METODA RUNGE-KUTTA

Profesor: Veljko M. Milutinović

Asistent: Jelena Hadzi - Purić

Aleksandar Mužina (mr14235)

Maja Vujović (mr14296)

Matematički fakultet u Beogradu

O ALGORITMU

Metode Runge-Kutta su razviljene početkom 19. veka od strane dvojice nemačkih matematičara, Carl Runge-a i Martin Kutta.

Postoji nekoliko metoda Runge-Kutta. Mi ćemo u ovom radu razmatrati samo Runge-Kutta četvrtog reda, poznatiju kao „klasična“ Runge-Kutta metoda ili „RK4“.

Carl Runge

Martin Kutta

UPOTREBA

- Metode Runge-Kutta su važan skup implicitnih i eksplicitnih iterativnih metoda.
- Metoda Runge-Kutta služi za rešavanje Košijevih problema diferencijalnih jednačina prvog ili višeg reda.
- Koristi se u temporalnoj(vremenskoj) diskretizaciji za aproksimaciju opstih rešenja diferencijalnih jednačina.

ALGORITAM

- Početni uslovi su nam: $\dot{y} = f(t,y)$, $y(t_0) = y_0$.
- Y je nepoznata funkcija koja zavisi od vremena t, koju želimo da aproksimiramo.

  $\dot{y}$ je brzina kojom se y menja, ona zavisi od t i samog y.

  Znamo vrednost funkcije y u početnom trenutku t_0. (f, t_0, y_0 su poznate)
- Sada izaberimo korak $h > 0$ i uvedimo sledeće definicije:

$$y_{n+1} = y_n + 1/6(k_1 + 2k_2 + 2k_3 + k_4)$$

$$t_{n+1} = t_n + h$$

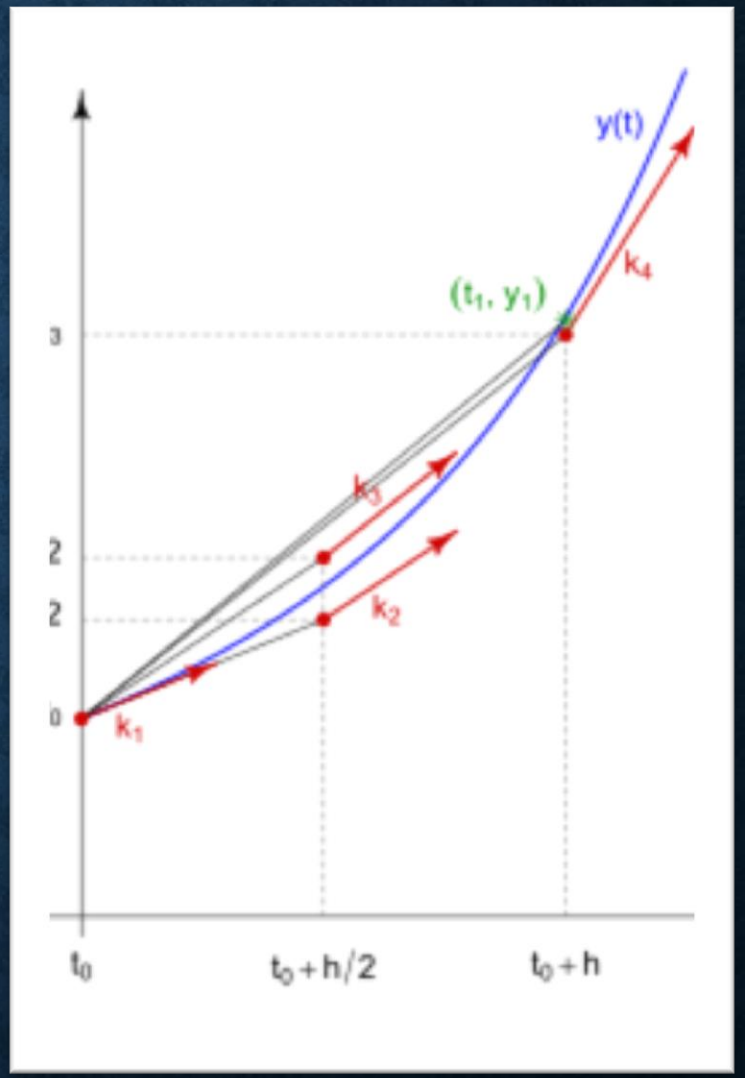
KORACI

Za n = 0,1,2,3… Računamo:

- $k_1 = h * f(t_n, y_n)$
- $k_2 = h * f(t_n + h/2, y_n + k_1/2)$
- $k_3 = h * f(t_n + h/2, y_n + k_2/2)$
- $k_4 = h * f(t_n + h, y_n + k_3)$

