Praktikum 1 : DGL

Oliver Steenbuck, Karolina Bernat

31.10.2012

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Steife Differentialgleichungen

Gegeben ist folgende Differentialgleichung 1. Grades an der mit Matlab und Simulink drei verschiedene Approximationsverfahren (Explizites und implizites Euler Verfahren sowie das Verfahren von Runge Kutta 2. Ordnung) für steife Differentialgleichungen angewendet werden sollen.

1.1 Gleichung

$$y(0) = 1 \tag{1}$$
$$y' = 10 - 500 \cdot y + 5000 \cdot x \tag{2}$$

1.2 Simulink

Die in 1.1 gezeigte Gleichung wurde zuerst in Simulink modelliert.

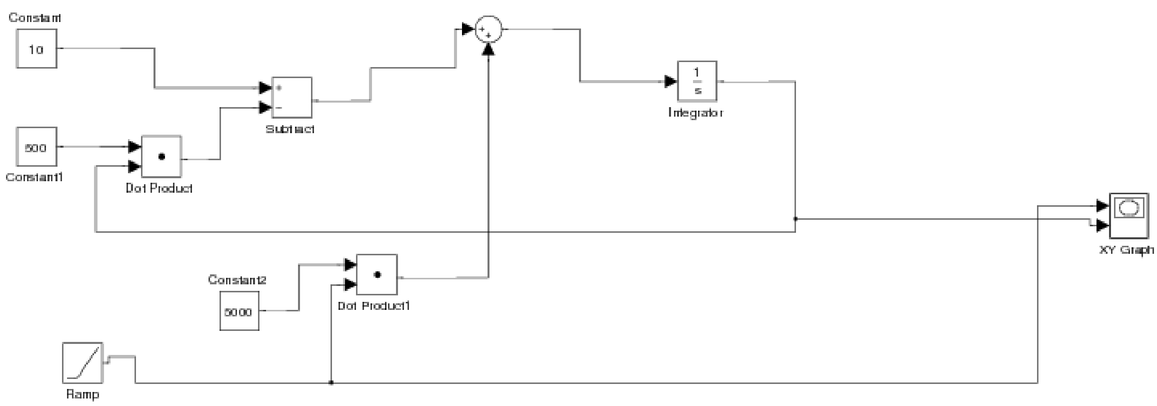

Abbildung 1: Steife Differentialgleichung Simulink

1.3 Iterationslgeichungen

Untenstehend wurden die für die jeweiligen Verfahren notwendigen Iterationsgleichungen hergeleitet und die Verfahren in Matlab implementiert, siehe 4.

1.3.1 Euler, explizit

Eingesetzt in das explizite Euler Verfahren ergibt sich folgende Iterationsgleichung.

$$y(0) = 1 \tag{3}$$
$$y_{j+1} = y_j + h \cdot (10 - 500 \cdot y_j + 5000 \cdot x_j) \tag{4}$$

1.3.2 Euler, implizit

Für das implizite Euler Verfahren ergibt sich die nachstehende Gleichung, wobei hier y_{j+1} mit dem Newton oder einem anderen Verfahren zur Nullstellenberechnung approximiert werden muß.

$$y(0) = 1 \tag{5}$$
$$y_{j+1} = y_j + h \cdot (10 - 500 \cdot y_{j+1} + 5000 \cdot x_{j+1}) \tag{6}$$

1.3.3 Runge Kutta 2. Ordnung

Zur Vereinfachung der Iterationsgleichung im Runge Kutta 2 Verfahren gelte (7)

$$f(x) = 10 - 500 \cdot y + 5000 \cdot x \tag{7}$$

Dann ergibt sich durch einsetzen in die Runge Kutta Gleichung.

$$y(0) = 1 \tag{8}$$
$$y_{j+1} = y_j + \frac{h}{2} \cdot (f(x_{j+1}, y_j) + f(x_{j+1}, h \cdot f(x_j, y_j))) \tag{9}$$

