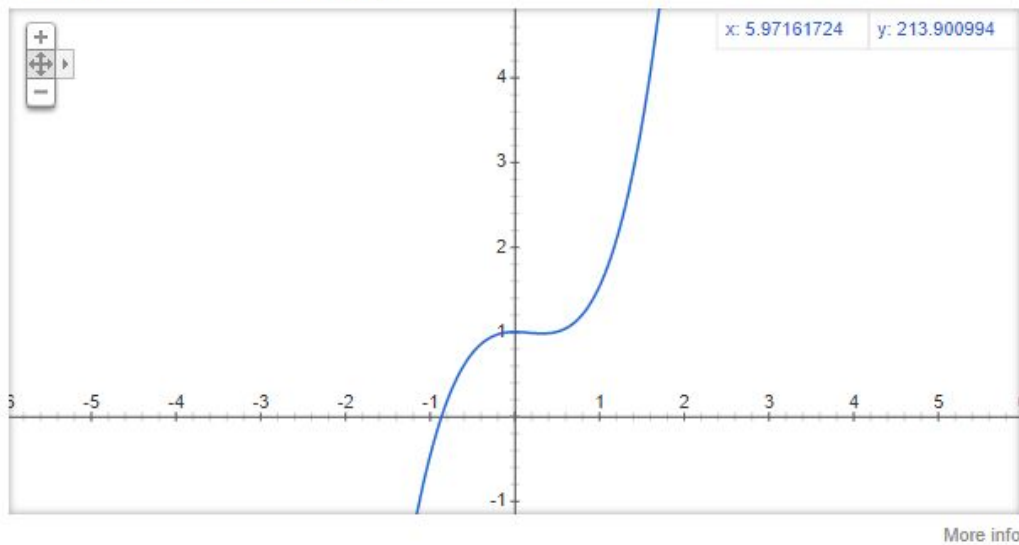
```
double width = (b - a) / n; // WIDTH OF EACH SUBINTERVAL - USED IN EACH "EXTENDED" VERSION
```

```
//a = a + i*width // b= a + (i+1)*width
```

```
for (int i = 0; i < n; i++) {  
    simpson_area += (width) / 6 * (func4(a + i * width) + 4 * func4((2*a+(2*i + 1)*width) / 2)  
    + func4(a + (i + 1)*width)) ;  
}
```

Considered function:

Graph for $x^3 + \cos(x)$



Output:

```
Func 3: x^3+cos(x)
Integration between -0.5 and 2, subintervals no: 15
Rectangular rule area=4.82665
Trapezoidal rule area=5.39592
Simpson rule area(one interval only)=5.39612
Simpson rule area=5.3731
```

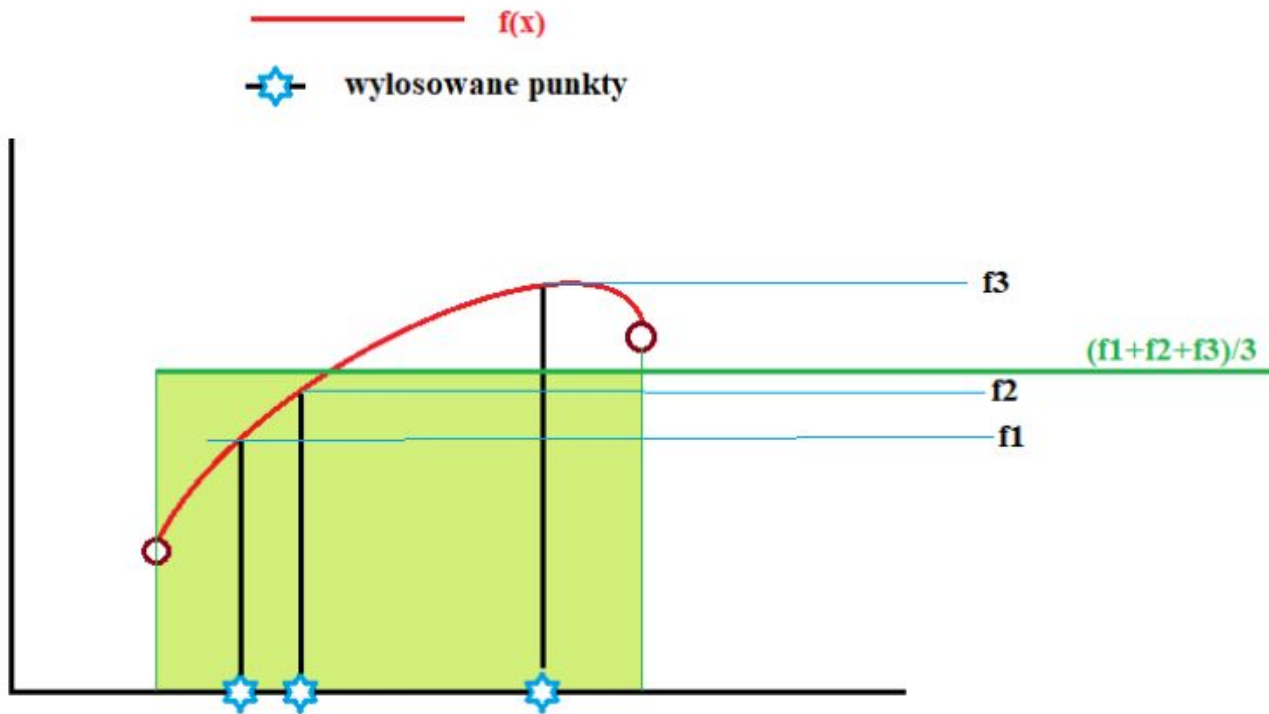
Output analysis:

