Li
Wenzler

Korrektur: lls 9,5/10

# notebook_1_f

May 6, 2025

## 1 Notebook 1 - ODE Löser

[56]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
```

In diesem Notebook wollen wir verschiedene ODE Löser anhand mehrerer Beispiele und Untersuchungen vergleichen. Wir werden die Löser allerdings nicht selbst implementieren, sondern zunächst Implementierungen benutzen welche im python-Paket `scipy` zur Verfügung gestellt werden. Genauer werden wir im Folgenden die Funktion `scipy.integrate.solve_ivp` nutzen. Machen Sie sich deshalb zunächst mit der Dokumentation vetraut.

## 2 Aufgabe 1: Populationsmodelle

3/3

In der ersten Aufgabe wollen wir uns mit dem SIR-Modell beschäftigen. Genauer betrachten wir das System

$$\begin{aligned} \dot{s} &= -\sigma i s & s(0) &= s_0 \geq 0 \\ \dot{i} &= \sigma i s - i & i(0) &= i_0 \geq 0 \end{aligned}$$

wobei wir $\sigma = 3$ wählen. Weiter ist $r(t) = 1 - i(t) - s(t)$ für alle $t \geq 0$ und $r_0 = 1 - s_0 - i_0 \geq 0$.

Definieren Sie den Modellparameter $\sigma$. Setzen Sie anschließend sinnvolle Anfangswerte $s_0, i_0$ und $r_0$.

[57]:
```
# Reproduction number sigma.
sigma = 3
# Initial values s0, i0 and r0.
s0 = 0.99
i0 = 0.01
r0 = 0
```

Implementieren Sie die rechte Seite des SIR-Modells als Funktion `rhs_SIR(t, y)`, die ein Array mit den Änderungsraten zurückgibt.

[58]:
```
def rhs_SIR(t, y):
    prod = np.einsum("ij, j... -> i...",
```

einsum ist nett für optimierten Code, aber bei kleinen Rsp. eher unnötig

```
                    sigma * np.array([[-1], [1]]),
                    np.prod(y, axis=0, keepdims=True))
    add_i = np.einsum("ij, j... -> i...",
                      np.array([[0, 0], [0, -1]]),
                      y)
    # dsdt = ...
    # didt = ...
    return prod + add_i
```

Lösen Sie das SIR-Modell numerisch mit scipy.integrate.solve_ivp über einen Zeitraum von 21 Tagen. Übergeben Sie in `scipy.integrate.solve_ivp` als keyword Argument `t_eval = np.linspace(0,21,101)`.

Welche Methode wird als default für die Integration der Differentialgleichung benutzt? Was bewirkt das Argument `t_eval`? Wie viele Zeitschritte hat der Integrator gemacht? Wie passt das zu der Anzahl der Funktionsauswertungen?

[59]:
```
t_eval = np.linspace(0,21,101)
sol = scipy.integrate.solve_ivp(fun=rhs_SIR, t_span=[0, 21],
                                y0=np.array([s0, i0]), t_eval=t_eval)

print(f"Anzahl an f Auswertungen: {sol.nfev}")
```

```
Anzahl an f Auswertungen: 116
```

<– Platz für Ihre Antwort –>

Als Default wird RK45 als Solver genutzt

t_eval ist die Menger der Zeitpunkte, zu denen der Solver eine Approximation geben soll. Das hat keine Auswirkungen auf die Schrittweitensteuerung, falls ein Element von `t_eval` nicht direkt von dem Verfahren approximiert wird, wird als Approximation die stetige Approximation genutzt

Wird `t_eval=None` ausgeführt, so wird eine Approximation an `19` Zeitpunkten zurückgegeben, also macht der Integrator 19 Zeitschritte inklusive dem initialen Wert. Da der Integrator `116` f Auswertungen macht folgt mit `18 * 5 = 90` und `116 - 1 - 90 = 25`, dass `5` mal eine Wiederholung für eine Schrittweitenanpassung gemacht wurde ~~—~~ sollten 6 sein (✓)

[60]:
```
sol
```

[60]:
```
  message: The solver successfully reached the end of the integration interval.
  success: True
   status: 0
        t: [ 0.000e+00  2.100e-01 …  2.079e+01  2.100e+01]
        y: [[ 9.900e-01  9.823e-01 …  5.838e-02  5.838e-02]
            [ 1.000e-02  1.509e-02 …  3.198e-07  2.689e-07]]
      sol: None
 t_events: None
 y_events: None
     nfev: 116
```