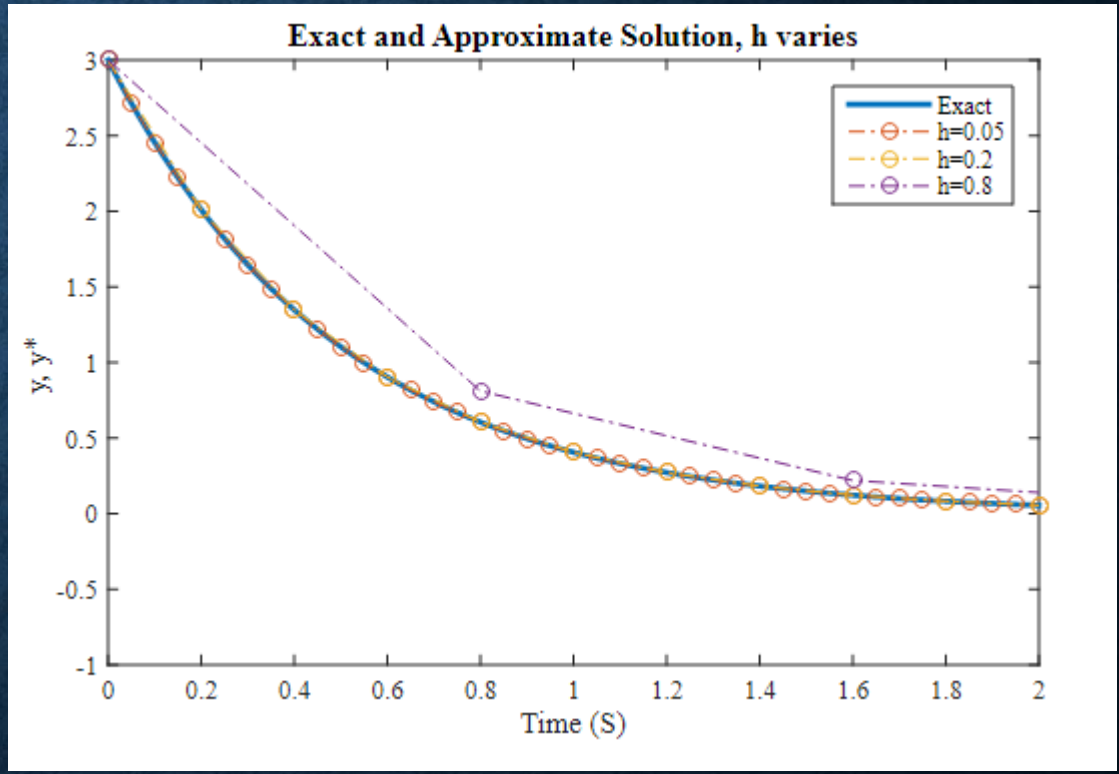
Ovde je y_{n+1} RK4 aproksimacija.

- k_1 je korak određen nagibom na početku intervala koristeći y
- k_2 je korak određen nagibom u sredini intervala koristeći $y + 1/2\ hk_1$
- $k3$ je korak određen nagibom u sredini intervala koristeći $y + 1/2\ hk_2$
- $k4$ je korak određen nagibom na kraju intervala koristeći $y + hk_3$



MATLAB KOD

```
% Rešiti y'(t)=-2y(t) sa y0=3, 4. red Runge Kutta

y0 = 3;                % Inicijalni uslovi

h=0.2;                 % Vremenski korak

t = 0:h:2;             % t je u intervalu 0 do 2 seconds.

yexact = 3*exp(-2*t)     % Egzaktno rešenje(Ovo suštinski nećemo znati)

ystar = zeros(size(t));  % Prealociranje niza

ystar(1) = y0;         % Početni uslovi dalju rešenje za t=0.

for i=1:(length(t)-1)

 k1 = -2*ystar(i)

 y1 = ystar(i)+k1*h/2;

 k2 = -2*y1

 y2 = ystar(i)+k2*h/2

 k3 = -2*y2

 y3 = ystar(i)+k3*h;

 k4 = -2*y3

 ystar(i+1) = ystar(i) + (k1+2*k2+2*k3+k4)*h/6;

end

plot(t,yexact,t,ystar);

legend('Exact','Approximate');
```