The output generated in **WolframAlpha: 5.3731**.

So, Simpson Rule with 0.0001 precision in this example provides less exact result than Trapezoidal Rule.

But, when we take into consideration Simpson Rule for every subinterval, then the result is the same as from Wolfram Alpha.

2. The Monte Carlo Method



Extended syntax, Riemann sum of subintervals:

```
for (int i = 0; i < n; i++) {

    r1 = width * ((double)rand() / (double)RAND_MAX) + a + i * width;
    r2 = width * ((double)rand() / (double)RAND_MAX) + a + i * width;
    r3 = width * ((double)rand() / (double)RAND_MAX) + a + i * width;

    f_r_avg = (func4(r1) + func4(r2) + func4(r3)) / 3;
    monte_carlo_area += width * f_r_avg;

}
```

Output:

```
Integration between -0.5 and 2, subintervals no: 15
Iteration0 r:-0.499791 -0.406069 -0.467783 , width= 0.166667 a=-0.5, b=-0.333333, fravg:0.798257
Iteration1 r:-0.198543 -0.235832 -0.253354 , width= 0.166667 a=-0.333333, b=-0.166667, fravg:0.961182
Iteration2 r:-0.108285 -0.0173396 -0.0295267 , width= 0.166667 a=-0.166667, b=0, fravg:0.997419
Iteration3 r:0.124434 0.029018 0.143157 , width= 0.166667 a=0, b=0.166667, fravg:0.995501
Iteration4 r:0.285084 0.252256 0.217332 , width= 0.166667 a=0.166667, b=0.333333, fravg:0.984651
Iteration5 r:0.335831 0.348567 0.394075 , width= 0.166667 a=0.333333, b=0.5, fravg:0.982925
Iteration6 r:0.524552 0.52765 0.664754 , width= 0.166667 a=0.5, b=0.666667, fravg:1.03387
Iteration7 r:0.740949 0.686514 0.667445 , width= 0.166667 a=0.666667, b=0.833333, fravg:1.10812
Iteration8 r:0.834819 0.896313 0.921944 , width= 0.166667 a=0.833333, b=1, fravg:1.32853
Iteration9 r:1.0952 1.10029 1.10119 , width= 0.166667 a=1, b=1.16667, fravg:1.7816
Iteration10 r:1.19437 1.27717 1.2418 , width= 0.166667 a=1.16667, b=1.33333, fravg:2.22738
Iteration11 r:1.39202 1.34284 1.43461 , width= 0.166667 a=1.33333, b=1.5, fravg:2.87032
Iteration12 r:1.63055 1.63377 1.58665 , width= 0.166667 a=1.5, b=1.66667, fravg:4.18394
Iteration13 r:1.71699 1.81266 1.78778 , width= 0.166667 a=1.66667, b=1.83333, fravg:5.3771
Iteration14 r:1.99265 1.98762 1.92323 , width= 0.166667 a=1.83333, b=2, fravg:7.23954
Rectangular rule area=4.82665
Trapezoidal rule area=5.39592
Simpson rule area(one interval only)=5.39612
Simpson rule area=5.3731
Monte Carlo rule area=5.47839
```

Output analysis:

More accurate than rectangular rule result, but less than trapezoidal rule.

