1) RK-Verfahren mit Ordnung 4 / 5 implementieren

Hierfür wurde die Programmstruktur für explizite RK-Verfahren aus Praktikum SW09 verwendet und als Funktion definiert. Bei Funktionsaufruf müssen der Funktion die Arrays des Butcher-Tableaus (a, b, c) übergeben werden.

```python
# ----------- Definition allgemeines Runge-Kutta Verfahren explizit -----------------
def Runge_Kutta(a, b, c, xend, h, y0, f):
    x = [0.]
    y = [y0]
    s = np.size(b)
    r = np.zeros(s)
    xalt = 0
    yalt = y0

    while x[-1] < xend-h/2:
        # Runge-Kutta Verfahren Schritt
        for i in range(s):
            r[i] = f(xalt + c[i] * h, yalt + h * sum(a[i]*r))
        yneu = yalt + h * sum(b*r)
        xneu = xalt + h

        # Speichern des Resultats
        y.append(yneu)
        x.append(xneu)

        yalt = yneu
        xalt = xneu
        r = np.zeros(s)
    return np.array(x), np.array(y)
```

Um die oben gezeigte Funktion mit den korrekten Butcher-Tableaus für Ordnung 4 und 5 aufzurufen, wurden zwei weitere Funktionen definiert:

```python
# ----------- Definition Runge-Kutta Verfahren explizit mit Butcher Tableau 4. Ordnung -----
def Runge_Kutta_4(xend, h, y0, f):
    # Implementation des Butcher-Tableaus
    a = np.array([[0,       0,        0,       0,     0,    0],
                  [2/9,     0,        0,       0,     0,    0],
                  [1/12,    1/4,      0,       0,     0,    0],
                  [69/128, -243/128, 135/64,  0,     0,    0],
                  [-17/12,  27/4,    -27/5,   16/15, 0,    0],
                  [65/432, -5/16,     13/16,  4/27,  5/144,0]])
    b = np.array([1/9, 0, 9/20, 16/45, 1/12, 0])
    c = np.array([0, 2/9, 1/3, 3/4, 1, 5/6])

    return Runge_Kutta(a, b, c, xend, h, y0, f)
```

```python
# ----------- Definition Runge-Kutta Verfahren explizit mit Butcher Tableau 5. Ordnung -----
def Runge_Kutta_5(xend, h, y0, f):
    # Implementation des Butcher-Tableaus
    a = np.array([[0,       0,        0,       0,     0,    0],
                  [2/9,     0,        0,       0,     0,    0],
                  [1/12,    1/4,      0,       0,     0,    0],
                  [69/128, -243/128, 135/64,  0,     0,    0],
                  [-17/12,  27/4,    -27/5,   16/15, 0,    0],
                  [65/432, -5/16,     13/16,  4/27,  5/144,0]])
    b = np.array([47/450, 0, 12/25, 32/225, 1/30, 6/25])
    c = np.array([0, 2/9, 1/3, 3/4, 1, 5/6])

    return Runge_Kutta(a, b, c, xend, h, y0, f)
```

2) Modellproblem lösen und absoluten Fehler darstellen

Um die in 1) implementierten Funktionen zu testen, wurde ein Modellproblem implementiert. Die analytische Lösung des Problems ya() ist bekannt und wurde ebenfalls implementiert, um anschliessend den absoluten Fehler zu berechnen.

```python
# ----------- Implementation des Modells und dessen analytischer Lösung -------------
def ya(x):      # analytische Lösung
    return - (16-(2/3)*x**3)**0.5   # für alle x grösser oder gleich 2*3**(1/3)

def f(x,y):
    return -(x**2/y)

def df(x,y):
    return x**2/y**2

y0 = -4
```

Für verschiedene Schrittweiten wurde das Modellproblem mit den in 1) implementierten Verfahren gelöst und der Fehler an der Endstelle ausgewertet.