```
     njev: 0
      nlu: 0
```

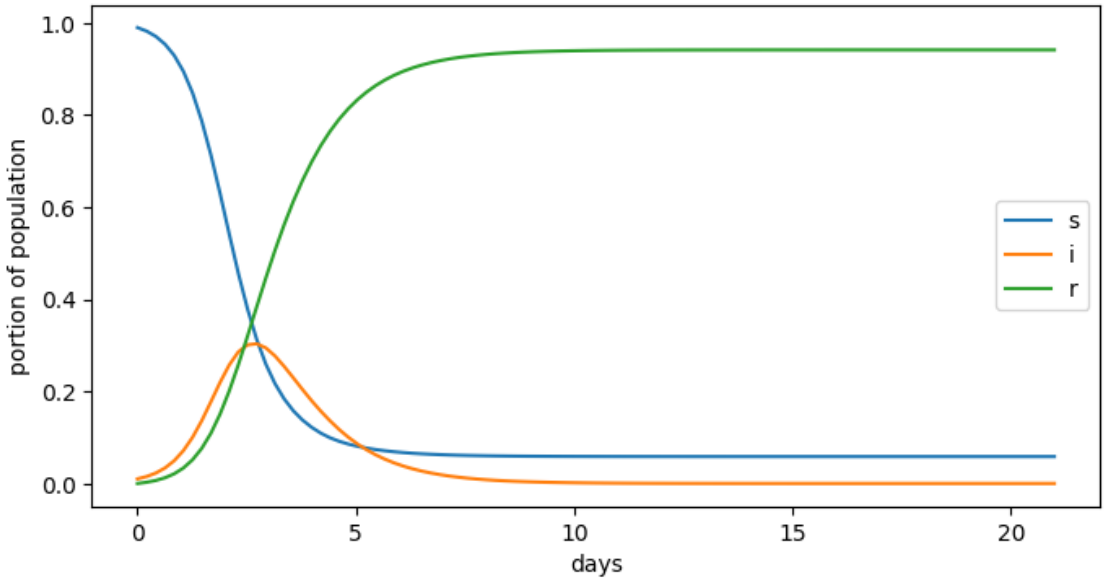
Die Funktion `scipy.integrate.solve_ivp` gibt ein Lösungsobjekt mit einer ganzen Reihe von Rückgaben zurück. Lesen Sie in der Dokumentation nach, was welcher Rückgabewert beschreibt. Sie können über den . Operator auf die verschiedenen Rückgaben zugreifen, z.B. `sol.t`. Was beschreibt die Rückgabe `sol.nfev`?

<– Platz für Ihre Antwort –>

`sol.nfev` beschreibt die Anzahl der f Auswertungen

Erstellen Sie ein Liniendiagramm mit den Verläufen von s(t), i(t) und r(t) über die Zeit. Nutzen Sie dazu das Paket `matplotlib.pyplot`. Erstellen Sie eine Legende und beschriften Sie die Achsen.

[61]:
```
s = sol.y[0,:]
i = sol.y[1,:]
r = np.ones_like(s) - s - i
# Plot the data on three separate curves for S(t), I(t) and R(t)
fig = plt.figure(figsize=(8,4))
plt.xlabel("days")
plt.ylabel("portion of population")
plt.plot(t_eval, s, label="s")
plt.plot(t_eval, i, label="i")
plt.plot(t_eval, r, label="r")
plt.legend(loc="right")
plt.show()
```

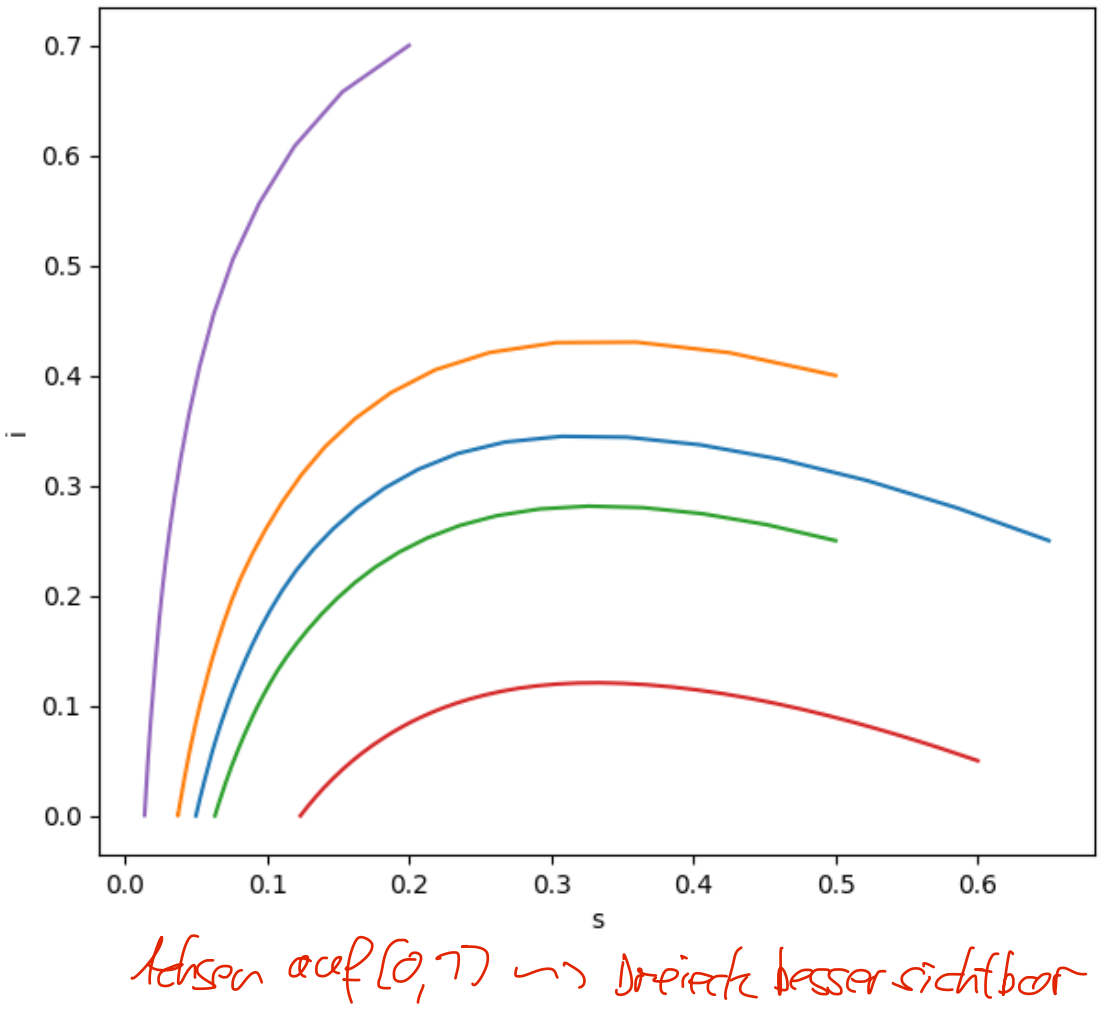



Als nächstes wollen wir ein Phasendiagramm des SIR-Modells im $(s, i)$-Raum erstellen. Sie visual-

isieren dabei die Richtungsfelder der Dynamik sowie einige Beispieltrajektorien.

Berechnen Sie zunächst für verschiedene Anfangswerte die Lösungs-Trajektorien im $(s, i)$-Raum und zeichnen Sie diese. Nutzen Sie dazu die Funktion `scipy.integrate.solve_ivp` für mehrere Anfangswerte.

[62]:
```
# Calculate examples trajectories for
s0s = [0.65,0.5,0.5,0.6, 0.2]
i0s = [0.25, 0.4,0.25, 0.05, 0.7]
t_eval = np.linspace(0,21,161)
# Create the phase plot
plt.figure(11, figsize=(7, 6))