CPU KOD

CPUCode/MovingAverageSimpleCpuCode.c

Save | Undo | Redo | Settings

```c
/**
 * MaxFile name: MovingAverageSimple
 * Koristimo 4-ti red metode Runge-Kutta da izracunamo sledecu tacku y vrednosti
 * Racunamo jednu po jednu za 16 vrednosti y, koja je data dormulom 3y^2 + 2
 * Poredimo nas rezultat sa ocekivanom vrednoscu
 */
#include "Maxfiles.h"
#include <MaxSLiCInterface.h>

// Ulazne vrednosti
const int size = 16;
float h = 2.0;
float x = 1.0;
const float k[16] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
const float y[16] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
float y_output[16] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
float expected_y_output[16] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

// Implementiramo metod izvoda kako bi izracunali izvod funkcije u bilo kojoj tacki
// dy/dx = 3y^2 + 2
static void calc_derivation(float x, float *y, float *k, int size){
  for(int j = 0 ; j < size ; j++){
    k[j] = 3.0 * y[j] * y[j] + 2.0;
  }
}
```

CPUCode/MovingAverageSimpleCpuCode.c

Save | Undo | Redo | Settings

```c

//Ovde implementiramo 4-ti red metode Runge-Kutta da izracunamo stvarnu ocekivanu vrednost izlaza
static void calc_order4_runge_kutta(const float y[], const float k[], int size,
                                    float x, float h, float y_output[]){
   //U y_output upisujemo rezultat
  float h_h, h_six, x_h, *y_t, *dy_t, *dy_m;

//Pomocne promenjive
  h_h = h * 0.5;
  h_six = h / 6.0;
  x_h = x + h_h;

  // add vector
  y_t = malloc(sizeof(void*) * size);
  dy_t = malloc(sizeof(void*) * size);
  dy_m = malloc(sizeof(void*) * size);

  for(int j = 0 ; j < size ; j++){
    y_t[j] = y[j] + h_h * k[j];             //PRVI KORAK
  }
  calc_derivation(x_h, y_t, dy_t, size);    //DRUGI KORAK

  for(int j = 0 ; j < size ; j++){
    y_t[j] = y[j] + h_h * dy_t[j];          //yt[i]=y[i]+hh*dyt[i]
  }
  calc_derivation(x_h, y_t, dy_m, size);    //TRECI KORAK
```

```c
  for(int j = 0 ; j < size ; j++){
    y_t[j]  = y[j] + h * dy_m[j];           //yt[i]=y[i]+h*dym[i]
    dy_m[j] = dy_m[j] + dy_t[j];            //dym[i] += dyt[i]
  }
  calc_derivation(x + h, y_t, dy_t, size);  //CETVRTI KORAK

  for(int j = 0 ; j < size ; j++){
    // Ovde je poslednji korak u kome koristimo prethodne 4 dobijene vrednosti!
    y_output[j] = y[j] + h_six * (k[j] + dy_t[j] + 2.0 * dy_m[j]);
  }
}

int main()
{
    printf("Running DFE Fourth Order Runge Kutta Method.\n");

    //Uzimimo stvarne vrednosti za y koristeci 4-ti red metode Runge-Kutta
    MovingAverageSimple(size, h, x, k, y, y_output);

    //Uzmimo ocekivane vrednosti za y koristeci 4-t red metode Runge-Kutta
    calc_order4_runge_kutta(y, k, size, x, h, expected_y_output);
```

```c
  //Uporedjujemo stvarne i ocekivane vrednosti
  for(int j=0; j < size; j++)
  {
     if(y_output[j] != expected_y_output[j]) {
        fprintf(stderr, "@ point %d: output y is %1.8g; expected output y is %1.8g. small 7th decimal error. \n",
              j, y_output[j], expected_y_output[j]);
     }else {
        fprintf(stderr, "@ point %d: output y is %1.8g; expected output y is %1.8g. no error. \n",
              j, y_output[j], expected_y_output[j]);
     }
  }
  return 0;
}
```

