НАД – Нумеричка анализа

Предиспитни рад

Студент: Драгана Нинковић, 2019/0052

Одсек: Системи и сигнали

Шифра предмета:13Е082НАДС

Електротехнички факултет, *22.12.2020,* Београд

Задатак 1.1 Одредити мањи корен једначине

$$x^2 - 1000x + 1 = 0$$

Применом квадратне формуле, методе половљења интервала, методе просте итерације и Њутнове методе. Коментарисати добијена решења и представити одговарајуће графике.

Решење:

За наведене алгоритме биће нам потребни позив функције, њеног извода, помоћна функција у случају методе просте итерације, метода за почетак итерације која налази крај интервала који задовољава услове конвергенције Њутнове методе , као и налажење интервала на коме се решење налази.

Кодови за наведене помоћне функције дати су на сликама:

```
double funkcija(double x)
{
    return x * x - 1000 * x + 1;
}

double funkcija_g(double x)
{
    return 0.001 * (x * x + 1);
}

double prvi_izvod(double x)
{
    return 2 * x - 1000;
}
```

```
double drugi_izvod(double x)
{
    return 2;
}

int pocetak_iteracije(double a, double b)
{
    if (funkcija(a) * drugi_izvod(a) > 0)
        return a;

    if (funkcija(b) * drugi_izvod(b) > 0)
        return b;
}
```

```
bool provera(double a, double b)
{
    if (funkcija(a) * funkcija(b) < 0)
        return true;

    return false;
}
```

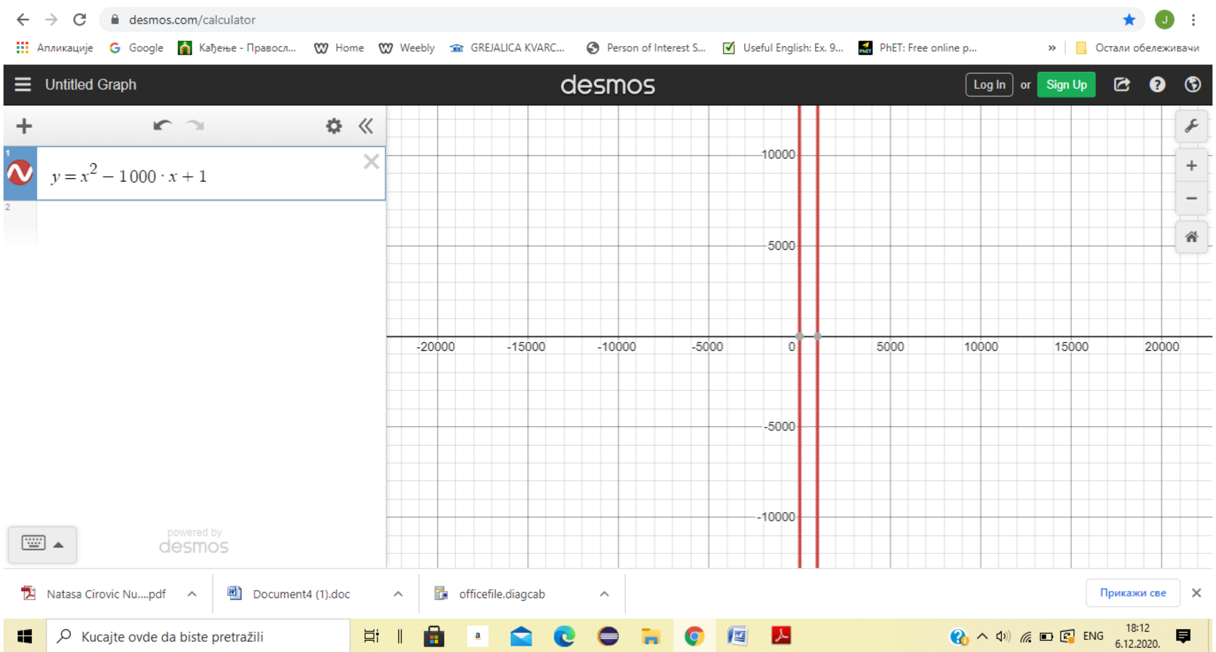
Интервал смо налазили тако да је производ функције у тачкама a и b негативан чиме смо испунили услов да функција мења знак и осигурали да се на том интервалу налази наша нула.