def plot_trajectories():
  plt.xlabel("s")
  plt.ylabel("i")
  for (s0, i0) in zip(s0s, i0s):
    sol = scipy.integrate.solve_ivp(fun=rhs_SIR, t_span=[0,21],
                                    y0=np.array([s0, i0]), t_eval=t_eval)
    s = sol.y[0,:]
    i = sol.y[1,:]
    r = np.ones_like(s) - s - i
    plt.plot(s, i)
plot_trajectories()
plt.show()
```




Achsen auf [0,1] => Dreieck besser sichtbar

Das Paket `matplotlib` bietet auch direkt Funktionen an, um Phasendiagramme zu erzeugen.

Erstellen Sie dazu zunächst ein `meshgrid` aus Wertepaaren $(s, i)$ für $s,i \in [0,1]$. Nutzen Sie beispielsweise die Funktionen `np.linspace` und `np.meshgrid`.

[63]:
```
# Define the grid
res = 100
s = np.linspace(0, 1, res)
i = np.linspace(0, 1, res)
S, I = np.meshgrid(s, i)
print(S.shape)
```

```
(100, 100)
```

Berechnen Sie die Richtungsfelder $ds/dt, di/dt$ auf dem Gitter.

[64]:
```
# Initialize dS/dt and dI/dt
t = 0 # system is autonomous
dy = rhs_SIR(t, np.stack((S, I), axis=0))
```

```
dS = dy[0,:,:]
dI = dy[1,:,:]
```

Wir maskieren nun alle Werte für die $s + i > 1$ gilt.

[65]:
```
# Mask the region where S + I > 1
mask = S + I > 1

