Assignment

Exercise sheet 8

Name TN 1: Tim Peinkofer

tim.peinkofer@student.uni-tuebingen.de

Name TN 2: Fabian Kostow

fabian.kostow@student.uni-tuebingen.de

Tutor: Jose Carlos Olvera Meneses

Date: January 6, 2025

Contents

1 Problem I

1.1 Task 2

We want to show that the 4th order Runge-Kutta-method applied to an ODE of the form $y' = Ay$ will be:

$$y_{n+1} = \left(1 + hA + \frac{1}{2}h^2A^2 + \frac{1}{6}h^3A^3 + \frac{1}{24}h^4A^4\right) \cdot y_n$$

We know that the 4th Runge-Kutta-method is given by:

$$\begin{aligned}
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5) \\
k_1 &= hf(x_n, y_n) \\
k_2 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1\right) \\
k_3 &= hf\left(x_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2\right) \\
k_4 &= hf(x_n + h, y_n + k_3)
\end{aligned}$$

Let us calculate the differnt k for $f(x, y) = f(y) = Ay$:

$$\begin{aligned}
k_1 &= hf(y_n) \\
&= hAy_n
\end{aligned}$$

$$\begin{aligned}
k_2 &= hf(y_n + 0.5k_1) \\
&= hAy_n + \frac{h^2}{2}A^2y_n
\end{aligned}$$

$$\begin{aligned}
k_3 &= hf(y_n + 0.5k_2) \\
&= hA \cdot (y_n + \frac{h}{2}Ay_n + \frac{h^2}{4}A^2y_n) \\
&= hAy_n + \frac{h^2}{2}A^2y_n + \frac{h^3}{4}A^3y_n
\end{aligned}$$

$$
\begin{aligned}
k_4 &= hf(y_n + k_3) \\
&= hA \cdot (y_n + hAy_n + \frac{h^2}{2}A^2y_n + \frac{h^3}{4}A^3y_n) \\
&= hAy_n + h^2A^2y_n + \frac{h^3}{2}A^3y_n + \frac{h^4}{4}A^4y_n
\end{aligned}
$$

If we now combine them based on the formula above we get:

$$
\begin{aligned}
y_{n+1} &= y_n + hAy_n + \frac{1}{2}h^2A^2y_n + \frac{1}{6}h^3A^3y_n + \frac{1}{24}h^4A^4y_n \\
&= \left(1 + hA + \frac{1}{2}h^2A^2 + \frac{1}{6}h^3A^3 + \frac{1}{24}h^4A^4\right) \cdot y_n
\end{aligned}
$$

What should be shown.

1.2 4th-order Adams with cubic approximation

We approximate

$$f(s) \approx f_n + s\Delta f_{n-1} + \frac{s(s+1)}{2}\Delta^2 f_{n-2} + \frac{s(s+1)(s+2)}{6}\Delta^3 f_{n-3}$$

with

$$\begin{aligned}\Delta f_{n-1} &= f_n - f_{n-1}\\ \Delta^2 f_{n-2} &= f_n - 2f_{n-1} + f_{n-2}\\ \Delta^3 f_{n-3} &= f_n - 3f_{n-1} + 3f_{n-2} - f_{n-3}\end{aligned}$$

$$\begin{aligned}y_{n+1} - y_n &= \int_{x_n}^{x_n+1} dy\\ &= h\int_0^1 f_n + s\Delta f_{n-1} + \frac{s(s+1)}{2}\Delta^2 f_{n-2} + \frac{s(s+1)(s+2)}{6}\Delta^3 f_{n-3} ds\\ &= h(f_n + \frac{1}{2}\Delta f_{n-1} + \frac{5}{12}\Delta^2 f_{n-2} + \frac{3}{8}\Delta^3 f_{n-3})\\ &= h(f_n + \frac{1}{2}(f_n - f_{n-1}) + \frac{5}{12}(f_n - 2f_{n-1} + f_{n-2}) + \frac{3}{8}(f_n - 3f_{n-1} + 3f_{n-2} - f_{n-3}))\end{aligned}$$

by computing the terms we get

$$y_{n+1} - y_n = \frac{h}{24}(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3})$$

and our final formula given by

$$y_{n+1} = y_n + \frac{h}{24}(55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3})$$

2 1st-Order ODEs

2.1 Codes

2.1.1 Runge-Kutta-method

```
import numpy as np
import matplotlib.pyplot as plt

# Definition of the function
def f_v(x, y):
    return y*np.cos(x+y)

# 4th Runge-Kutta-method
def runge_kutta(f, y_0, x_start, x_end, n):
    h = (x_end - x_start) / n  # Stepsize
    x_values = np.linspace(x_start, x_end, n + 1)
    y_values = [y_0]  # Solution values
    y = y_0

    for i in range(n):
        x_i = x_values[i]
        k_1 = h * f(x_i, y)
        k_2 = h * f(x_i + 0.5 * h, y + 0.5 * k_1)
        k_3 = h * f(x_i + 0.5 * h, y + 0.5 * k_2)
        k_4 = h * f(x_i + h, y + k_3)

        y = y + (1 / 6) * (k_1 + 2 * k_2 + 2 * k_3 + k_4)
        y_values.append(y)

    return x_values, np.array(y_values)

# Values for calculation
n = 1000
x_start = 0
x_end = 30
y_0 = 1


x, res = runge_kutta(f_v, y_0, x_start, x_end, n)
```

```
# Plot
plt.figure(figsize=(16, 9))
plt.plot(x, res, label="Numerical Solution")
plt.title("Numerical Solution of the ODE $y' = y \\cos(x + y)
                                $")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.legend()
plt.show()
```