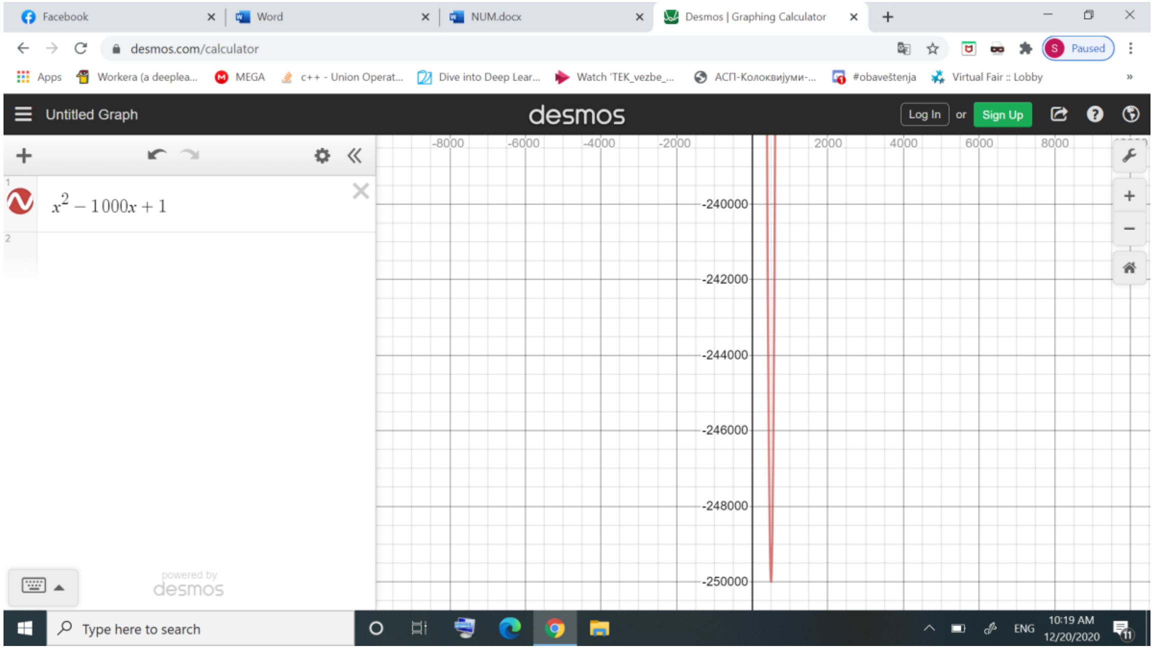
```
int main()
{
    double niz[100], epsilon = 0.00001, a, b;
    int testovi[5] = { 1, 2, 3, 4 }, j = 0;

    for (int i = 0; i < 100; i++)
        niz[i] = i;

    while (j < 100)
    {
        if ((funkcija(niz[j]) > 0 && funkcija(niz[j + 1]) < 0) || (funkcija(niz[j]) < 0 && funkcija(niz[j + 1]) > 0))
        {
            a = niz[j];
            b = niz[j + 1];
            break;
        }
        j++;
    }
```

Прво ћемо наћи корен једначине применом формуле за квадратну једначину како бисмо могли да упоредимо прецизности осталих метода. Код за налажење решења је:

```
void kvadratna_formula()
{
    double x_1, x_2;
    x_1 = 0.5 * (1000 + sqrt(1000 * 1000 - 4));
    x_2 = 0.5 * (1000 - sqrt(1000 * 1000 - 4));

    if (x_1 < x_2)
        printf("Resenje primenom kvadratne formule je %lf.\n", x_1);
    else
        printf("Resenje primenom kvadratne formule je %lf.\n", x_2);
}
```

А график функције





А резултат који се добија је 0.001000.