# Apply the mask
dS = np.ma.array(dS, mask=mask)
dI = np.ma.array(dI, mask=mask)
```

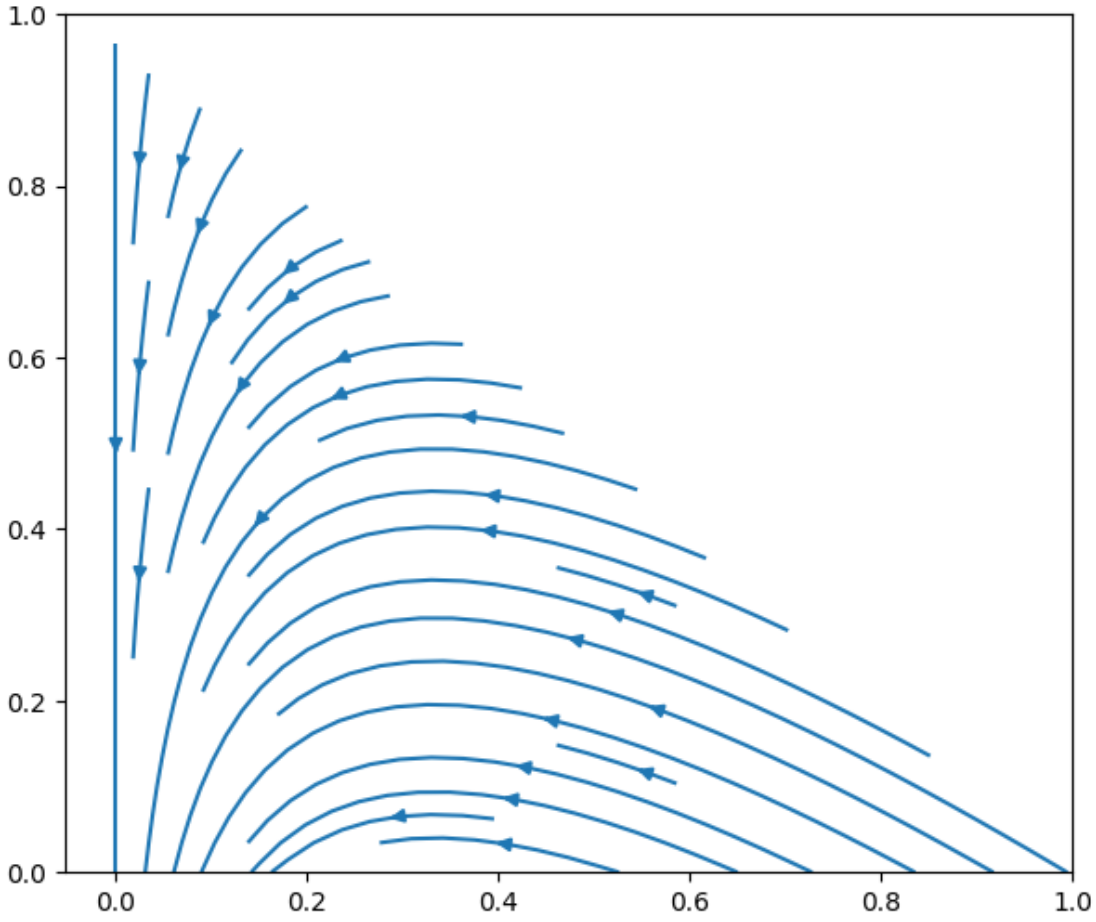
Erstellen Sie nun ein Phasendiagramm. Zeichnen Sie dabei nur Werte mit $s + i \leq 1$. Warum macht diese Restriktion und die damit verbundene Maskierung Sinn?

Hinweis: Sie können zwischen `plt.streamplot(...)` und `plt.quiver(...)` wählen, um das Richtungsfeld zu visualisieren. `streamplot` erzeugt glattere Linien, `quiver` verwendet Pfeile. Schlagen Sie die Dokumentation der jeweiligen Funktion oder in der matplotlib gallery nach, um sich mit den Funktionen vertraut zu machen.

<– Platz für Ihre Antwort –>

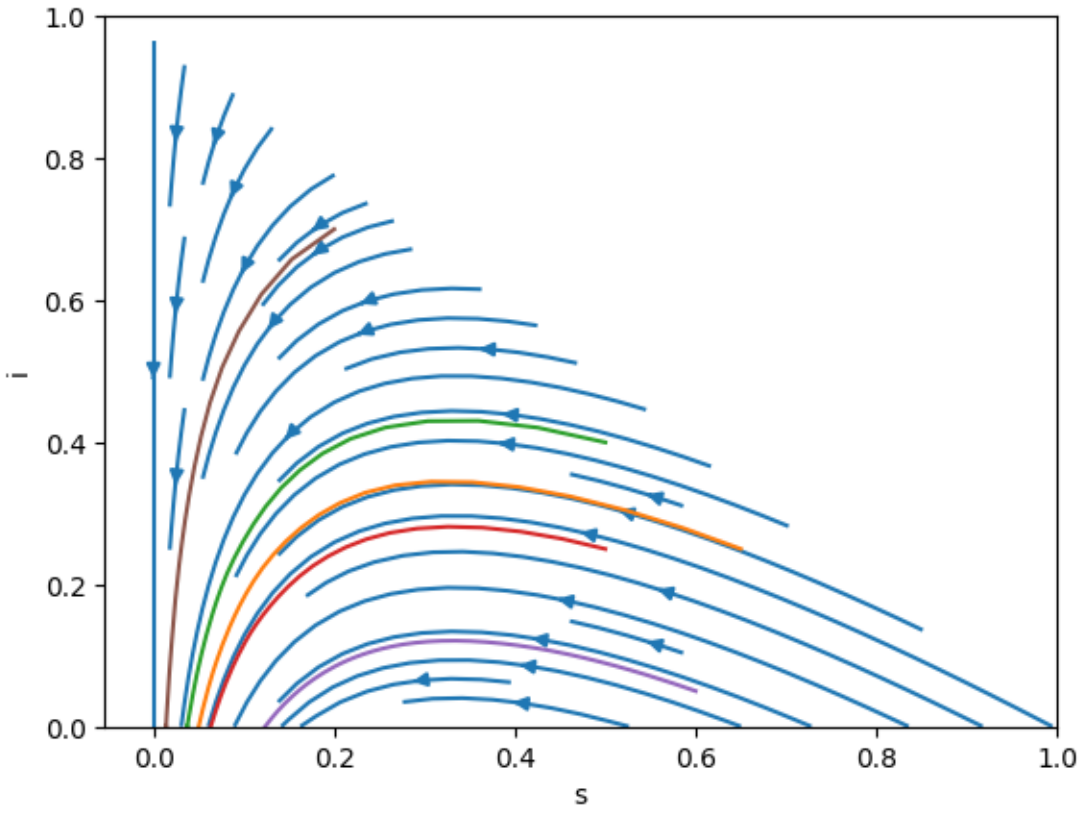
Das macht Sinn, da $s + i \leqslant 1 \iff r \geqslant 0$ und ein negativer Anteil an Genesenen macht keinen Sinn ✓

[66]:
```
# Create the phase plot
plt.figure(figsize=(7, 6))
plt.streamplot(s, i, dS, dI)
plt.show()
```




Zeichnen Sie Lösungstrajektorien zu den verschiedenen Anfangswerten zusätzlich in Ihr Phasendiagramm ein.

[67]:
```
plt.figure(11)