2.1.2 Adams-method

```
import numpy as np
import matplotlib.pyplot as plt

def Fprime(x, y):
    yprime = np.sin(x*y)*np.cos(x+y)
    #yprime = y * np.cos(x + y)
    return yprime

def Adam(x0,xm,y0,n,f):
  h=(xm-x0)/n
  x=np.arange(x0,xm+h,h)
  y=np.zeros(len(x))

  y[0] = y0
  y[1] = y[0] + h * Fprime(x[0], y[0])  #first two values via
                                         heun
  y[2] = y[1] + h * Fprime(x[1], y[1])
  for i in range(3, len(x)):      #Adam
        k = y[i-1] + (h / 12) * ( 5 * f(x[i-1], y[i-1]) + 8 *
                                          f(x[i-2], y[i-2]) - f
                                         (x[i-3], y[i-3]))
        y[i]=k
  return x,y

x0=0
xm=30
n=1000
y0=1

x,f=Adam(x0,xm,y0,n,Fprime)

plt.figure(figsize=(16, 9))
plt.plot(x, f)
plt.title("Numerical solution Adam")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()
```

2.1.3 Heun-method

```
import numpy as np
import matplotlib.pyplot as plt


def Fprime(x, y):
    yprime = np.sin(x*y)*np.cos(x+y)
    #yprime = y * np.cos(x + y)
    return yprime

def Heun(x0,xm,y0,n,f):
  h=(xm-x0)/n
  x=np.arange(x0,xm+h,h)
  y=np.zeros(len(x))

  y[0] = y0
  for i in range(1,len(x)):
        ypred = y[i-1] + h * f(x[i-1], y[i-1])
        ycorr = y[i-1] + h/2 * (f(x[i-1], y[i-1]) + f(x[i],
                                          ypred))
        y[i]=ycorr
  return x,y

x0=0
xm=30
n=1000
y0=1

x,f=Heun(x0,xm,y0,n,Fprime)

plt.figure(figsize=(16, 9))
plt.plot(x, f)
plt.title("Numerical solution Heun")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()
```

2.2 $y' = y\,cos(x + y)$

We will now use the Runge-Kutta, Heun and Adams-method to calculate the solution of the following ODE:

$$y' = y\,cos(x + y)$$

For the equation we get the following result with all methods:

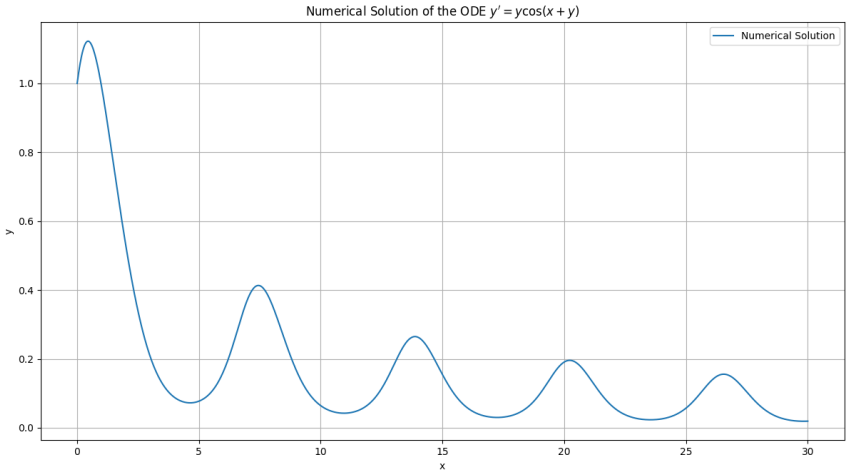

Figure 1: Solution of the ODE

2.3 $y' = sin(xy)cos(x+y)$

And for the second ODE

$$y\prime = \sin(xy) \cdot \cos(x+y)$$

we get for all methods the following solution

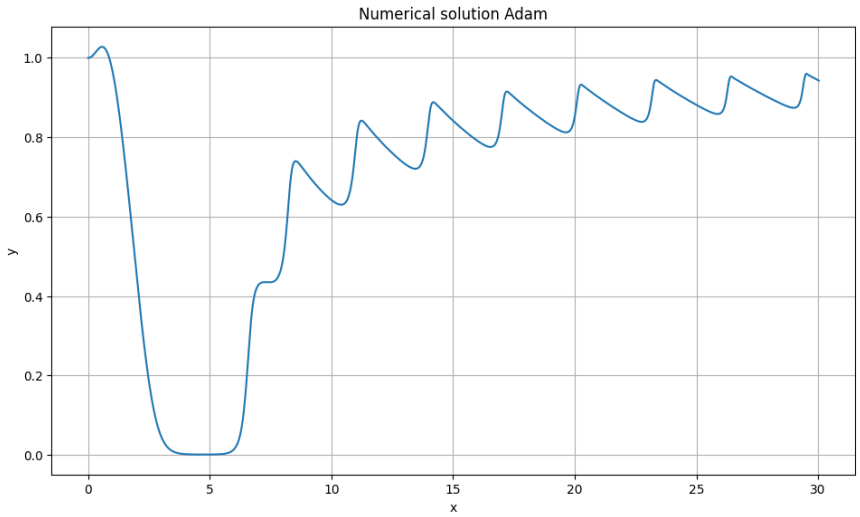

Figure 2: Solution of the ODE

3 Problem III

We now want to take a look at nonlinear first order ODE of the following type:

$$\begin{aligned} y'(x) &= b(x)y^{\alpha}(x) + a(x)y(x) \\ \alpha &> 0 \end{aligned}$$

We try to calculate the solution in three different ways:

- 4th order Runge-Kutta
- Adams-Multon without corrector method
- Adams-Multon with corrector method

The codes are the following are given below.