1) Метод половљења интервала

Када знамо интервал на коме је наша нула лоцирана наћи ћемо колика је функција у тачки на његовој средини. Имамо два случаја. Први је да је функција у тој тачки једнака нули чиме је она нула функције. Други да је различита од нуле и тада та тачка и нека од тачака крајева интервала заједно чине подинтервал на коме је лоцирана наша нула. Коју тачку бирамо, одлучујемо на основу производа вредности функције у средишту интервала и крајевима интервала и бирамо ону за коју је тај производ негативан из претходно наведеног разлога. Овај поступак се понавља рекурзивно и што је већи број примењивања то је резултат прецизнији. Грешка резултата дата је формулом:

$$|x^* - \bar{x}_n| \leq \frac{b_n - a_n}{2} \leq \frac{b_0 - a_0}{2^{n+1}} \leq \varepsilon.$$

Из које лако можемо одредити зависност прецизности од броја итерација. Такође приметимо да је пожељно кренути од што мањег интервала.
У коду ћемо са p обележити бројач који нам говори до које итерације смо стигли, а са i и j крајеве интервала при чему је i почетак а j крај интервала. У променљивој претходни чувамо решење које смо добили у претходној итерацији, и у свакој итерацији исписујемо претходно и тренутно добијено решење.

```c
void metoda_polovljenja_intervala(double a, double b, double epsilon)
{
    int p = 0;
    double t, x_0, i, j;
    double prethodni = 0;

    for (i = a, j = b, p = 0; i <= j; p++)
    {
        x_0 = (i + j) / 2;

        printf("%d. iteracija:\n", p);
        printf("%lf\t%lf\n", prethodni, x_0);
```

Прво гранање нам проверава који од наведена два случаја је у питању, а угњеждено гранање у оквиру другог случаја поставља нове крајеве на одговарајуће на горе описан начин.

```c
if (funkcija(x_0) == 0)
{
    printf("Resenje sa tacnoscu %lf je %lf.\n", epsilon, x_0);
    break;
}
else
{
    if (provera(x_0, j))
        i = x_0;

    else
        j = x_0;
}
```

Последње гранање проверава да ли је разлика суседних итерација мања од тражене тачности, ако јесте то значи да смо завршили задатак и треба да испишемо решење. Ако није настављамо даље, при чему у променљивој претходни треба да се нађе наше тренутно решење.

```c
        t = prethodni - x_0;

        if (fabs(t) < epsilon)
        {
            printf("Resenje sa tacnoscu %lf je %lf.\n", epsilon, x_0);
            break;
        }

        prethodni = x_0;
    }
}
```

Решење до којег долазимо овим поступком је:

```
/*=======================================*/
/*METODA POLOVLJENJA INTERVALA*/
0. iteracija:
0.000000        0.500000
1. iteracija:
0.500000        0.250000
2. iteracija:
0.250000        0.125000
3. iteracija:
0.125000        0.062500
4. iteracija:
0.062500        0.031250
5. iteracija:
0.031250        0.015625
6. iteracija:
0.015625        0.007813
7. iteracija:
0.007813        0.003906
8. iteracija:
0.003906        0.001953
```

```
9. iteracija:
0.001953        0.000977
10. iteracija:
0.000977        0.001465
11. iteracija:
0.001465        0.001221
12. iteracija:
0.001221        0.001099
13. iteracija:
0.001099        0.001038
14. iteracija:
0.001038        0.001007
15. iteracija:
0.001007        0.000992
16. iteracija:
0.000992        0.000999
Resenje sa tacnoscu 0.000010 je 0.000999.
/*=======================================*/
```

Приметимо да нам је било потребно чак 16 итерација и да смо добили решење приближно исто као применом саме формуле.

2) Метода просте итерације

Проблем налажења нуле функције се може преформулисати у проблем тражења непокретних тачака функције.