plt.streamplot(s, i, dS, dI)
plot_trajectories()
plt.show()
```




## 3 Aufgabe 2 - Die van-der-Pol-Gleichung

Als zweites Beispiel betrachten wir die van-der-Pol-Gleichung (siehe Beispiel 9.19 im Skript)

$$y_1' = y_2 \qquad y_1(0) = 2$$
$$y_2' = \mu(1 - y_1^2)y_2 - y_1 \qquad y_2(0) = 0$$

mit positivem Paramter $ $ > 0 $ $.

Schreiben Sie zunächst eine Funktion `rhs_vdP(t,y)` welcher die rechte Seite der van-der-Pol-Gleichung in Abhängigkeit der Zeit `t`und dem Lösungsvektor `y` beschreibt.

[68]:
```
mu = 2
rhs_vdP = lambda t,y: [c := np.transpose(np.squeeze(
    np.matmul(
        [np.place(a := np.tile(np.array([[0, 1], [-1, 0]], dtype=np.float64),
                               (y.shape[1] if len(y.shape) > 1 else 1, 1, 1)),
                  np.tile(np.array([[False, False], [False, True]]),
                          (y.shape[1] if len(y.shape) > 1 else 1, 1, 1)),
                  mu * (1 - y[0,...]**2)), a][1],
        np.transpose(y)[:, :, np.newaxis] if y.ndim == 2 else y[np.newaxis, :,␣
 ↪np.newaxis]), axis=2)),
```