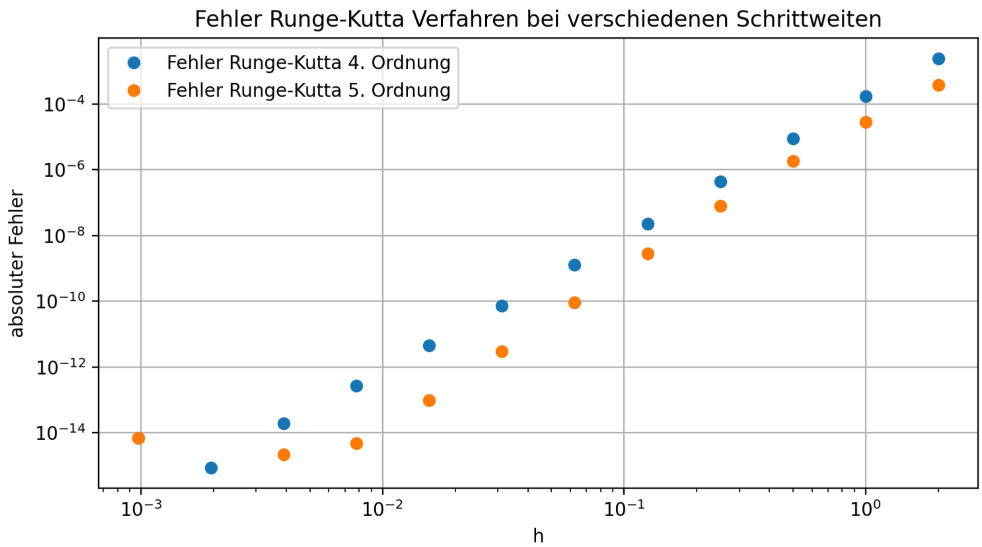
```python
# ------------ Berechnung und Darstellung des absoluten Fehlers ------------------------
def absError(f, ya, y0):
    xend = 2
    h = []
    err_rk4 = []
    err_rk5 = []
    for j in range(0, 12, 1):
        hnew = 2 / (2 ** ((j+1) - 1))

        xrk4, yrk4 = Runge_Kutta_4(xend, hnew, y0, f)
        xrk5, yrk5 = Runge_Kutta_5(xend, hnew, y0, f)

        h.append(hnew)
        err_rk4.append(abs(ya(xend) - yrk4[-1]))
        err_rk5.append(abs(ya(xend) - yrk5[-1]))
```

Die doppelt-logarithmische Darstellung des absoluten Fehlers der beiden Verfahren für verschiedene Schrittweiten h zeigt folgende in der Vorlesung behandelte Erkenntnisse:

- Der Fehler geht beim Verfahren höherer Ordnung schneller gegen 0
- Wird die Schrittweite im Verfahren höherer Ordnung zu klein gewählt, wird der Fehler wieder grösser.



3) Implementation RK45 Verfahren mit Schrittweitensteuerung

In einem nächsten Schritt wurden die Verfahren 4. und 5. Ordnung zusammengefasst und mit einer Schrittweitensteuerung versehen. Aus Platzgründen soll hier nur der Ausschnitt der Funktion mit der Schrittweitensteuerung abgebildet werden.

```
# Schrittweite anpassen
if Delta < tol/20:
    h_neu = 2*h
elif Delta <= tol:
    h_neu = h
else:
    h_neu = h/2
    h = h_neu
    continue  # y_{k+1} nicht aktzeptieren und wiederholen mit halber Schrittweite
```

4) RK45-Verfahren mit verschiedenen Toleranzen lösen

Das in 3) definierte RK45-Verfahren wurde für verschiedene Toleranzen aufgerufen und jeweils der maximale absolute Fehler berechnet (rechts). Logischerweise wird der maximale Fehler kleiner wenn die Toleranz kleiner gewählt wird.

```
# ------------ Lösen des Modellproblems mit dem RK45-Verfahren für versch. Toleranzen ------------
for j in range(1, 13, 1):
    # Toleranz festlegen und RK45 mit dieser Toleranz aufrufen
    tol = 10**-j
    xrk45, yrk45 = Runge_Kutta_45(2, 0.1, y0, f, tol)

    # Maximalen absoluten Fehler berechnen und ausgeben
    max_err = 0
    for k in range(len(xrk45)):
        max_err = max(max_err, abs(ya(xrk45[k]) - yrk45[k]))
    print(max_err)
```

Ausgabe:

```
0.0023867953353531313
0.0003779030025499175
2.66945956362008e-06
2.66945956362008e-06
2.66945956362008e-06
3.2221440791069256e-06
4.5218827038340237e-07
1.703253076357214e-07
4.236760631215475e-08
2.395318610126651e-09
6.484137671236567e-10
5.282707604692405e-11
3.11928260998684e-12
1.2230216839270724e-12
```