Теорема Нека је функција g непрекидна на [a,b] и a≤ g(x)≤b за свако x ∈[a,b] . Додатно нека је g диференцијабилна на (a,b) и нека постоји константа 0<k<1 таква да је |g’(x)|≤k за свако x ∈[a,b]. Тада за произвољно x_0 ∈ [a, b] низ дефинисан са x_n = g(x_{n-1}),(n ∈ N) конвергира ка јединственој тачки .

За грешку важи релација:

$$|x_n - x^*| \leq \frac{k^n}{1-k}|x_1 - x_0|, \quad (\forall x \in (a,b)) .$$

$$\frac{k^n}{1-k}|x_1 - x_0| \leq \varepsilon.$$

из чега поново као у претходној методи можемо лако добити зависност тачности и броја итерација.

У датом програму променљиве Xk, и Xk_1 представљају решења добијена у тренутној и претходној итерацији. С обзиром да немамо никакав додатан услов за почетну тачку, за њу ћемо узети почетак првог интервала.

```
void metoda_proste_iteracije(double a, double b, double epsilon)
{
    int k = 0;
    double Xk, Xk_1;

    Xk_1 = a;

    printf("Iterativni proces:\n");
    printf("k \t\t X_k\n");
    printf("%d \t\t %lf\n", k, Xk_1);

    Xk = Xk_1;
    Xk_1 = funkcija_g(Xk);
    k++;
```

Рекурзивно примењујемо дату формулу све док разлика решења две суседне итерације не буде мања од тражене тачности

```
    Xk_1 = funkcija_g(Xk);
    k++;

    while (fabs(Xk - Xk_1) > epsilon && funkcija_g(Xk_1) != 0)
    {
        printf("%d \t\t %lf\n", k, Xk_1);
        Xk = Xk_1;
        Xk_1 = funkcija_g(Xk);
        k++;
    }

    printf("%d \t\t %lf\n", k, Xk_1);
    printf("Resenje sa tacnoscu %lf je %lf.\n", epsilon, Xk_1);

}
```

Овим поступком добијамо следеће решење:

```
/*========================================*/
/*METODA PROSTE ITERACIJE*/
Iterativni proces:
k               X_k
0               0.000000
1               0.001000
2               0.001000
Resenje sa tacnoscu 0.000010 je 0.001000.
/*========================================*/
```

Приметимо да смо овим методом имали само три итерације и да смо добили решење с великом тачношћу.

3) Њутнова метода

Нека је функција f два пута непрекидно диференцијабилна на интервалу и нека је $x_0 \in$ [a, b] почетна апроксимација решења.Тада можемо применити тејлоров развој функције f у околини тачке x_0 из чега добијамо

$$f(x) = f(x_0) + f'(x_0)\,(x - x_0) + \frac{f''(\xi)}{2!}\,(x - x_0)^2,$$

Ако функцију апроксимирамо линеанром функцијом у околини x_0 (што је заправо њена тангента у тој тачки) и посматрамо њену вредност у нули добијамо:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Када рекурзивно применимо овај поступак добијамо Њутнову методу:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Теорема о конвергенцији Њутнове методе

Нека је функција $f = C^2[a,b]$и нека важи да је:

1) $f(a)f(b) < 0$
2) $f'(x) \neq 0, \forall x \epsilon [a,b]$
3) f'' не мења знак на интервалу $[a,b]$

Уколико за почетну тачку итерације изаберемо $x_0 \epsilon [a,b]$ такво да важи

$$f(x_0)f''(x_0) > 0$$

Тада итеративни низ дефинисан Њутновом методом ковергира ка јединственом решењу x* једначине $f(x) = 0$.