3.1 4th-order Runge Kutta

```
import numpy as np
import matplotlib.pyplot as plt

# Definition of the function
def f(x,y,alpha=0):
    b = x**2-2*x+np.sin(x)
    a = 1
    return b*y**alpha+ a*y

# 4th Runge-Kutta-method
def runge_kutta(f, y_0, x_start, x_end, n):
    h = (x_end - x_start) / n  # Stepsize
    x_values = np.linspace(x_start, x_end, n + 1)
    y_values = [y_0]  # Solution values
    y = y_0

    for i in range(n):
        x_i = x_values[i]
        k_1 = h * f(x_i, y)
        k_2 = h * f(x_i + 0.5 * h, y + 0.5 * k_1)
        k_3 = h * f(x_i + 0.5 * h, y + 0.5 * k_2)
        k_4 = h * f(x_i + h, y + k_3)

        y = y + (1 / 6) * (k_1 + 2 * k_2 + 2 * k_3 + k_4)
        y_values.append(y)

    return x_values, np.array(y_values)

# Values for calculation
n = 1000
x_start = 0
x_end = 2
y_0 = 0.1

x, res = runge_kutta(f, y_0, x_start, x_end, n)

# Plot
plt.figure(figsize=(16, 9))
plt.plot(x, res, label="4th order Runge Kutta method")
plt.title("Numerical Solution of the ODE $y' = y + x^2-2x+sin
                                (x)$")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.legend()
plt.show()
```

3.2 Adams-Multon method

We have combined the last both Adams methods (with and without corrector method) in one code:

```
import numpy as np
import matplotlib.pyplot as plt

#defintion of the function
def f(x,y,alpha=0):
    b = x**2-2*x+np.sin(x)
    a = 1
    return b*y**alpha+ a*y


#Heun's method for the first steps
def heun_method(f, y_0, x_0, h):
    k1 = f(x_0, y_0)
    k2 = f(x_0 + h, y_0 + h * k1)
    return y_0 + (h / 2) * (k1 + k2)

def adam_predictor(f, y_0, x, n, h): #Predictor method
    y_values = [y_0]  # Solution values list

    for i in range(3): #Calculate the first values via Heun
                                    method
        y_values.append(heun_method(f, y_values[i], x[i], h))

    for i in range(3, n - 1): #Calculte the rest via Adams
                                    method
        y = y_values[i]
            + (h / 24) * ( 55 * f(x[i], y_values[i])
            - 59 * f(x[i-1], y_values[i-1])
            +37* f(x[i-2], y_values[i-2]) -9*f(x[i-3],
                                        y_values[i-3]))
        y_values.append(y)

    return np.array(y_values)
```

```
def adams_corrector(f, y_values, x, n, h): #Corrector method

    for i in range(3, n - 1):
        y_corr = y_values[i]
            + (h / 24) * (9 * f(x[i + 1], y_values[i + 1])
            + 19 * f(x[i], y_values[i])
            - 5 * f(x[i - 1], y_values[i - 1])
            + f(x[i - 2], y_values[i - 2]))
        y_values[i + 1] = y_corr

    return np.array(y_values)

def adams_ode_int(f, y_0, x, n, h): #Combination of both
                                    methods
    y_pred = adam_predictor(f, y_0, x, n, h)
    y_corr = adams_corrector(f, y_pred, x, n, h)
    return y_corr, y_pred


x = np.linspace(0, 2, 1000)
y_corr, y_pred = adams_ode_int(f, 0.1, x, 1000, 2/1000)

#Plot
plt.figure(figsize=(16, 9))
plt.plot(x, y_pred, color='blue', label='Adams method without
                                    corrector')
plt.plot(x, y_corr, color='red', linestyle='--', label='Adams
                                    method with corrector')
plt.title("Numerical solution of the ODE $y' = x^2-2x+sin(x)+
                                    y$")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()
```

3.3 Application testing

We now want to test these code for the following testcase:

$$\begin{aligned} y' &= y + x^2 - 2x + \sin(x) \\ y(0) &= 0.1 \end{aligned}$$

From above we can find that:

$$\begin{aligned} a(x) &= 1 \\ b(x) &= x^2 - 2x + \sin(x) \\ \alpha &= 0 \end{aligned}$$

As a result we get the following:

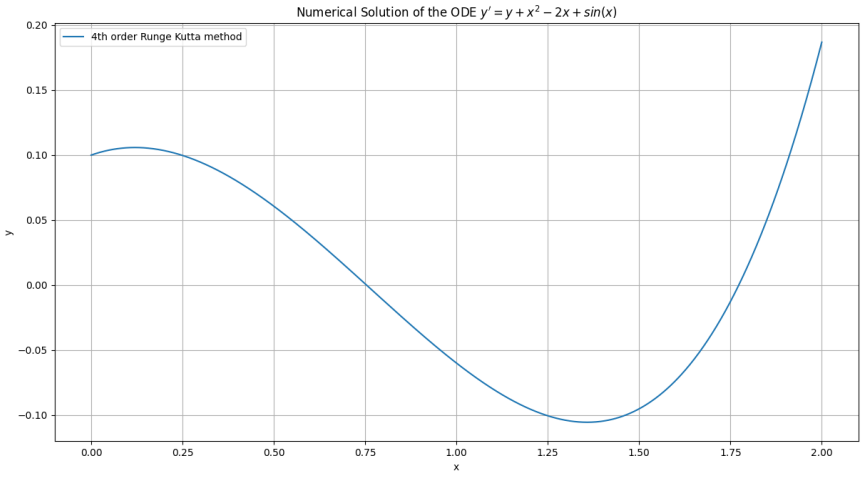

Figure 3: 4th-order Runge-Kutta-method

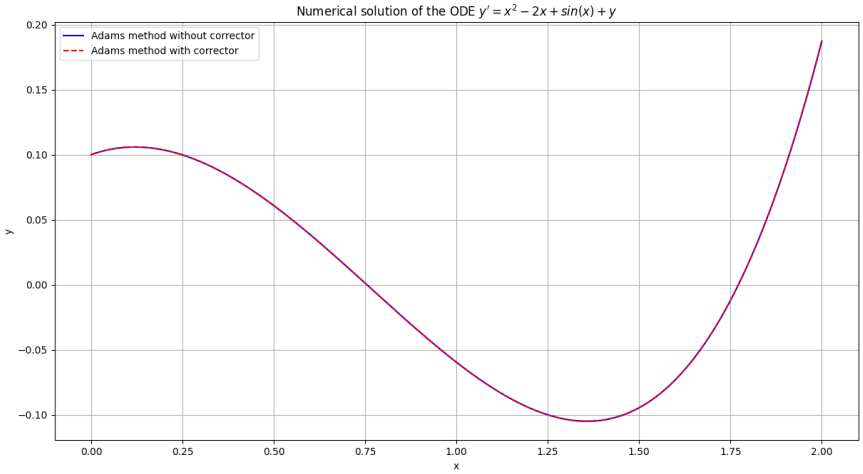

Figure 4: Adams-Multon method

We used the 4th order equations for both cases so that we can compare them. As we see, we get the same result in all three cases.