1.4 Matlab Programme

Listing 1: Stiff

```
function [] = stiff(h)

    vec_ana_x = 0:h:0.2;
    vec_ana_y = mtp0101_ana(vec_ana_x);
    [vec_eulerexpl_x, vec_eulerexpl_y] = eulerE(@f, 0.2, h, [1]);
    [vec_eulerimpl_x, vec_eulerimpl_y] = euler_impl(1, h, 0.2, @f);
```

```
    [vec_runge_x, vec_runge_y] = rungeKutta(1, h, 0.2, @f);


    fh = figure('color','w');
    screen_size = get(0, 'ScreenSize');
    set(fh, 'Position', [0 0 screen_size(3) screen_size(4) ] );
    a = subplot(1,2,1);
    %set(gca,'fontName','Humor Sans111','fontSize',14,'lineWidth',3,'box','off')

    lw = 1;


    %annotation(fh,'textarrow',[0.38 0.34],[0.42 0.55],...
    %     'string',sprintf('analytische Loesung',char(10)),'headStyle','none','lineWidth',1.5,...
    %     'fontName','Comic Sans MS','fontSize',14,'verticalAlignment','middle','horizontalAlignment','left')


    hold on;
    plot(vec_ana_x, vec_ana_y, 'r','lineWidth',lw);
    plot(vec_eulerexpl_x, vec_eulerexpl_y, 'k','lineWidth',lw);
    plot(vec_eulerimpl_x, vec_eulerimpl_y, 'g','lineWidth',lw);
    plot(vec_runge_x, vec_runge_y, 'b','lineWidth',lw);
    grid off;
    title('Approximation');
    legend('analytische Loesung', 'Expl Euler', 'Impl Euler', 'Runge-Kutta');
    axis([0, 0.04, -1.5, 1.5]);

    %xkcdify(a)



    subplot(1,2,2)
    %
    hold on;
    plot(vec_eulerexpl_x, vec_eulerexpl_y.' - vec_ana_y, 'k');
    plot(vec_eulerimpl_x, vec_eulerimpl_y - vec_ana_y, 'g');
    plot(vec_runge_x, vec_runge_y - vec_ana_y, 'b');
    grid on;
    title('Error');
    legend( 'Expl Euler','Impl Euler', 'Runge-Kutta');
    axis([0,0.2, -2.5, 2.5]);

end
```

Listing 2: Steife Differentialgleichung

```
function [ z ] = f( x,y )
    z = 10-500*y+5000*x;
end
```

1.5 Ergebnisse

Im folgenden sind die Approximationen durch alle 3 Verfahren mit Schrittweiten von (0.001 bis 0.005) grapthisch dargestellt. Deutlich erkennbar wird hier wie die expliziten Verfahren (Expliziter Euler, Runge Kutta 2. Ordnung) gegenüber dem impliziten

Euler Verfahren bei wachsender Schrittweite an Genauigkeit verlieren, wie dies auch zu erwarten war.

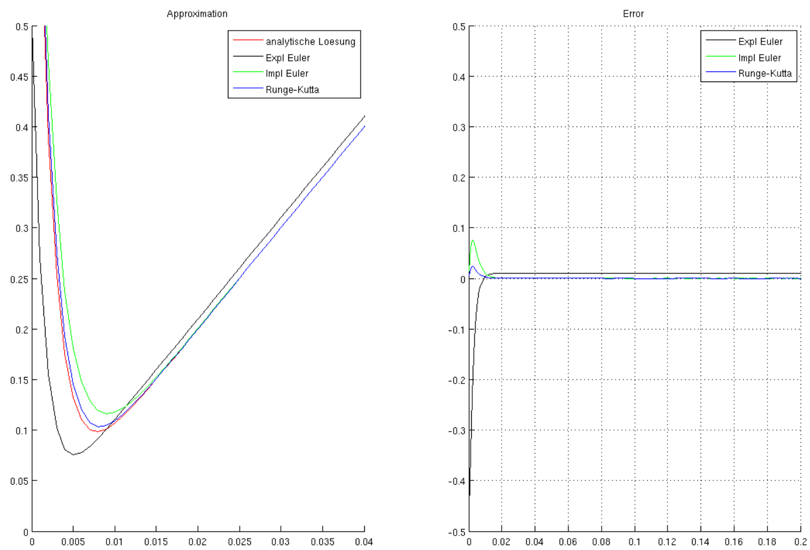

Abbildung 2: Stiff (h=0.001)

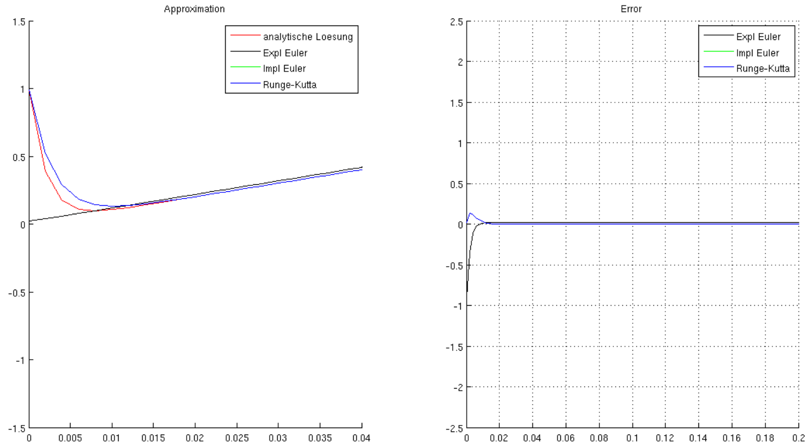

Abbildung 3: Stiff (h=0.002)