При томе важи следећа апсолутна оцена грешке

$$|x^* - x_n| \leq \frac{M_2}{2m_1}(x_n - x_{n-1})^2,$$

Све ознаке су нам исте као у претходној методи. Метода почетак итерације нам налази који од крајева задовољава услов конвергенције Њутнове методе и враћа га као повратну вредност.

```
void Njutnova_metoda(double a, double b, double epsilon)
{
    double Xk, Xk_1;
    int k = 0;

    printf("Iterativni proces:\n");

    Xk = pocetak_iteracije(a, b);

    printf("k \t\t Xk\n");
    printf("%d \t\t %lf\n", k, Xk);

    Xk_1 = Xk - funkcija(Xk) / prvi_izvod(Xk);
```

У свакој итерацији примењујемо рекурзивну формулу и проверсвсмо да ли смо постигли тражену тачност

```c
    Xk_1 = Xk - funkcija(Xk) / prvi_izvod(Xk);

    while (fabs(Xk - Xk_1) > epsilon && funkcija(Xk_1) != 0)
    {
        k++;
        printf("%d \t\t %lf\n", k, Xk_1);
        Xk = Xk_1;
        Xk_1 = Xk - funkcija(Xk) / prvi_izvod(Xk);
    }

    printf("%d \t\t %lf\n", k + 1, Xk_1);
    printf("Resenje sa tacnoscu %lf je %lf.\n", epsilon, Xk_1);
}
```

```
/*=========================================*/
/*NJUTNOVA METODA*/
Iterativni proces:
k                 Xk
0                 0.000000
1                 0.001000
2                 0.001000
Resenje sa tacnoscu 0.000010 je 0.001000.
/*=========================================*/
```

Приметнимо да Њутновом методом добијамо решење из два корака са највећом тачношћу од свих примењених метода.

Задатак 2.2 Дата је Хилбертова матрица A димензије n

$$\mathbf{A} = \begin{bmatrix} 1 & 1/2 & 1/3 & \cdots \\ 1/2 & 1/3 & 1/4 & \cdots \\ 1/3 & 1/4 & 1/5 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}.$$

Развити програм који решава систем једначина Ax = b методом LU декомпозиције где је A хилбертова матрица произвољне димензије n . Једини улаз који програм добија од корисника је димензија система n. Користећи развијени програм одредити највеће n за које је добијено решење тачно на 5 децимала у односу на тачно решење.

Решење:

Матрица A може се представити у облику A = LU где је L горње троугаона матрица а U доње троугаона матрица. Зато када нађемо матрице L и U за које је ово задовољено добијамо систем Ly=A и Ux=y који се лако може решити поступцима решавања у напред и у назад тако да дати проблем сводимо на налажење ових матрица.

Код за формирање саме Хилбертове матрице и слободног члана дат је са:

```c
}

double* formirajB(double** A) {

    double* b = alocirajNiz();

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            b[j] += A[i][j];
        }
    }
    return b;
}
```

```c
void formirajHilberta(double** matrix) {

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

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

            matrix[i][j] = 1.0 / (i + j + 1);
        }
    }

}
```

Поступци решавања у напред и у назад:

```c
double* resavanjeGornjeTrougaone(double** A, double* b) {
    double* x = (double*)calloc(n, sizeof(double));

    for (int i = n-1; i >=0; i--) {

        x[i] = (1 / A[i][i]) * (b[i] - sumI(A, x, i));


    }
    return x;
}
```

```c
double sumI(double** L, double** U, int i, int j) {

    double sum = 0;

    for (int k = 0; k <=(i - 1); k++) {
        sum += L[i][k] * U[k][j];
    }

    return sum;
}
```

```c
double* resavanjeDonjeTrougaone(double** A, double* b)
{
    double* x = (double*)calloc(n , sizeof(double));

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

        x[i] = (1 / A[i][i]) * (b[i] - sumIK(A, x, i));
    }

    return x;
}
```

```
double sumIK(double** A, double* b, int i)
{
    double sum = 0;
    for (int k = 0; k < i; k++)
    {
        sum += A[i][k] * b[k];

    }

    return sum;
}
```

Ове две матрице можемо наћи на три начина:

1. Применом Гаусове методе елиминације
2. Применом Краутовог алгоритма
3. Применом Дулитловог алгоритма

С обзиром да је решавање Гаусовом методом елиминације најспорије и најмање концизно, решићемо задатак само на друга два начина.