Let us now take a look at the error and efficiency of our used methods.

3.4 Analytical error and efficiency

We first want to take a look a the error and convergence behavior of our used methods. The code for our error evaluation is the following:

```
import numpy as np
import matplotlib.pyplot as plt
from Adams_multon import adam_predictor, adams_corrector, f
from Runge_Kutta_Task_2 import runge_kutta

# Define N as an array of integers (10 to 5000 steps)
N_values = np.linspace(10, 5000, 100, dtype=int)
errors_1 = []
errors_2 = []
errors_3 = []

for i in range(len(N_values)-1):  # calculate the error for
                                  every N and 2N
    N_v = N_values[i]
    N_2 = N_values[i + 1]

    # Create grids for N and 2N
    x_N = np.linspace(0, 2, N_v)
    x_N_2 = np.linspace(0, 2, N_2)
    h_N = x_N[1] - x_N[0]
    h_N_2 = x_N_2[1] - x_N_2[0]

    # Compute the values via Adams-Moulton method
    y_pred_N = adam_predictor(f, 0.1, x_N, N_v, h_N)
    y_corr_N = adams_corrector(f, y_pred_N, x_N, N_v, h_N)

    y_pred_N_2 = adam_predictor(f, 0.1, x_N_2, N_2, h_N_2)
    y_corr_N_2 = adams_corrector(f, y_pred_N_2, x_N_2, N_2,
                                    h_N_2)

    # Compute the values via Runge-Kutta method
    y_runge_N_2 = runge_kutta(f, 0.1, 0, 2, N_2)[1]
    y_runge_N = runge_kutta(f, 0.1, 0, 2, N_v)[1]

    # Compute the absolute error at x = 2
    error_1 = abs(y_corr_N[-1] - y_corr_N_2[-1])  # Error
                                    between N and 2N for Adams
                                     with corrector
    errors_1.append(error_1)
```

```
    error_2 = abs(y_pred_N[-1] - y_corr_N_2[-1])  # Error
                                    between N and 2N for Adams
                                     without corrector
    errors_2.append(error_2)

    error_3 = abs(y_runge_N[-1] - y_runge_N_2[-1])  # Error
                                    between N and 2N for Runge
                                    -Kutta
    errors_3.append(error_3)

# Plot the absolute error in a log-log plot
plt.figure(figsize=(10, 6))
plt.loglog(N_values[:-1], errors_1, linestyle='-', color='b',
                                  label="Adams with corrector")
plt.loglog(N_values[:-1], errors_3, linestyle='-', color='g',
                                  label="Runge-Kutta")
plt.loglog(N_values[:-1], errors_2, linestyle='--', color='r'
                                , label="Adams without
                                corrector")
plt.title("Absolute Error of Different Methods")
plt.xlabel("Number of steps (N)")
plt.ylabel("Absolute error (E)")
plt.legend()
plt.grid(True)
plt.show()
```

We imported our methods from the last step files.

As a result we get the following error plot:

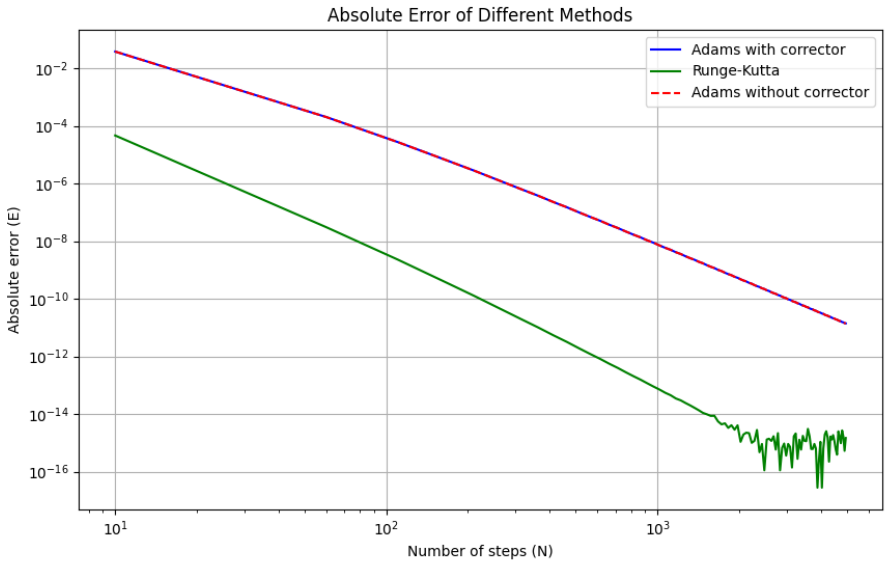

Figure 5: Error-N-log Plot

The error decreases for all three methods. However, it decreases faster with the Runge-Kutta method than with the Adam-Multon method. For the Adam-Multon method, we have the same error decrease for the case with and without corrector. However, for the Runge-Kutta method, the error decreases faster.

It can also be seen that for the Runge-Kutta method, the error no longer follows a straight line at high N. This is due to the fact that for large N, the Runge-Kutta method can diverge or provide incorrect solutions.

Let us now look at the convergence behavior. You can see from the plot that the Adam-Multon method converges to the power of 3, while the Runge-Kutta method converges to the power of 4.