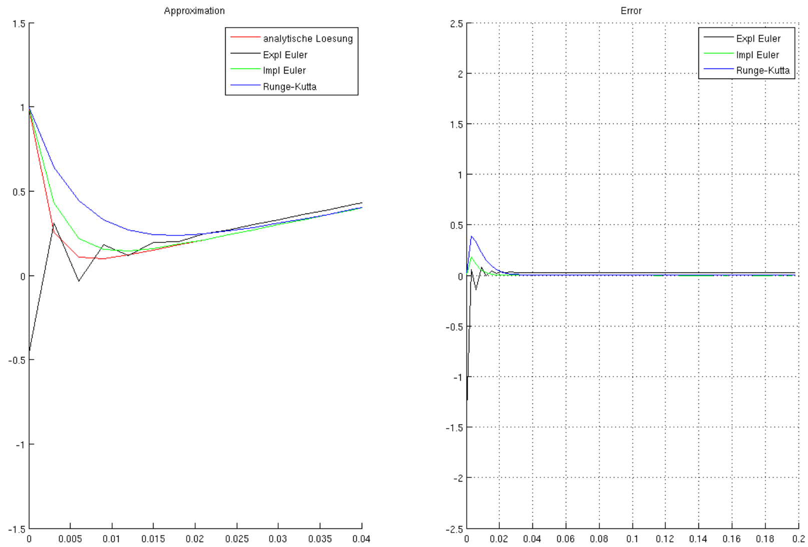

Abbildung 4: Stiff (h=0.003)

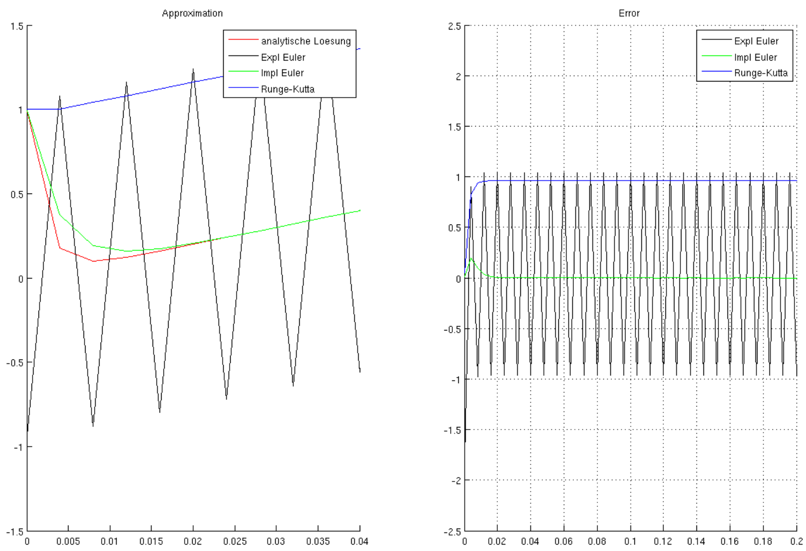

Abbildung 5: Stiff (h=0.004)

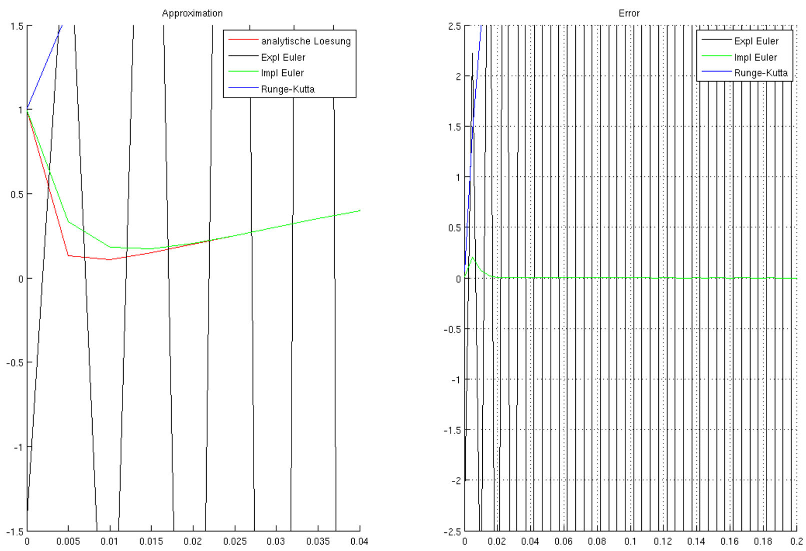

Abbildung 6: Stiff (h=0.005)