Краутов алгоритам

Поступак је следећи:

1) Постављамо јединице на дијагонали U (прва петља)
2) Прва врста матрице L једнака је првој врсти матрице A

```
for (int i = 0; i < n; i++) {
    U[i][i] = 1;
    L[i][0] = a[i][0];
}
```

3) Прва колона матрице U једнака је:

```
for (int j = 1; j < n; j++) {
    U[0][j] = a[0][j] / L[0][0];
}
```

4) Остали елементи у колони j матрице L једнаки су

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \ (i = j, \ldots, n)$$

У коду је ово реализовано на следећи начин:

```
for (int x = j; x < n; x++) {
    L[x][j] = a[x][j] - sumJ(L, U, x, j);
}
```

```
double sumJ(double** L, double** U, int i, int j) {
    double sum = 0;

    for (int k = 0; k <=(j - 1); k++) {
        sum += L[i][k] * U[k][j];
    }

    return sum;
}
```

5) Остали елементи матрице U у врсти i дати су формулом

$$u_{ij} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right), \quad (j = i+1, \ldots, n)$$

А у коду је реализован:

```
for (int y = i + 1; y < n; y++) {
    U[i][y] = 1 / (L[i][i]) * (a[i][y] - sumI(L, U, i, y));
}
```

```
double sumI(double** L, double** U, int i, int j) {

    double sum = 0;

    for (int k = 0; k <=(i - 1); k++) {
        sum += L[i][k] * U[k][j];
    }

    return sum;
}
```

6) Овај поступак се понавља све док не прођемо кроз све врсте и колоне матрице A

Цео код дат је на слици:

```
NAD_seminarski                    (Global Scope)
void KrautovAlgoritam(double** a, double** L, double** U) {
    for (int i = 0; i < n; i++) {
        U[i][i] = 1;
        L[i][0] = a[i][0];
    }

    for (int j = 1; j < n; j++) {
        U[0][j] = a[0][j] / L[0][0];
    }

    for (int j = 1, i = 1; j < n && i < n; j++, i++) {
        for (int x = j; x < n; x++) {
            L[x][j] = a[x][j] - sumJ(L, U, x, j);
        }

        for (int y = i + 1; y < n; y++) {
            U[i][y] = 1 / (L[i][i]) * (a[i][y] - sumI(L, U, i, y));
        }
    }
}
```

Дулитлов алгоритам

Поступак је следећи:

1) Постављамо јединице на дијагонали L (прва петља)
2) Прва колона матрице U једнака је првој врсти матрице A

```
for (int j = 0; j < n; j++) {
    L[j][j] = 1;
    U[0][j] = a[0][j];
}
```

3) Прва врста матрице L једнака је:

```
for (int i = 1; i < n; i++) {
    L[i][0] = a[i][0] / U[0][0];
}
```

4) Остали елементи у колони i матрице U једнаки су

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}, \ (j = i, \ldots, n)$$

и то је у коду реализовано као:

```
for (int y = i; y < n; y++) {
    U[i][y] = a[i][y] - sumI(L, U, i, y);
}
```

```
double sumI(double** L, double** U, int i, int j) {

    double sum = 0;

    for (int k = 0; k <=(i - 1); k++) {
        sum += L[i][k] * U[k][j];
    }

    return sum;
}
```

5) Остали елементи матрице L у реду i дати су формулом

$$l_{ij} = \frac{1}{u_{ii}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right), \\ (i = j+1, \ldots, n)$$

Што је у коду реализовано као:

```
for (int x = j + 1; x < n; x++) {
    L[x][j] = 1 / (U[j][j]) * (a[x][j] - sumJ(L, U, x, j));
}
```

```
double sumJ(double** L, double** U, int i, int j) {
    double sum = 0;

    for (int k = 0; k <=(j - 1); k++) {
        sum += L[i][k] * U[k][j];
    }

    return sum;
}
```