Finally, let's take a look at the number of evaluation compared to the error for every method. We used the following code for our plot:

```
import numpy as np
import matplotlib.pyplot as plt
from Adams_multon import adam_predictor, adams_corrector, f
from Runge_Kutta_Task_2 import runge_kutta


N_values = np.linspace(10, 5000, 100, dtype=int)
errors_1 = []
errors_2 = []
errors_3 = []
evaluations_1 = []  # For Adams with corrector
evaluations_2 = []  # For Adams without corrector
evaluations_3 = []  # For Runge-Kutta

for i in range(len(N_values)-1):  # calculate the error for
                                  every N and 2N
    N_v = N_values[i]
    N_2 = N_values[i + 1]


    x_N = np.linspace(0, 2, N_v)
    x_N_2 = np.linspace(0, 2, N_2)
    h_N = x_N[1] - x_N[0]
    h_N_2 = x_N_2[1] - x_N_2[0]

    # Compute the values via Adams-Moulton method
    y_pred_N = adam_predictor(f, 0.1, x_N, N_v, h_N)
    y_corr_N = adams_corrector(f, y_pred_N, x_N, N_v, h_N)

    y_pred_N_2 = adam_predictor(f, 0.1, x_N_2, N_2, h_N_2)
    y_corr_N_2 = adams_corrector(f, y_pred_N_2, x_N_2, N_2,
                                 h_N_2)

    # Compute the values via Runge-Kutta method
    y_runge_N_2 = runge_kutta(f, 0.1, 0, 2, N_2)[1]  # Only
                                 get the solution values
    y_runge_N = runge_kutta(f, 0.1, 0, 2, N_v)[1]  # Only get
                                  the solution values

    # Compute the absolute error at x = 2
    error_1 = abs(y_corr_N[-1] - y_corr_N_2[-1])  # Error
                                 between N and 2N for Adams
                                  with corrector
```

```
    errors_1.append(error_1)
    evaluations_1.append(2 * N_v)  # 2N evaluations for Adams
                                    with corrector

    error_2 = abs(y_pred_N[-1] - y_corr_N_2[-1])  # Error
                                    between N and 2N for Adams
                                    without corrector
    errors_2.append(error_2)
    evaluations_2.append(N_v)  # N evaluations for Adams
                                    without corrector

    error_3 = abs(y_runge_N[-1] - y_runge_N_2[-1])  # Error
                                    between N and 2N for Runge
                                    -Kutta
    errors_3.append(error_3)
    evaluations_3.append(4 * N_v)  # 4N evaluations for Runge
                                    -Kutta

# Plot the error vs function evaluations
plt.figure(figsize=(10, 6))
plt.loglog(evaluations_1, errors_1, linestyle='-', color='b',
                                label="Adams with corrector (
                                2N)")
plt.loglog(evaluations_2, errors_2, linestyle='-', color='r',
                                label="Adams without
                                corrector (N)")
plt.loglog(evaluations_3, errors_3, linestyle='-', color='g',
                                label="Runge-Kutta (4N)")
plt.title("Error vs Function Evaluations")
plt.xlabel("Number of function evaluations")
plt.ylabel("Absolute error")
plt.legend()
plt.grid(True)
plt.show()
```

As a result we get the following plot:

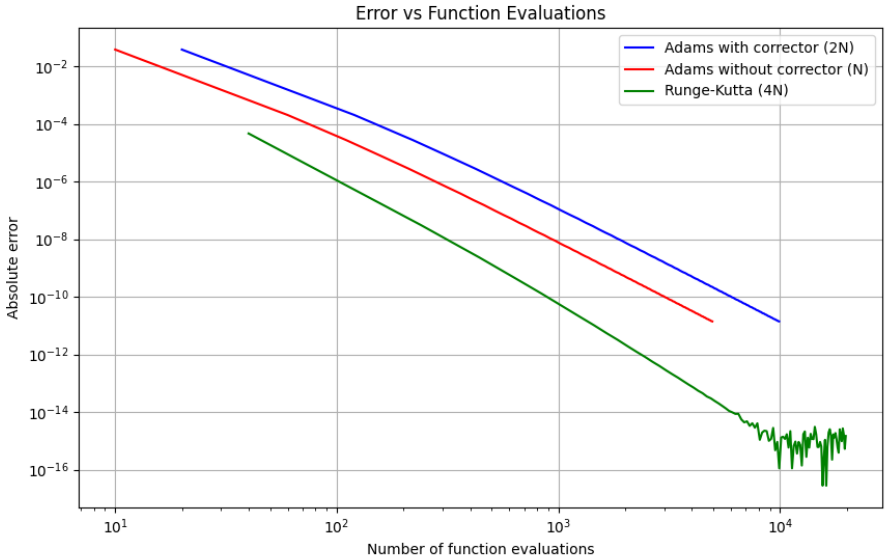

Figure 6: Number of evaluations vs. error

As we can see for Number of evaluation until 10^4 the Runge-Kutta-method is most efficient. Otherwise the Adams with corrector is more efficient for higher number of iterations.

3.5 Application: Nonlinear Air-resistance

We now want to do the same thing for the folllwing equation:

$$v'(t) = -\mu(t)v(t) + \omega(t)^2 v^3(t)$$
$$v(0) = 2$$

Which leads us, based on the function from above, to the following equations:

$$a(x) = \mu(t)$$
$$b(x) = \omega(t)^2$$
$$\alpha = 3$$

We can use the code from the task before to calculate the error:

```
import numpy as np
import matplotlib.pyplot as plt
from Adams_multon import adam_predictor, adams_corrector

def f(v,x):
    return 0.707**2*v**3-2*v

# Define N as an array of integers (10 to 5000 steps)
N_values = np.linspace(10, 5000, 100, dtype=int)
errors_1 = []


for i in range(len(N_values)-1):  # calculate the error for
                                  every N and 2N
    N_v = N_values[i]
    N_2 = N_values[i + 1]

    # Create grids for N and 2N
    x_N = np.linspace(0, 2, N_v)
    x_N_2 = np.linspace(0, 2, N_2)
    h_N = x_N[1] - x_N[0]
    h_N_2 = x_N_2[1] - x_N_2[0]

    # Compute the values via Adams-Moulton method
    y_pred_N = adam_predictor(f, 0.1, x_N, N_v, h_N)
    y_corr_N = adams_corrector(f, y_pred_N, x_N, N_v, h_N)

    y_pred_N_2 = adam_predictor(f, 0.1, x_N_2, N_2, h_N_2)
    y_corr_N_2 = adams_corrector(f, y_pred_N_2, x_N_2, N_2,
                                 h_N_2)
```

```
    # Compute the absolute error at x = 2
    error_1 = abs(y_corr_N[-1] - y_corr_N_2[-1])  # Error
                                    between N and 2N for Adams
                                     with corrector
    errors_1.append(error_1)



# Plot the absolute error in a log-log plot
plt.figure(figsize=(10, 6))
plt.loglog(N_values[:-1], errors_1, linestyle='-', color='b',
                                label="Adams with corrector")
plt.title("Absolute Error of Different Methods")
plt.xlabel("Number of steps (N)")
plt.ylabel("Absolute error (E)")
plt.legend()
plt.grid(True)
plt.show()
```