2 Van der Pol DGL

2.1 Gleichung

$$y(0) = 0 \tag{10}$$
$$\dot{y}(0) = 1 \tag{11}$$
$$\ddot{y} = 6 \cdot (1 - y^2) \cdot \dot{y} - y \tag{12}$$

2.2 Gleichung als DGL 1. Ordnung

$$\dot{z} = 6 \cdot (1 - y^2) \cdot z - y \tag{13}$$
$$\dot{y} = z \tag{14}$$

2.3 Simulink

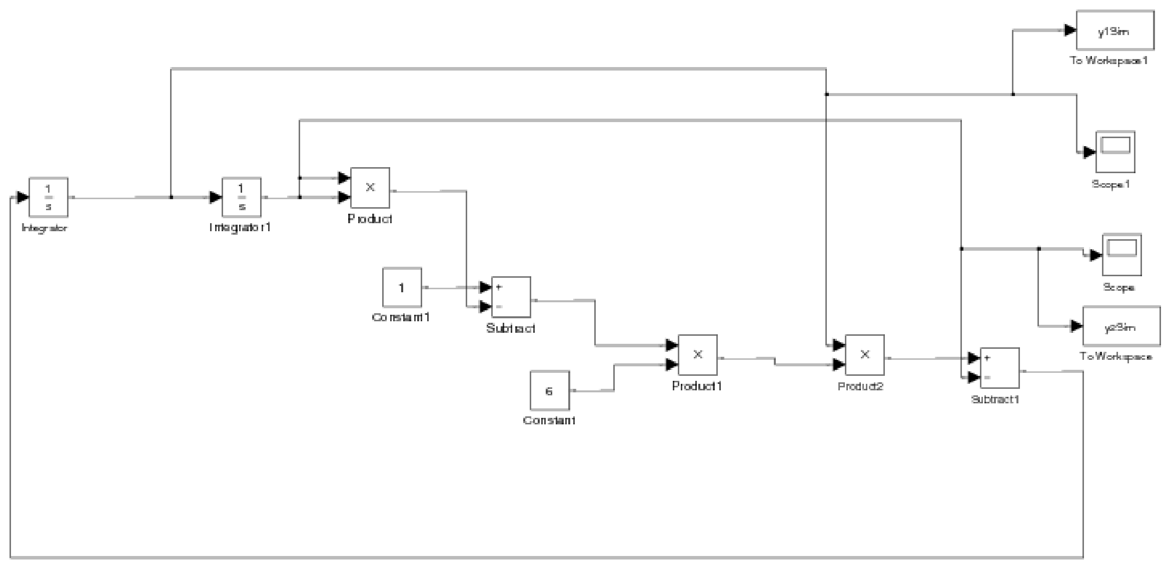

Abbildung 7: Van Der Pol Simulink

2.4 Iterationsgleichungen

2.4.1 Euler Verfahren

$$z_{1_{n+1}} = z_{1_n} + h \cdot (6 \cdot (1 - z_{2_n}^2) \cdot z_{1_n} - z_{2_n}) \tag{15}$$
$$z_{2_{n+1}} = z_{2_n} + h * z_{1_n} \tag{16}$$

2.4.2 Runge Kutta 2. Ordnung

Es gelte

$$g(t, y) = z \tag{17}$$
$$f(y, z) = 6 \cdot (1 - y^2) \cdot z - y \tag{18}$$

Dann können wir durch einsetzen von (17) und (18) in Runge Kutta 2. Ordnung die Iterationsgleichungen erstellen:

$$y_{j+1} = y_j + \frac{h}{2} \cdot [g(t_j, y_j) + g(t_{j+1}, y_i h \cdot g(t_j, y_j))] \tag{19}$$
$$z_{j+1} = z_j + \frac{h}{2} \cdot [f(y_j, z_j) + f(y_{j+1}, z_j + h \cdot f(y_j, z_j))] \tag{20}$$