6) Овај поступак се понавља све док не прођемо кроз све врсте и колоне матрице А

Цео код дат је на слици:

```
void DulitlovAlgoritam(double** a, double** L, double** U) {

    for (int j = 0; j < n; j++) {
        L[j][j] = 1;
        U[0][j] = a[0][j];
    }
    for (int i = 1; i < n; i++) {
        L[i][0] = a[i][0] / U[0][0];
    }
    for (int j = 1, i = 1; j < n && i < n; j++, i++) {
        for (int y = i; y < n; y++) {
            U[i][y] = a[i][y] - sumI(L, U, i, y);
        }
        for (int x = j + 1; x < n; x++) {
            L[x][j] = 1 / (U[j][j]) * (a[x][j] - sumJ(L, U, x, j));
        }

    }
}
```

Најмање н за које је решење тачно на 5 децимала добијамо решавањем система за све н-ове докле год је разлика нашег решења и правог мања од 0.00001 по свакој координати. Решавање се може урадити преко Дулитловог и Краутовог алгоритма као и мало пре.

```
Unesite broj koji odgovara akciji koja zelite da se desi:
1.RESAVANJE SISTEMA ZA KONKRETNO N KORISCENJEM DULITLOVOG ALGORITMA
2.RESAVANJE SISTEMA ZA KONKRETNO N KORISCENJEM KRAUTOVOG ALGORITMA
3.NALAZENJE NAJVECEG N ZA KOJE JE RESENJE TACNO NA 5 DECIMALA DULITLOVIM ALGORITMOM
4.NALAZENJE NAJVECEG N ZA KOJE JE RESENJE TACNO NA 5 DECIMALA KRAUTOVIM ALGORITMOM
3
1.000000
1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000002 0.999992 1.000015 0.999983 1.000010 0.999997
Najvece n za koje je resenje tacno na 5 decimala dobijeno Dulitlovim algotirmom je 8

C:\Users\test4\Desktop\NAD_seminarski\Debug\NAD_seminarski.exe (process 20584) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Microsoft Visual Studio Debug Console

```
Unesite broj koji odgovara akciji koja zelite da se desi:
1.RESAVANJE SISTEMA ZA KONKRETNO N KORISCENJEM DULITLOVOG ALGORITMA
2.RESAVANJE SISTEMA ZA KONKRETNO N KORISCENJEM KRAUTOVOG ALGORITMA
3.NALAZENJE NAJVECEG N ZA KOJE JE RESENJE TACNO NA 5 DECIMALA DULITLOVIM ALGORITMOM
4.NALAZENJE NAJVECEG N ZA KOJE JE RESENJE TACNO NA 5 DECIMALA KRAUTOVIM ALGORITMOM
4
1.000000
1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000001 0.999997 1.000006 0.999993 1.000004 0.999999
1.000000 1.000000 1.000002 0.999977 1.000109 0.999698 1.000497 0.999517 1.000254 0.999944
Najvece n za koje je resenje tecno na 5 decimala dobijeno Krautovim algoritmom je 9

C:\Users\test4\Desktop\NAD_seminarski\Debug\NAD_seminarski.exe (process 5712) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
```

NAD_seminarski | (Global Scope) | ResiDulitlom()

```cpp
    case 3:
        n = 0;
        do {
            if (x) { free(x); }
            n++;
            x = ResiDulitlom();
            ispisiNiz(x);
        } while (check(x));
        cout <<"Najvece n za koje je resenje tacno na 5 decimala dobijeno Dulitlovim algotirmom je "<< (n-1) << endl;
        break;
    case 4:
        n = 0;
        do {
            if (x) { free(x); }
            n++;
            x = ResiKrautom();
            ispisiNiz(x);
        } while (check(x));
        cout << "Najvece n za koje je resenje tecno na 5 decimala dobijeno Krautovim algoritmom je " << (n - 1) << endl;
        break;
    };
```