3. Gaussian Quadrature

$$\int_a^b f(x)dx = \frac{b-a}{2} \int_{-1}^1 f(t)dt = \frac{b-a}{2} \sum_i A_i f(t_i), \text{ gdzie } t = \frac{a+b}{2} + \frac{b-a}{2}x.$$

2 nodes:

Basic syntax:

```
double x1 = -sqrt(3) / 3;
double x2 = -x1;
double A1 = 1;
double A2 = 1;

double t1 = (a + b) / 2 + x1 * (b - a) / 2;
double t2 = (a + b) / 2 + x2 * (b - a) / 2;

gaussian_area = 2 * ((t1*t1*t1 + 2) + (t2*t2*t2) + 2);
```

Extended syntax, Riemann sum of subintervals:

```
for (int i = 0; i < n; i++) {
    double t1 = (2*a+(2*i+1)*width) / 2 + x1 * (width) / 2;
    double t2 = (2 * a + (2 * i + 1)*width) / 2 + x2 * (width) / 2;
    gaussian_area += width/2 * (func4(t1)+func4(t2));
}
```

Output:

```
Func 4: x^3+cos(x)
Integration between -0.5 and 2, subintervals no: 8
Rectangular rule area=4.38594
Trapezoidal rule area=5.45333
Simpson rule area(one interval only)=5.39612
Simpson rule area=5.3731
Monte Carlo rule area=4.94527
Gaussian quadrature rule area(one interval only)=5.35756
Gaussian quadrature rule area=5.37309
```

```
Func 4: x^3+cos(x)
Integration between -0.5 and 2, subintervals no: 9
Rectangular rule area=4.48771
Trapezoidal rule area=5.43649
Simpson rule area(one interval only)=5.39612
Simpson rule area=5.3731
Monte Carlo rule area=4.99293
Gaussian quadrature rule area(one interval only)=5.35756
Gaussian quadrature rule area=5.3731
```

Output analysis:

Considering the extended version, where we calculate the integral for each subinterval, the first result with decent precision (**5.3731**, same as in Wolfram Alpha) **we get when we divide the interval to 9 parts.**

4 nodes:

Extended version: user enters the number of parts (subintervals):

```
//points:
double d1 = -(1.0 / 35)*sqrt(525 + 70 * sqrt(30));
double d2 = -(1.0 / 35)*sqrt(525 - 70 * sqrt(30));
double d3 = (1.0 / 35)*sqrt(525 - 70 * sqrt(30));
double d4 = (1.0 / 35)*sqrt(525 + 70 * sqrt(30));
//weights:
double B1 = (1.0 / 36) * (18 - sqrt(30));
double B2 = (1.0 / 36)*(18 + sqrt(30));
double B3 = (1.0 / 36)*(18 + sqrt(30));
double B4 = (1.0 / 36)*(18 - sqrt(30));

for (int i = 0; i < n; i++) {

    double p1 = (2 * a + (2 * i + 1)*width) / 2 + d1 * (width) / 2;
    double p2 = (2 * a + (2 * i + 1)*width) / 2 + d2 * (width) / 2;
    double p3 = (2 * a + (2 * i + 1)*width) / 2 + d3 * (width) / 2;
    double p4 = (2 * a + (2 * i + 1)*width) / 2 + d4 * (width) / 2;
    gaussian_area_4points += width / 2 * (func4(p1)*B1 + func4(p2)*B2 + func4(p3)*B3 +
func4(p4)*B4);
}
```

Output:

```
Func 4: x^3+cos(x)
Integration between -0.5 and 2, subintervals no: 1
TESTTTTTT B1 0.347855
TESTTTTTT x1 -0.57735
Gaussian area: 5.3731, func4(p1)=0.912416, B1= 0.347855
Rectangular rule area=1.88146
Trapezoidal rule area=10.4205
Simpson rule area(one interval only)=5.39612
Simpson rule area=5.39612
Monte Carlo rule area=2.60152
Gaussian quadrature rule area(one interval only)(2 points)=5.35756
Gaussian quadrature rule area(2 points)=5.35756
Gaussian quadrature rule area(4 points)=5.3731
```

Output analysis:

For the considered function, even when we calculate one interval, the result has the same precision as Wolfram Alpha (5.3731).

```

Func 4: x^3+cos(x)
Integration between -0.5 and 8, subintervals no: 1
TESTTTTTT B1 0.347855
TESTTTTTT x1 -0.57735
Gaussian area: 1025.52, func4(p1)=0.996671, B1= 0.347855
Rectangular rule area=6.39695
Trapezoidal rule area=2178.58
Simpson rule area(one interval only)=1020.37
Simpson rule area=1020.37
Monte Carlo rule area=230.192
Gaussian quadrature rule area(one interval only)(2 points)=1029.37
Gaussian quadrature rule area(2 points)=1029.37
Gaussian quadrature rule area(4 points)=1025.52
Press any key to continue . . .