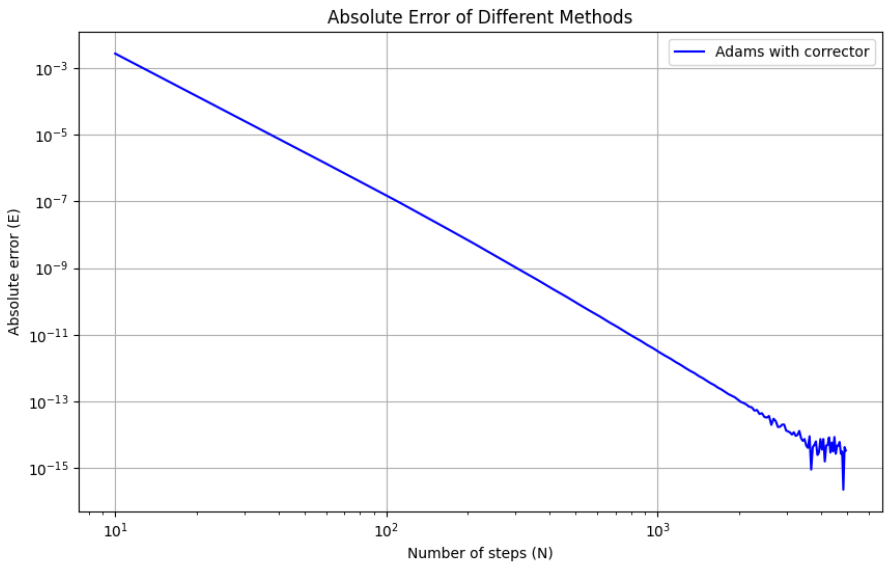

Figure 7: Error behavior

In the next step we want to take a look at the behavior of $v(t)$ for different values of ω_0 in the range of $\frac{1}{\sqrt{2}}$. The code is the following:

```
import numpy as np
import matplotlib.pyplot as plt
b = 0.707
a = 2

# Definition of the function
def f(x, y):
    return -a * y + b**2 * y**3

# Runge-Kutta 4th order
def runge_kutta(f, y_0, x_start, x_end, n):
    h = (x_end - x_start) / n
    x_values = np.linspace(x_start, x_end, n + 1)
    y_values = [y_0]
    y = y_0

    for i in range(n):
        x_i = x_values[i]
        k_1 = h * f(x_i, y)
        k_2 = h * f(x_i + 0.5 * h, y + 0.5 * k_1)
        k_3 = h * f(x_i + 0.5 * h, y + 0.5 * k_2)
        k_4 = h * f(x_i + h, y + k_3)

        y = y + (1 / 6) * (k_1 + 2 * k_2 + 2 * k_3 + k_4)
        y_values.append(y)

    return x_values, np.array(y_values)

# Parameter
n = 1000
x_start = 0
x_end = 10
y_0 = 2
w_0 = 1/np.sqrt(2)
w_0_values = np.linspace(w_0-0.5, w_0, 7)
w_0_values_2 = np.linspace(w_0, w_0+0.5, 7)


plt.figure(figsize=(16, 9))
```

```
#Plot for w_0 < 0.707
for i in w_0_values:
    b = i
    x, res = runge_kutta(f, y_0, x_start, x_end, n)
    plt.plot(x, res, label="b = {}".format(i.round(3)))

plt.title("Numerical Solution der DGL $y' = -ay + b^2y^3$")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.legend()
plt.show()

plt.figure(figsize=(16, 9))

#Plot for w_0 > 0.707
for i in w_0_values_2:
    b = i
    x, res = runge_kutta(f, y_0, x_start, x_end, n)
    plt.plot(x, res, label="b = {}".format(i.round(3)))

plt.title("Numerical Solution der DGL $y' = -ay + b^2y^3$")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.legend()
plt.show()
```

We generated a plot for $\omega_0 < 0.707$ and $\omega_0 > 0.707$ so that we can compare the differnces for both cases.