2.5 Ergebnisse

2.5.1 h=0.001

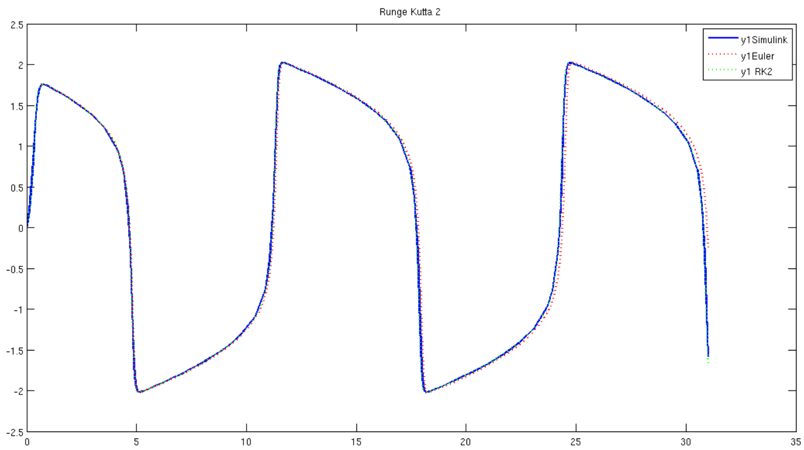

Abbildung 8: Van Der Pol DGL Y1 h=0.001

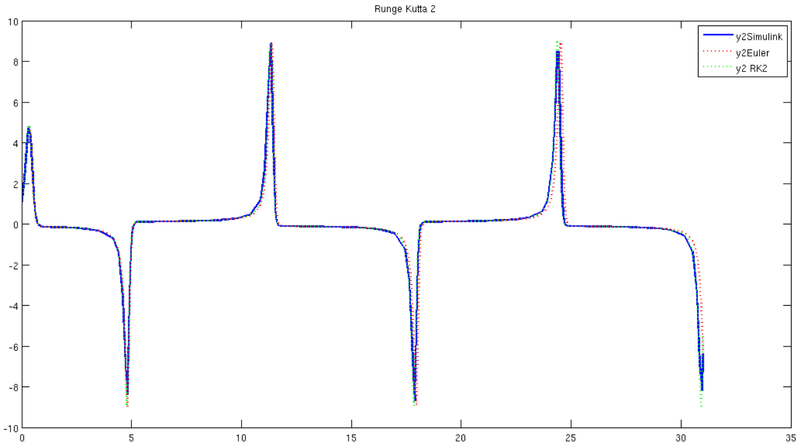

Abbildung 9: Van Der Pol DGL Y2 h=0.001

Bei einer Schrittweite h von 0.001 ist zu erkennen das beide Approximationsverfahren (Expliziter Euler und Runge Kutta 2. Ordnung) mit der aus Simulink extrahierten Approximation (Dormand-Prince, Variable Step Size) übereinstimmen.

2.5.2 h=0.02

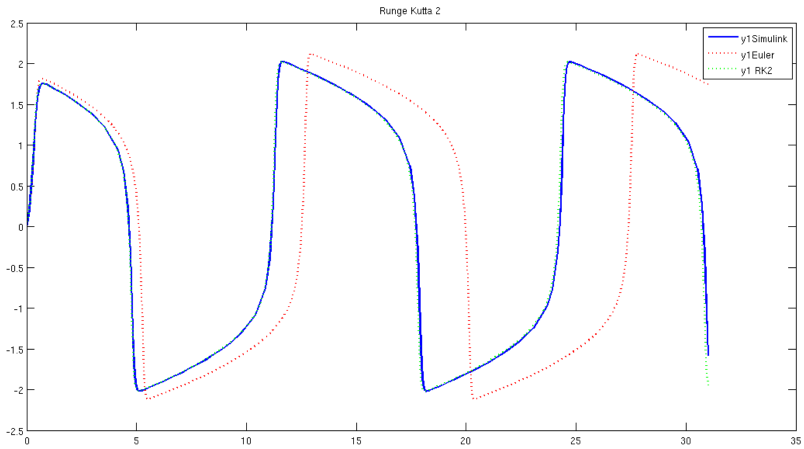

Abbildung 10: Van Der Pol DGL Y1 h=0.02

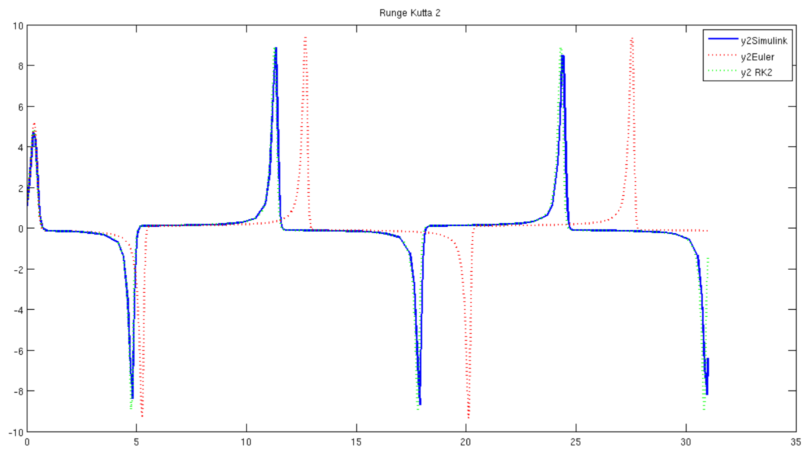

Abbildung 11: Van Der Pol DGL Y2 h=0.02

Bei einer Schrittweite h von 0.02 ist zu erkennen das das simplere Approximationsverfahren (Expliziter Euler) deutlich von der aus Simulink extrahierten Approximation (Dormand-Prince, Variable Step Size) abweicht während das komplexere Verfahren (Runge Kutta 2. Ordnung) auch hier noch sehr dicht an Simulink liegt.