```

```

Func 4: x^3+cos(x)
Integration between -0.5 and 8, subintervals no: 2
TESTTTTTT B1 0.347855
TESTTTTTT x1 -0.57735
Gaussian area: 49.3307, func4(p1)=0.970474, B1= 0.347855
Gaussian area: 1025.45, func4(p1)=65.5697, B1= 0.347855
Rectangular rule area=223.832
Trapezoidal rule area=1309.92
Simpson rule area(one interval only)=1020.37
Simpson rule area=1025.79
Monte Carlo rule area=1159.64
Gaussian quadrature rule area(one interval only)(2 points)=1029.37
Gaussian quadrature rule area(2 points)=1025.22
Gaussian quadrature rule area(4 points)=1025.45
Press any key to continue . . . ■

```

The difference between Wolfram Alpha result and 4-point-gaussian-quadrature we can observe only when the range is significantly extended. Then calculating on 1 intervals is not that exact, as it is when we divide to at least two subintervals.

A10/ Solving Differential Equations. Euler & Runge-Kutta methods.

Funkcja:

```
// y(xp) = yp // warunek poczatkowy // n- liczba krokow obliczen // xk - x koncowe
void solveDE(double xp, double yp, int n, double xk) {

    double h = (xk - xp) / n; // wielkosc pojedynczego kroku
    cout << "Liczba krkokow: " << n << endl;

    de1_formula();
    cout << "Warunek poczatkowy: y(" << xp << ")=" << yp << ", krok obliczen h=" << h << ", x
koncowe xk=" << xk << endl;

    double xp_t = xp;
    double xk_t = xk;
    double yp_t = yp;

    //euler
    double a = xp;
    double b = xk;
    double c = yp;
    while ((b - a) > h / 2.0) {
        c += (h * de1(a, c));
        a += h;
        cout << "v(" << a << ")=" << c << endl;
    }
    cout << "Rozwiazanie Eulera dla y(" << a << ") to " << c << endl;

    //RK2
    double k1;
    double k2;
    double phi;

    while ((xk - xp) > h / 2.0) {
        k1 = de1(xp, yp);
        k2 = de1(xp + h, yp + h * k1);
        phi = 0.5 * (k1 + k2);
        yp += h * phi;
        xp += h;
        cout << "v(" << xp << ")=" << yp << endl;
    }
    cout << "Rozwiazanie RK2 dla y(" << xp << ") to " << yp << endl;

    //RK4
    double p1;
    double p2;
    double p3;
    double p4;
    double theta;

    while ((xk_t - xp_t) > h / 2.0) {
        p1 = de1(xp_t, yp_t);
        p2 = de1(xp_t + 0.5*h, yp_t + 0.5 * h * p1);
        p3 = de1(xp_t + 0.5*h, yp_t + 0.5 * h * p2);
        p4 = de1(xp_t + h, yp_t + h * p3);
        theta = (1.0 / 6) * (p1 + 2 * p2 + 2 * p3 + p4);
        yp_t += h * theta;
        xp_t += h;
        cout << "v(" << xp_t << ")=" << yp_t << endl;
    }
    cout << "Rozwiazanie RK4 dla y(" << xp_t << ") to " << yp_t << endl;
}
```

Wywołanie dla przykładu z zadania. $dy/dx = x^2 + y$, $y(0) = 0.1$


```

C:\Users\grzegorz\Source\Repos\NumericalMethods2\NumericalMethods3\Debug\NumericalMethods3.exe
Liczba krokow: 3
Func 1: x^2 + y
Warunek poczatkowy: y(0)=0.1, krok obliczen h=0.1, x koncowe xk=0.3
v(0.1)=0.11
v(0.2)=0.122
v(0.3)=0.1382
Rozwiazanie Eulera dla y(0.3) to 0.1382
v(0.1)=0.111
v(0.2)=0.125205
v(0.3)=0.145052
Rozwiazanie RK2 dla y(0.3) to 0.145052
v(0.1)=0.110859
v(0.2)=0.124946
v(0.3)=0.144704
Rozwiazanie RK4 dla y(0.3) to 0.144704
Press any key to continue . . .