```
                        np.squeeze(c, axis=1) if len(y.shape) == 1 else c][1]
rhs_vdP(5,np.array([0,1])) # should result in array([1, 2])
```

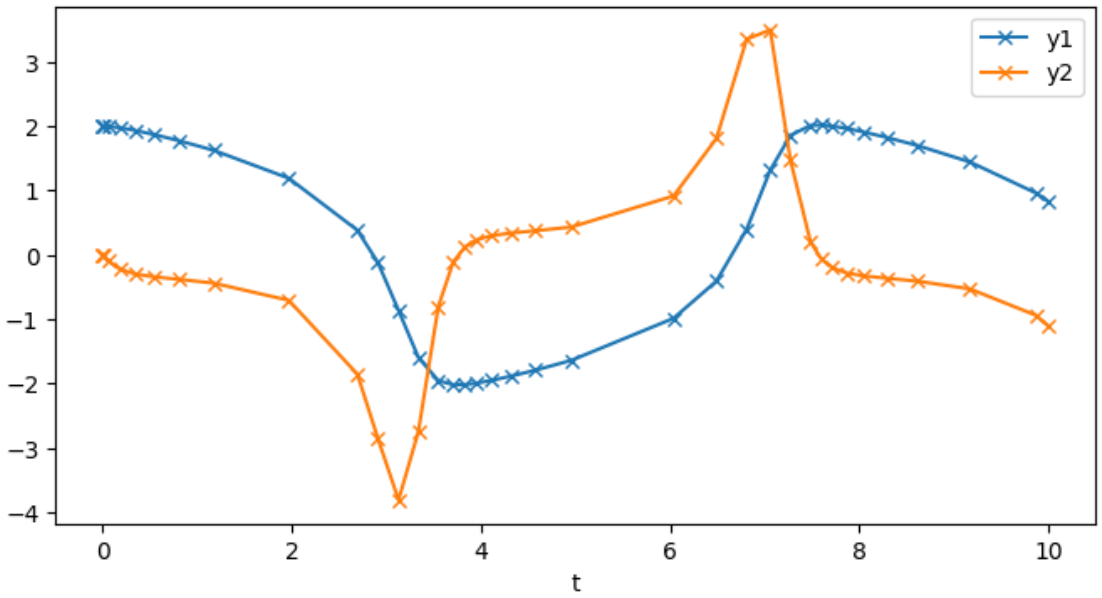
[68]:
```
array([1., 2.])
```

Nutzen Sie nun die Funktion `scipy.integrate.solve_ivp` um auf dem Zeitinterval [0,10] eine Approximation der Lösung der van-der-Pol-Gleichung zu berechnen.

[69]:
```
t_span = (0,10)
y0 = np.array([2,0])
sol = scipy.integrate.solve_ivp(fun=rhs_vdP, t_span=t_span, y0=y0)
```

Erstellen Sie eine Abbildung mithilfe von `matplotlib` in der Sie die beiden Lösungen $y_1$ und $y_2$ über die Zeit plotten. Benutzen Sie verschiedene Marker um die einzelnen Schritte des Lösers zu sehen.

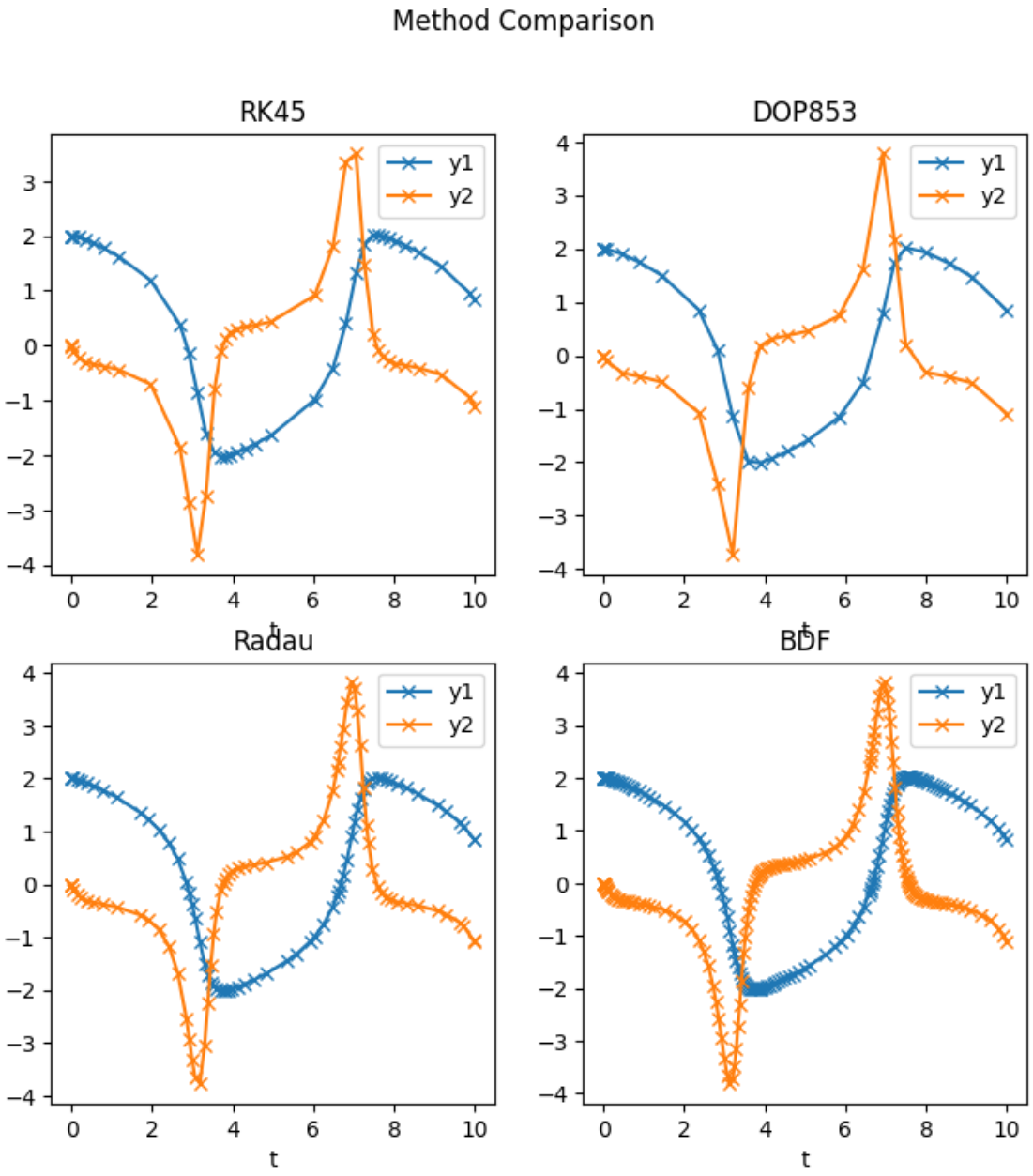
[70]:
```
y1 = sol.y[0,:]
y2 = sol.y[1,:]
fig = plt.figure(figsize=(8, 4))
plt.xlabel("t")
plt.plot(sol.t, y1, "x", linestyle="-", label="y1")
plt.plot(sol.t, y2, "x", linestyle="-", label="y2")
plt.legend()
plt.show()
```




Testen Sie die Methoden `RK45`, `DOP853`, `Radau`, `BDF`. Welche Methoden stecken hinter diesen Kürzeln. Welche Verfahren sind explizit, welche implizit?

Erzeugen Sie für jede, der vier Methoden ein Schaubild der Lösungen $y_1$ und $y_2$ über die Zeit. Nutzen Sie dazu `plt.subplots`. Geben Sie jeder Grafik einen Titel und beschriften Sie die Achsen. Was fällt Ihnen an der Schrittweitensteuerung auf?

[71]:
```
sol_RK45 = scipy.integrate.solve_ivp(fun=rhs_vdP, t_span=t_span, y0=y0,␣
 ↪method="RK45")
sol_DOP853 = scipy.integrate.solve_ivp(fun=rhs_vdP, t_span=t_span, y0=y0,␣
 ↪method="DOP853")
sol_Radau = scipy.integrate.solve_ivp(fun=rhs_vdP, t_span=t_span, y0=y0,␣
 ↪method="Radau", vectorized=True)
sol_BDF = scipy.integrate.solve_ivp(fun=rhs_vdP, t_span=t_span, y0=y0,␣
 ↪method="BDF", vectorized=True)