2.6 Matlab Programme

Listing 3: VanDerPol GDL

```
function [ res ] = vdp( x, y )
% Van-Der-Pol-Gleichung zu Aufgabe 2

res = [y(2); 6 * ( 1 - y(1)^2 ) * y(2) - y(1)];

end
```

Listing 4: VanDerPol

```
function [ ] = vanDerPollZeigen( y1Sim, y2Sim, h )
    sum(Prak1Aufg2Simulink)
    [~, Y_E] = eulerE(@vdp, 31, h, [0;1]);
    [X, Y_RK] = rk2(@vdp, 31, h, [0;1]);


   %hold on;
    plot(121);
    p1=plot(y2Sim.time, y2Sim.signals.values ,'b-', X,Y_E(:,1),'r:', X,Y_RK(:,1), 'g:');
    set(p1,'LineWidth',2);
    title('Runge Kutta 2');
    legend('y1Simulink', 'y1Euler', 'y1 RK2');

    figure
    plot(122);
    p2=plot(y1Sim.time, y1Sim.signals.values ,'b-', X,Y_E(:,2),'r:', X,Y_RK(:,2), 'g:');
    set(p2,'LineWidth',2);
    title('Runge Kutta 2');
    legend('y2Simulink', 'y2Euler', 'y2 RK2');

end
```

3 Lorenz Attraktor

3.1 Gleichung

$$\dot{x} = -10 \cdot (x - y) \tag{21}$$
$$\dot{y} = (40 - z) \cdot x - y \tag{22}$$
$$\dot{z} = x \cdot y - 2.67 \cdot z \tag{23}$$
$$x(0) = 0.01 \tag{24}$$
$$y(0) = 0.01 \tag{25}$$
$$z(0) = 0.0 \tag{26}$$

3.2 Simulink

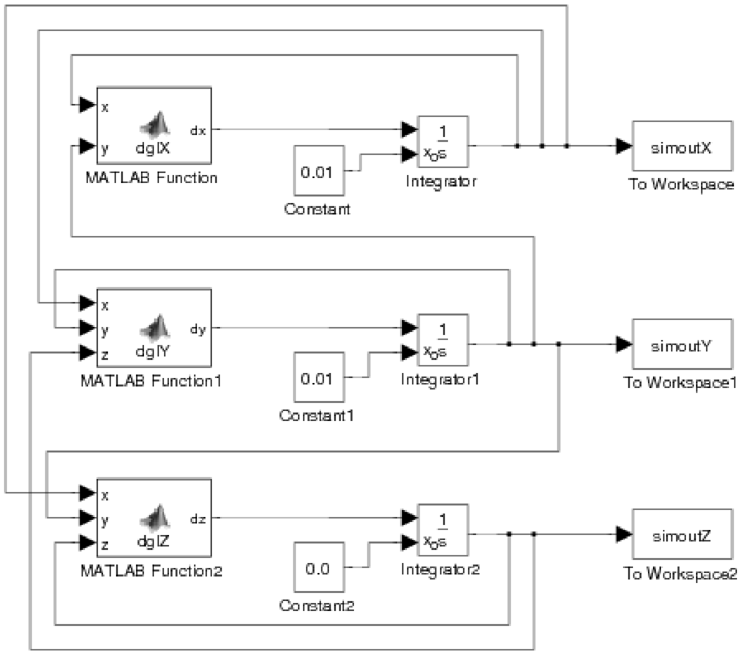

Abbildung 12: Lorenz Attraktor Simulink

3.3 RK2

Gegeben

$$f(t,x) = -10 \cdot (x - y) \tag{27}$$
$$g(t,y) = (40 - z) \cdot x - y \tag{28}$$
$$k(t,z) = x \cdot y - 2.67 \cdot z \tag{29}$$

So erhalten wir durch einsetzen in das Runge Kutta Verfahren 2. Ordnung folgende Iterationsgleichungen:

$$x_{j+1} = x_j + \frac{h}{2} \cdot (f(t_{j+1}, x_j) + f(t_{j+1}, h \cdot f(t_j, x_j))) \tag{30}$$

$$y_{j+1} = y_j + \frac{h}{2} \cdot (g(t_{j+1}, y_j) + g(t_{j+1}, h \cdot g(t_j, y_j))) \tag{31}$$

$$z_{j+1} = z_j + \frac{h}{2} \cdot (k(t_{j+1}, z_j) + k(t_{j+1}, h \cdot k(t_j, z_j))) \tag{32}$$