```

Przykład obliczony analitycznie: $\frac{dy}{dx} = y-1$, $y(0) = -1$

Akcja	Równanie	Założenia:
Przekształcenie na równanie o zmiennych rozdzielonych:	$\frac{1}{y-1} dy = dx$	$y \neq 1$
Zapisanie całki ogólnej równania	$\int \frac{1}{y-1} dy = \int dx$	
Obliczenie	$\ln y-1 = x + C_1$ $ y-1 = e^x e^{C_1}$ $y-1 = \pm e^x e^{C_1}$	$C_1 \in \mathbb{R}$
Oznaczenie $e^{C_1} = C_2$	$y = 1 + C_2 e^x$	$C_2 \neq 0$
Gdy $C_2 = 0$ to $y = 1$, więc dopuszczamy taką możliwość	$y = 1 + C_2 e^x$	$C_2 \in \mathbb{R}$
Podstawiam warunek początkowy $y(0) = -1$, aby znaleźć całkę szczególną	$-1 = C_2 e^0 \Leftrightarrow C_2 = -2$	
Zapisuję całkę szczególną	$y = 1 - 2e^x$	

Rozwiązanie analityczne dla $x = 0.3$:

$$y = 1 - 2e^{0.3} = 1 - 2(1.34986) = 1.69972$$

Rozwiązania numeryczne:

```

Liczba kroków: 3
dy/dx = y-1 ;
Warunek początkowy: y(0)=-1, krok obliczeń h=0.1, x końcowe xk=0.3
v(0.1)=-1.2
v(0.2)=-1.42
v(0.3)=-1.662
Rozwiązanie Eulera dla y(0.3) to -1.662
v(0.1)=-1.21
v(0.2)=-1.44205
v(0.3)=-1.69847
Rozwiązanie RK2 dla y(0.3) to -1.69847
v(0.1)=-1.21034
v(0.2)=-1.44281
v(0.3)=-1.69972
Rozwiązanie RK4 dla y(0.3) to -1.69972

```

Jak widać rozwiązanie dla wybranego kroku obliczeń 0.1, oraz przedziału od 0 do 0.3 jest takie samo dla metody RK4 oraz analitycznie.

Dla kroku obliczeń 0.02 jest tak samo, jednak z metody RK2 oraz Eulera wyniki są bardziej dokładne przy większej ilości kroków.

```

Liczba kroków: 15
dy/dx = y-1 ;
Warunek początkowy: y(0)=-1, krok obliczeń h=0.02, x końcowe xk=0.3
v(0.02)=-1.04
v(0.04)=-1.0808
v(0.06)=-1.12242
v(0.08)=-1.16486
v(0.1)=-1.20816
v(0.12)=-1.25232
v(0.14)=-1.29737
v(0.16)=-1.34332
v(0.18)=-1.39019
v(0.2)=-1.43799
v(0.22)=-1.48675
v(0.24)=-1.53648
v(0.26)=-1.58721
v(0.28)=-1.63896
v(0.3)=-1.69174
Rozwiązanie Eulera dla y(0.3) to -1.69174
v(0.02)=-1.0404
v(0.04)=-1.08162
v(0.06)=-1.12366
v(0.08)=-1.16656
v(0.1)=-1.21033
v(0.12)=-1.25498
v(0.14)=-1.30053
v(0.16)=-1.347
v(0.18)=-1.39441
v(0.2)=-1.44277
v(0.22)=-1.49212
v(0.24)=-1.54246
v(0.26)=-1.59382
v(0.28)=-1.64621
v(0.3)=-1.69966
Rozwiązanie RK2 dla y(0.3) to -1.69966
v(0.02)=-1.0404
v(0.04)=-1.08162
v(0.06)=-1.12367
v(0.08)=-1.16657
v(0.1)=-1.21034
v(0.12)=-1.25499
v(0.14)=-1.30055
v(0.16)=-1.34702
v(0.18)=-1.39443
v(0.2)=-1.44281
v(0.22)=-1.49215
v(0.24)=-1.5425
v(0.26)=-1.59386
v(0.28)=-1.64626
v(0.3)=-1.69972
Rozwiązanie RK4 dla y(0.3) to -1.69972

```