## plot in subplots
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(8, 8))
fig.suptitle("Method Comparison")

#RK45
ax1.set_title("RK45")
ax1.set_xlabel("t")
ax1.plot(sol_RK45.t, sol_RK45.y[0,:], "x", linestyle="-", label="y1")
ax1.plot(sol_RK45.t, sol_RK45.y[1,:], "x", linestyle="-", label="y2")
ax1.legend()

#DOP853
ax2.set_title("DOP853")
ax2.set_xlabel("t")
ax2.plot(sol_DOP853.t, sol_DOP853.y[0,:], "x", linestyle="-", label="y1")
ax2.plot(sol_DOP853.t, sol_DOP853.y[1,:], "x", linestyle="-", label="y2")
ax2.legend()

#Radau
ax3.set_title("Radau")
ax3.set_xlabel("t")
ax3.plot(sol_Radau.t, sol_Radau.y[0,:], "x", linestyle="-", label="y1")
ax3.plot(sol_Radau.t, sol_Radau.y[1,:], "x", linestyle="-", label="y2")
ax3.legend()

#BDF
ax4.set_title("BDF")
ax4.set_xlabel("t")
ax4.plot(sol_BDF.t, sol_BDF.y[0,:], "x", linestyle="-", label="y1")
ax4.plot(sol_BDF.t, sol_BDF.y[1,:], "x", linestyle="-", label="y2")
ax4.legend()

plt.show()
```




<– Platz für Ihre Antwort –>

`RK45` ist ein explizites Runge - Kutta Verfahren mit `6` Stufen und Ordnung `5`, für die Schrittweitensteuerung wird ein eingebettetes Verfahren mit `7` Stufen und Ordnung `4` genutzt

`DOP853` ist ein explizites Runge - Kutta Verfahren mit `12` Stufen und Ordnung `8`, für die Schrittweitensteuerung werden 2 eingebettete Verfahren genutzt

`Radau` ist ein Kollokationsverfahren mit den Stützstellen der Radau - Quadraturformel mit `3` Stufen und Ordnung `5`, also äquivalent zu einem impliziten Runge - Kutta Verfahren mit `3` Stufen und Ordnung `5`

`BDF` ist ein implizites BDF Verfahren mit variabler Ordnung 1 bis 5

**Schrittweite**

Da `mu = 2` ist die DGL nicht steif, also funktionieren auch explizite Verfahren gut

`DOP853` hat am wenigsten Schritte, was naheligend ist, da es die höchste Ordnung hat

`RK45` und `Radau` haben beide Ordnung `5`, aber Radau macht mehr Schritte, was daran liegen kann, dass `Radau` die Schrittweitensteuerung mit einem eingebetteten Verfahren Ordnung `3` macht und `RK45` die Schrittweitensteuerung mit einem eingebetteten Verfahren Ordnung `4` macht, also `Radau` den lokalen Fehler im Allgemeinen möglicherweise größer schätzt

`BDF` macht am meisten Schritte, was möglicherweise daran liegt, dass die Implementierung möglicht lange eine konstante Schrittweite nutzt, um die Berechnung zu vereinfachen

Untersuchen Sie, wie sich die Wahl von `atol` und `rtol` auf die Lösung der Van-der-Pol-Gleichung mit $\mu = 100$ und $t \in [0, 400]$ auswirkt.

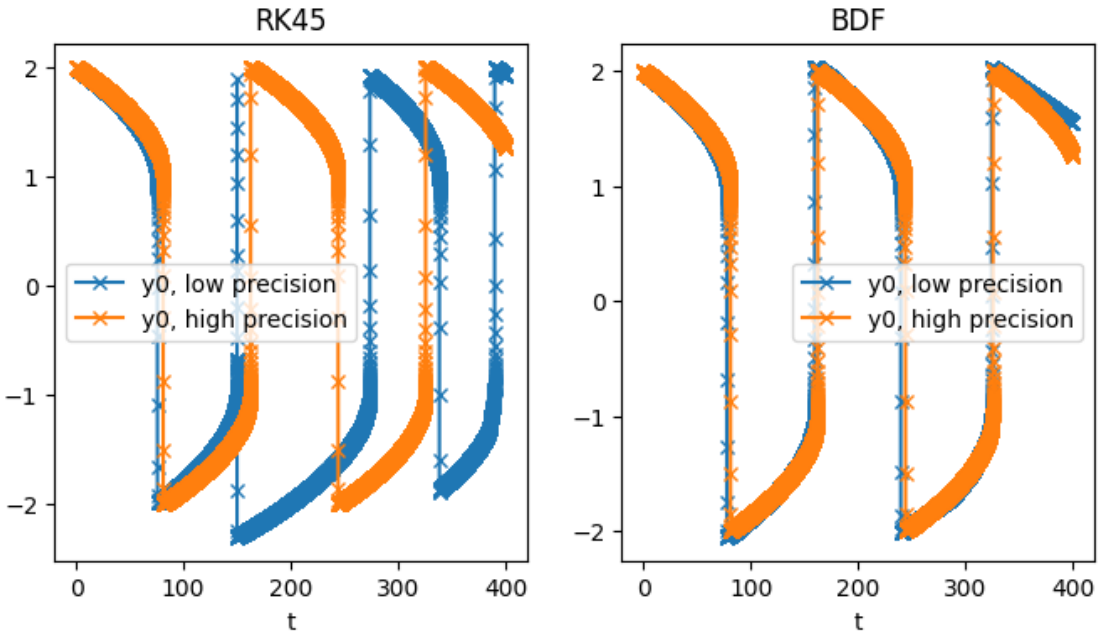
Vergleichen Sie die Lösung für zwei verschiedene Toleranzeinstellungen: - Fall A (niedrige Genauigkeit): rtol=1e-1, atol=1e-1 - Fall B (hohe Genauigkeit): rtol=1e-8, atol=1e-10

Zeichnen Sie für beide Fälle die erste Komponente der Lösung $y_1$ in einem Plot und kommentieren Sie die Unterschiede.

[72]:
```
# Parameter für die Van-der-Pol-Gleichung
mu = 100
# Anfangswerte
y0 = np.array([2, 0])
t_span = (0, 400)
t_eval = np.linspace(t_span[0], t_span[1], 80000)

# rhs
rhs_vdP = lambda t,y: [c := np.transpose(np.squeeze(
    np.matmul(
        [np.place(a := np.tile(np.array([[0, 1], [-1, 0]], dtype=np.float64),
                               (y.shape[1] if len(y.shape) > 1 else 1, 1, 1)),
                  np.tile(np.array([[False, False], [False, True]]),
                          (y.shape[1] if len(y.shape) > 1 else 1, 1, 1)),
                  mu * (1 - y[0,...]**2)), a][1],
        np.transpose(y)[:, :, np.newaxis] if y.ndim == 2 else y[np.newaxis, :,
 ↪np.newaxis]), axis=2)),
                       np.squeeze(c, axis=1) if len(y.shape) == 1 else c][1]
# case A: low precision
sol_lo = scipy.integrate.solve_ivp(fun=rhs_vdP, t_span=t_span, y0=y0,
                                   t_eval=t_eval, method="RK45",
                                   atol=1e-1, rtol=1e-1)
sol_lo_BDF = scipy.integrate.solve_ivp(fun=rhs_vdP, t_span=t_span, y0=y0,
                                   t_eval=t_eval, method="BDF",
                                   vectorized=True, atol=1e-1, rtol=1e-1)
# case B: high precision
```