KERNEL KOD

EngineCode/src/movingaveragesimple/MovingAverageSimpleKernel.maxj

Save | Undo | Redo | Settings

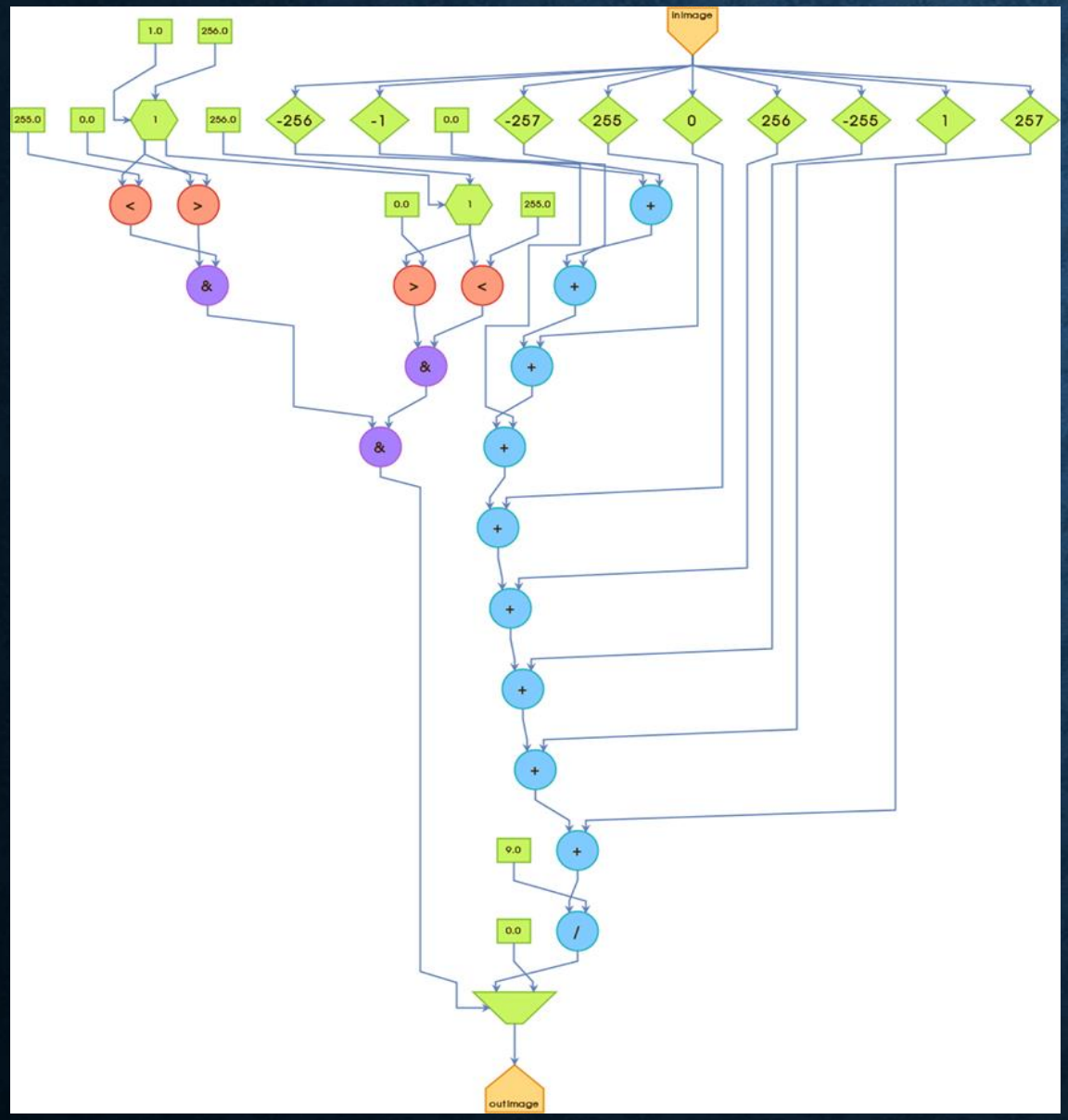
```
/**
 * MaxFile name: MovingAverageSimple
 * Koristimo 4-ti red metode Runge-Kutta da izracunamo sledecu tacku y vrednosti
 * Racunamo jednu po jednu za 16 vrednosti y, koja je data dormulom 3y^2 + 2
 * Poredimo nas rezultat sa ocekivanom vrednoscu
 */

package movingaveragesimple;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class MovingAverageSimpleKernel extends Kernel {

    private static final DFEType type = dfeFloat(8,24);

    MovingAverageSimpleKernel(KernelParameters parameters) {
        super(parameters);