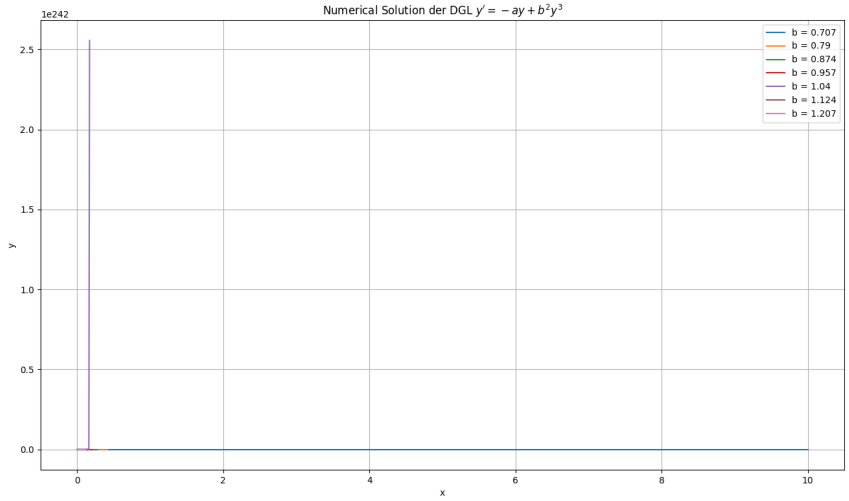

Figure 8: $\omega_0 > 0.707$

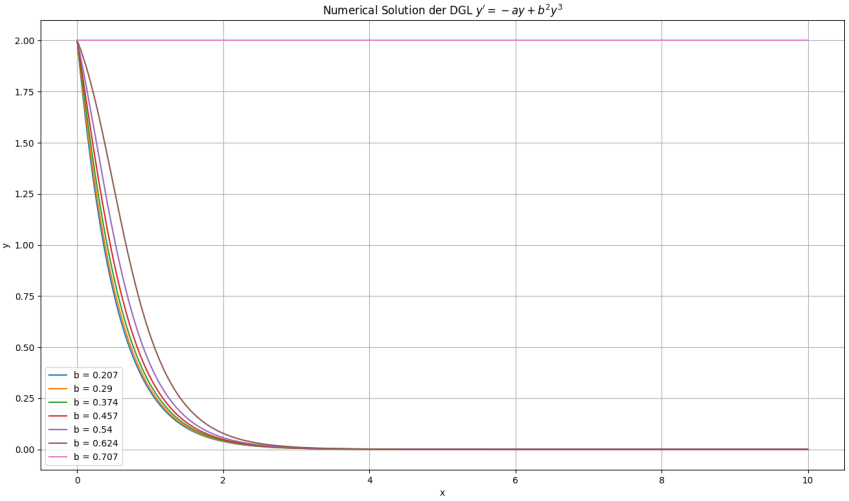

Figure 9: $\omega_0 < 0.707$

For $\omega_0 \leq 0.707$ we obtain an exponentially decreasing curve, which was to be expected.

However, the case $\omega_0 > 0.707$ is more interesting. Here we see values in the range of a and strong jumps. The reason for this is that the speed increases exponentially for this case, which is why our methods cannot calculate good approximate solutions. This results in the plot shown above.

```
import numpy as np
import matplotlib.pyplot as plt

# Definition of the function
def f(x, y, a):
    b = (1 / np.sqrt(2)) * np.tan(x)
    return -a * y + b**2 * y**3

# Runge-Kutta 4th order
def runge_kutta(f, y_0, x_start, x_end, n, a):
    h = (x_end - x_start) / n
    x_values = np.linspace(x_start, x_end, n + 1)
    y_values = [y_0]
    y = y_0

    for i in range(n):
        x_i = x_values[i]
        k_1 = h * f(x_i, y, a)
        k_2 = h * f(x_i + 0.5 * h, y + 0.5 * k_1, a)
        k_3 = h * f(x_i + 0.5 * h, y + 0.5 * k_2, a)
        k_4 = h * f(x_i + h, y + k_3, a)

        y = y + (1 / 6) * (k_1 + 2 * k_2 + 2 * k_3 + k_4)
        y_values.append(y)

    return x_values, np.array(y_values)

# Parameter
n = 1000
x_start = 0
x_end = 10
y_0 = 2
mu_0_values = np.linspace(2, 1.5, 20)

plt.figure(figsize=(16, 9))

# Plot for different mu_0 values
for mu_0 in mu_0_values:
    x, res = runge_kutta(f, y_0, x_start, x_end, n, mu_0)
    plt.plot(x, res, label="mu_0 = {:.3f}".format(mu_0))

plt.title("Numerical solution of the DGL $y' = -ay + b^2y^3$"
                                    )
plt.xlabel("t")
plt.ylabel("y(t)")
plt.grid(True)
plt.legend()
plt.show()
```

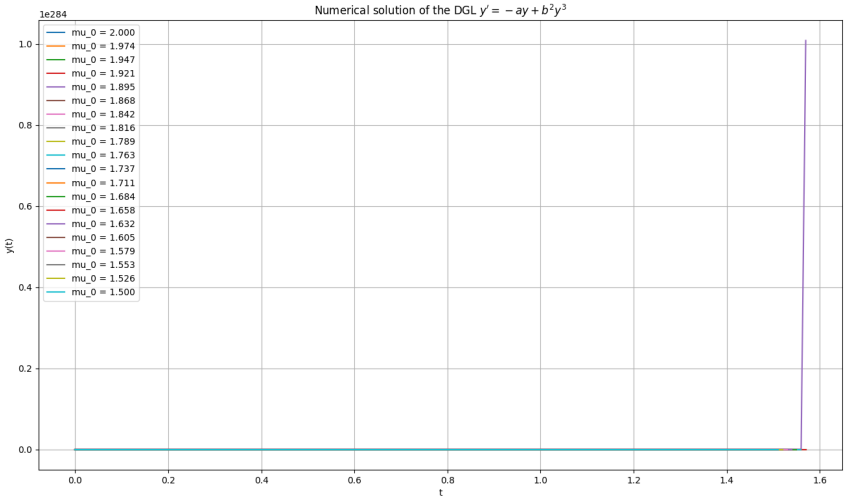

Figure 10: Decreasing beavior of μ_0

Like before we have the same behavior if we decrease μ_0 and use $\omega(t) = \omega_0 \cdot tan(t)$. The term of power three dominates the function and the solution diverges. This leads to the same result as we explained for $\omega_0 > 0.707$.

4 Problem IV

4.1 IV-2

The complete python code is on Github, here we go over the important parts and the solutions of the code.

```
for i in range(1, len(x)):
        #k1 and l1
        k1 = Fy(x[i-1], y[i-1], z[i-1])
        l1 = Fz(x[i-1], y[i-1], z[i-1])
        k1 *= dx
        l1 *= dx

        #k2 and l2
        k2 = Fy(x[i-1] + 0.5 * dx, y[i-1] + 0.5 * k1, z[i-1]
                                        + 0.5 * l1)
        l2 = Fz(x[i-1] + 0.5 * dx, y[i-1] + 0.5 * k1, z[i-1]
                                        + 0.5 * l1)
        k2 *= dx
        l2 *= dx

        #k3 and l3
        k3 = Fy(x[i-1] + 0.5 * dx, y[i-1] + 0.5 * k2, z[i-1]
                                        + 0.5 * l2)
        l3 = Fz(x[i-1] + 0.5 * dx, y[i-1] + 0.5 * k2, z[i-1]
                                        + 0.5 * l2)
        k3 *= dx
        l3 *= dx

        #k4 and l4
        k4 = Fy(x[i-1] + dx, y[i-1] + k3, z[i-1] + l3)
        l4 = Fz(x[i-1] + dx, y[i-1] + k3, z[i-1] + l3)
        k4 *= dx
        l4 *= dx

        # Update y and z usingRunge-Kutta
        y[i] = y[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6
        z[i] = z[i-1] + (l1 + 2*l2 + 2*l3 + l4) / 6

    return x, y, z
```