3.4 Ergebnisse

Mit $h = 0.002$, $t_{End} = 120$ und dem Parameter in der zweiten Gleichung auf 40 ergeben sich folgende Funktionsplots für $x(t), z(t)$

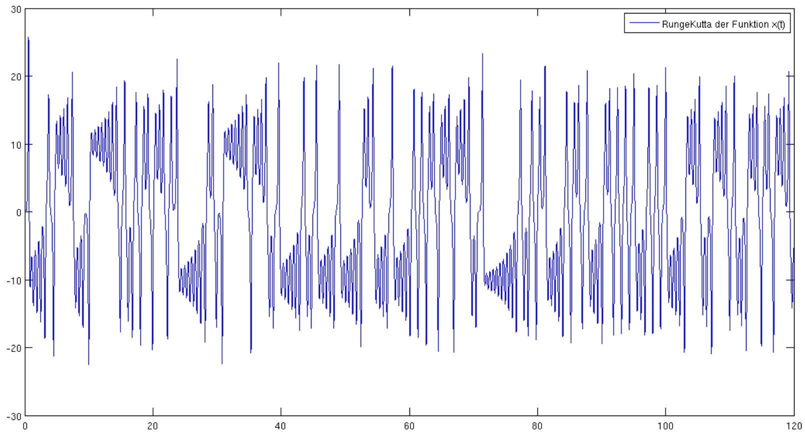

Abbildung 13: Lorenz Attraktor x(t)

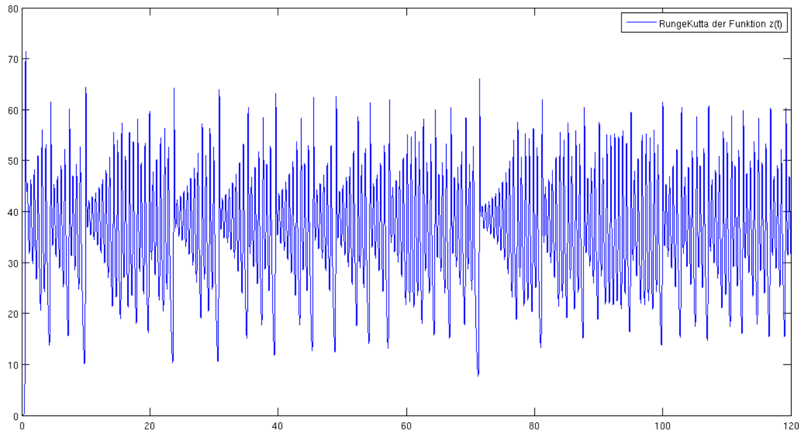

Abbildung 14: Lorenz Attraktor z(t)

Wenn der konstante Parameter der zweiten Funktion auf 40.000000001 geändert wird ergibt sich für $x(t)$ die folgende Veränderung. Es wird eine Verschiebung der Funktion erkennbar.

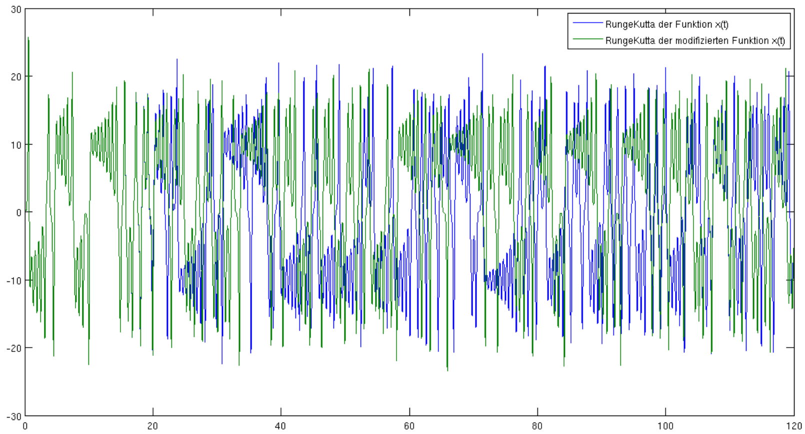

Abbildung 15: Lorenz Attraktor Diff 40, 40.000000001

Die Auswertung der durch Simulink erzeugten Daten ergibt folgendes 3D Muster:

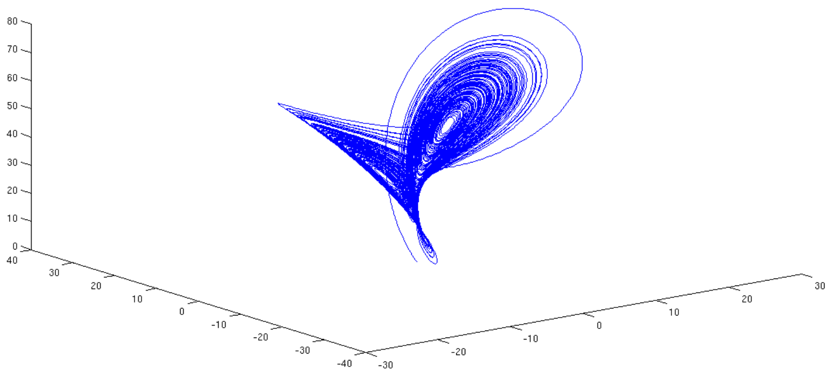

Abbildung 16: Lorenz Attraktor 3D Plot

3.5 Matlab Programme

Listing 5: Lorenz Attraktor

```
function [ res ] = dgls( t, z )

dx = -10 * (z(1) - z(2));
dy = (40 - z(3)) * z(1) - z(2);
dz = z(1) * z(2) -2.67 * z(3);

res = [dx, dy, dz];

end
```

Listing 6: Lorenz Attraktor mit verändertem Parameter

```
function [ res ] = dglsUngenau( t, z )

dx = -10 * (z(1) - z(2));
dy = (40.000000001 - z(3)) * z(1) - z(2);
dz = z(1) * z(2) -2.67 * z(3);

res = [dx, dy, dz];

end
```

Listing 7: Lorenz

```
function [ ] = LorenzAttraktor( h, x_end )
%   Lorenz-Attraktor mit Runge-Kutta-Verfahren 2. Ordnung, Aufgabe 3
%   Parameter:
%       h: Schrittweite
%       x_end: Endzeitpunkt der Simulation

% Lorenz-Attraktor mit dem Runge-Kutta-Verfahren berechnen.
[X, RK_Y] = rk2(@dgls, x_end, h, [0.01,0.01,0.0]);

% Funkton x plotten.
figure(1);
plot(X,RK_Y(:,1));
legend('RungeKutta der Funktion x(t)');

% Funkton z plotten.
figure(2);
plot(X,RK_Y(:,3));
legend('RungeKutta der Funktion z(t)');

% Modifizierten Lorenz-Attraktor mit dem Runge-Kutta-Verfahren berechnen.
[X, RK_YM] = rk2(@dglsUngenau, x_end, h, [0.01,0.01,0.0]);

% Funktion x und modifizierte Funktion x plotten.
figure(3);
plot(X,RK_Y(:,1),X,RK_YM(:,1));
legend('RungeKutta der Funktion x(t)', 'RungeKutta der modifizierten ↩
    Funktion x(t)');

end
```

4 Matlab Programme

Listing 8: Explizites Euler Verfahren

```
function [ X, Y ] = eulerE( func, t, h, init )
%Generisches explizites Euler verfhren
%   Parameter:
%       func: Zeiger auf die DGL
%       t: Endzeitpunkt der Simulation
%       h: Schrittweite
%       init: Anfangswert der Funktion

Y = zeros(t/h,length(init));
X = zeros(t/h,1);

y = init;
index = 1;

X(index) = 0;
Y(index,:) = y;

for k=0:h:t
    y = y + h * func(k, y);

    X(index) = k;
    Y(index,:) = y;
    index = index + 1;
end
```

```
end
```

Listing 9: Implizites Euler Verfahren

```
function [x, y] = euler_impl(begin, h, xend, func)
    yn=begin;
    xn=0;

    vec_x_tmp=[xn];
    vec_y_tmp=[yn];
    xn = xn + h;

    for xn1 = xn:h:xend
        options = optimset('Display', 'off');
        yn1Approx = fsolve(@(n)n-yn-h*func(xn1,n), xn1, options);
        yn1 = yn + h * func(xn1, yn1Approx);

        vec_y_tmp=[vec_y_tmp,yn1];
        vec_x_tmp=[vec_x_tmp,xn1];

        xn=xn1;
        yn=yn1;
    end
    y = vec_y_tmp;
    x = vec_x_tmp;
end
```

Listing 10: Runge Kutta 2. Ordnung

```
function [ X, Y ] = rk2( func, t, h, init )
%Generisches erstes Runge-Kutta-Verfahren 2. Ordnung
%   Parameter:
%       func: Zeiger auf die DGL
%       t: Endzeitpunkt der Simulation
%       h: Schrittweite
%       init: Anfangswert der Funktion

Y = zeros(t/h,length(init));
X = zeros(t/h,1);

y = init;
index = 1;

X(index) = 0;
Y(index,:) = y;

for k=0:h:t
    f = y + h * func(k, y);
    y = y + (h/2) * (func(k, y) + func(k+h, f));

    X(index) = k;
    Y(index,:) = y;
    index = index + 1;
end
end
```