```

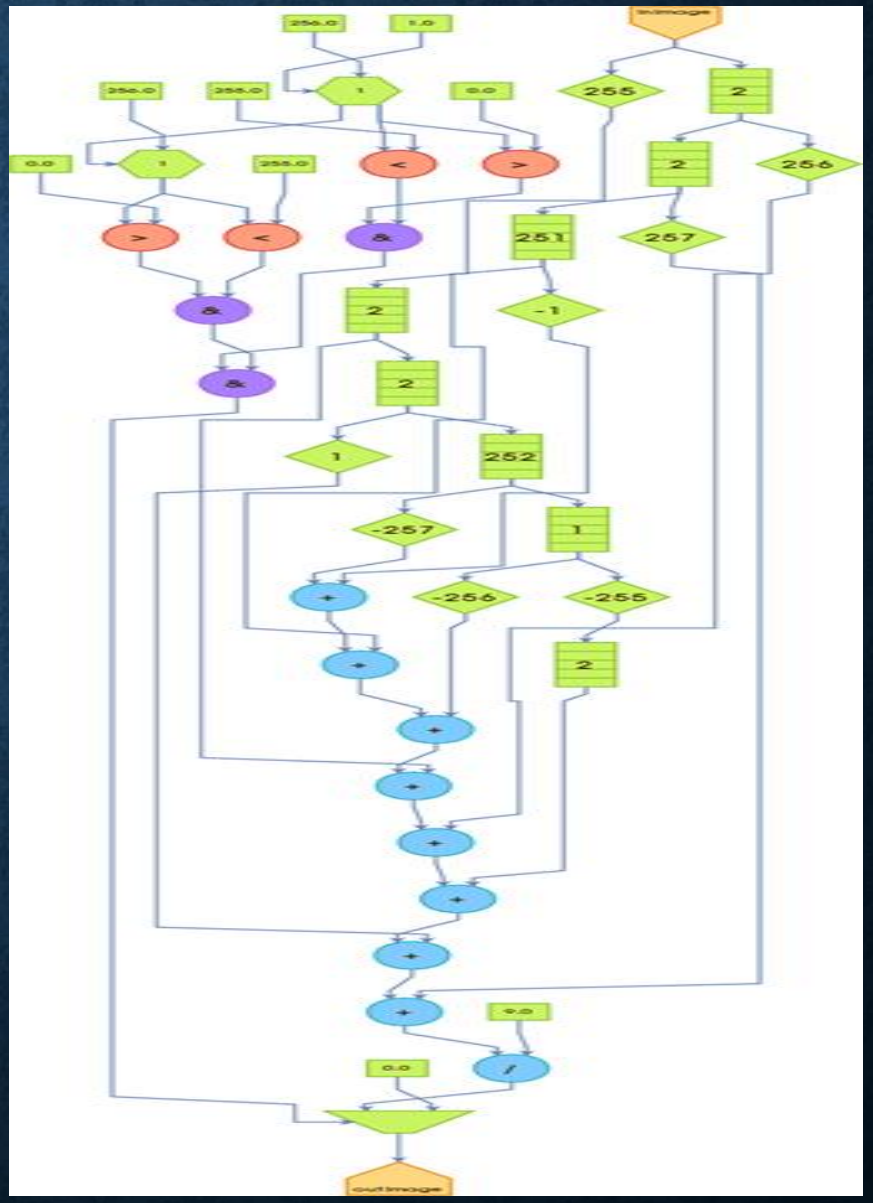
EngineCode/src/movingaveragesimple/MovingAverageSimpleKernel.maxj

Save | Undo | Redo | Settings

```

        // Ulazne vrednosti
        /*
           k - izvod od y dat sa x(dy/dx)
           y - vrednost y za dato x
           x - koordinate tacke x
           h - interval koji smo izabrali
        */
        DFEVar h = io.scalarInput("h", type);   //konstanta
        DFEVar x = io.scalarInput("x", type);   //konstanta
        DFEVar k = io.input("dy_dx", type);
        DFEVar y = io.input("y", type);

        // Inicijalizujemo ostale pomocne promenjive
        DFEVar y_t, dy_t, dy_m, h_h, h_six, x_h;

        h_h = h * 0.5;
        h_six = h / 6.0;
        x_h = x + h_h;

        /*
         Algoritam za 4-ti red metode Runge-Kutta proverava da li radi kako treba
        */
        y_t = y + h_h * k;                  //PRVI KORAK
        dy_t = 3.0 * y_t * y_t + 2.0;       //DRUGI KORAK

        y_t = y + h_h * dy_t;
        dy_m = 3.0 * y_t * y_t + 2.0;       //TRECI KORAK

        y_t  = y + h * dy_m;
        dy_m = dy_m + dy_t;
        dy_t = 3.0 * y_t * y_t + 2.0;       //CETVRTI KORAK

        // yout[i]=y[i]+h6*(dydx[i]+dyt[i]+2.0*dym[i])
        DFEVar result = y + h_six * (k + dy_t + 2.0 * dy_m);

        //Izlazna vrednost
        io.output("y_output", result, type);

    }
}
```

GRAFOVI


inImage
1.0
256.0
255.0
0.0
1
256.0
-256
-1
0.0
-257
255
0
256
-255
1
257
<
>
&
0.0
1
255.0
+
9.0
0.0
/
outImage


255
2
256
251
257
-1
252
-257
-256
-255

HVALA NA PAŽNJI