Here we calculate for each step i the constants k and l, to calculate the coresbonding y and z values.
The functions we need are defined as follows

```
def Fy(x, y, z):
    yprime = np.sin(y) + np.cos(z * x)
    return yprime

def Fz(x, y, z):
    if x!=0:
      zprime = np.exp(-y * x) + np.sin(z * x) / x
      return zprime
    else:
      return 0
```

The following inital conditions are given:

$$\begin{aligned} x &\in [-1, 4] \\ dx &= 0.25 \\ y(-1) &= 2.37 \\ z(-1) &= -3,48 \end{aligned}$$

the following table shows each value y and z for each x

x	y	z
-1.00	2.370000	-3.480000
-0.75	2.439910	-1.646617
-0.50	2.753486	-0.628563
-0.25	3.057687	0.058067
0.00	3.297349	0.449781
0.25	3.483730	0.694100
0.50	3.619657	0.965929
0.75	3.685964	1.245404
1.00	3.643931	1.520443
1.25	3.474004	1.736774
1.50	3.227209	1.850712
1.75	2.990372	1.869686
2.00	2.815888	1.828189
2.25	2.713767	1.756398
2.50	2.673875	1.672686
2.75	2.681007	1.586712
3.00	2.720709	1.503140
3.25	2.780937	1.424003
3.50	2.852314	1.349989
3.75	2.927910	1.281118
4.00	3.002850	1.217095

For the plots we use smaller dx to get smoother curves.

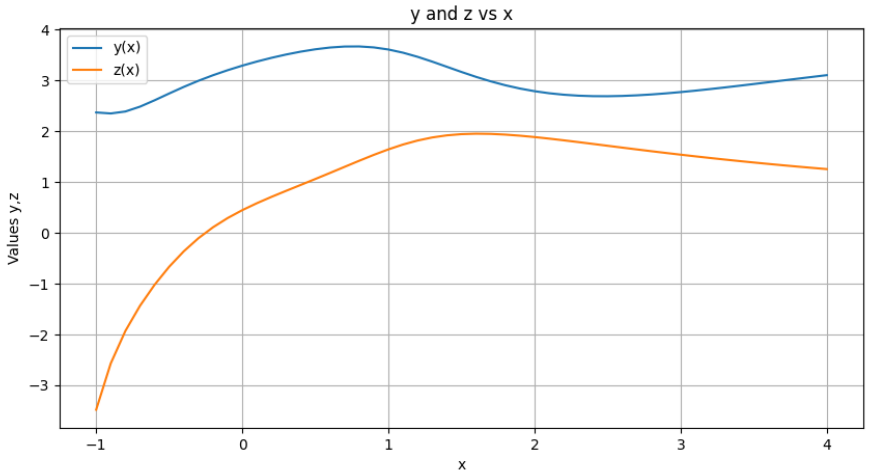

Figure 11: y and z over x

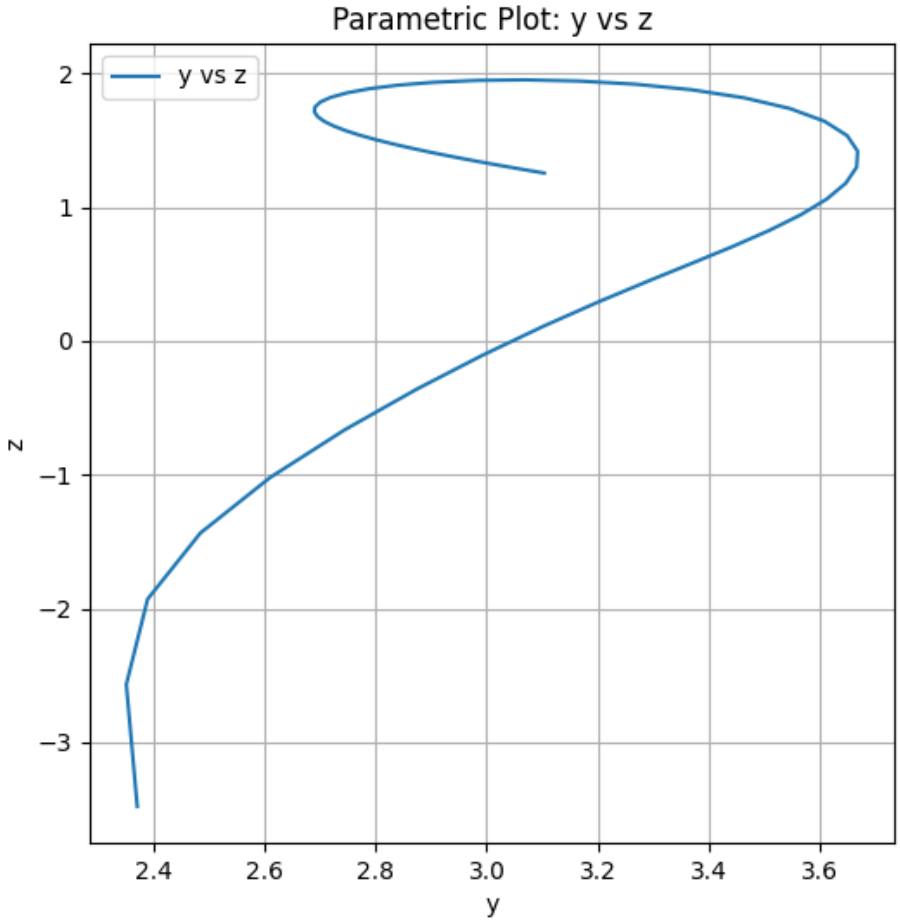

Figure 12: z over y