```
sol_hi = scipy.integrate.solve_ivp(fun=rhs_vdP, t_span=t_span, y0=y0,
                                   t_eval=t_eval, method="RK45",
                                   atol=1e-10, rtol=1e-8)
sol_hi_BDF = scipy.integrate.solve_ivp(fun=rhs_vdP, t_span=t_span, y0=y0,
                                   t_eval=t_eval, method="BDF",
                                   vectorized=True, atol=1e-10, rtol=1e-8)
# plot of the solutions
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
# RK45
ax1.set_title("RK45")
ax1.set_xlabel("t")
ax1.plot(t_eval, sol_lo.y[0,:], "x", linestyle="-", label="y0, low precision")
ax1.plot(t_eval, sol_hi.y[0,:], "x", linestyle="-", label="y0, high precision")
ax1.legend()
# BDF
ax2.set_title("BDF")
ax2.set_xlabel("t")
ax2.plot(t_eval, sol_lo_BDF.y[0,:], "x", linestyle="-", label="y0, low␣
 ↪precision")
ax2.plot(t_eval, sol_hi_BDF.y[0,:], "x", linestyle="-", label="y0, high␣
 ↪precision")
ax2.legend()
plt.show()
```

Ein Laufzeit-Vergleich zw. einfacher und eurer Implem. wäre eventuell noch interessant gewesen



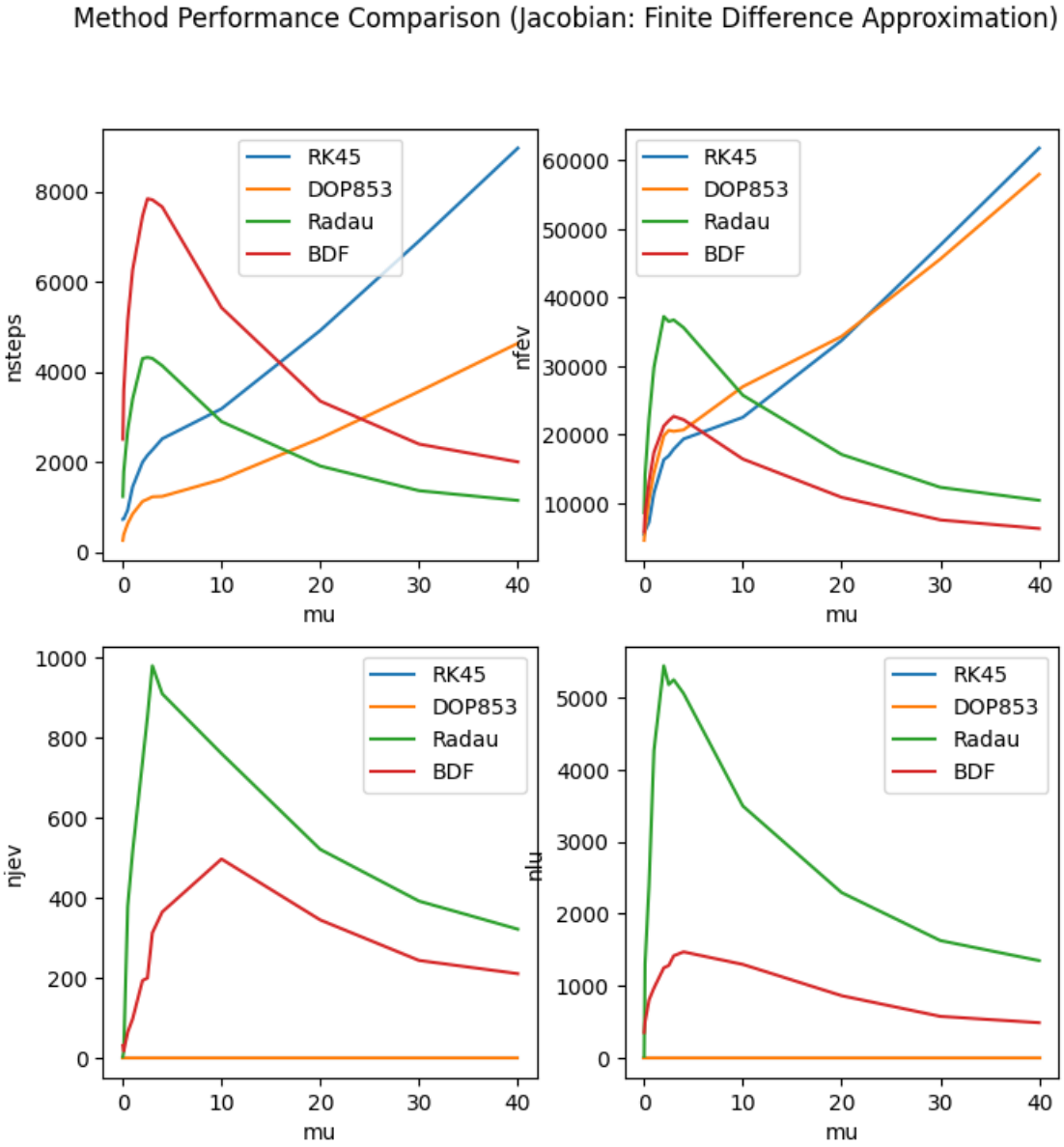

<– Platz für Ihre Antwort –>

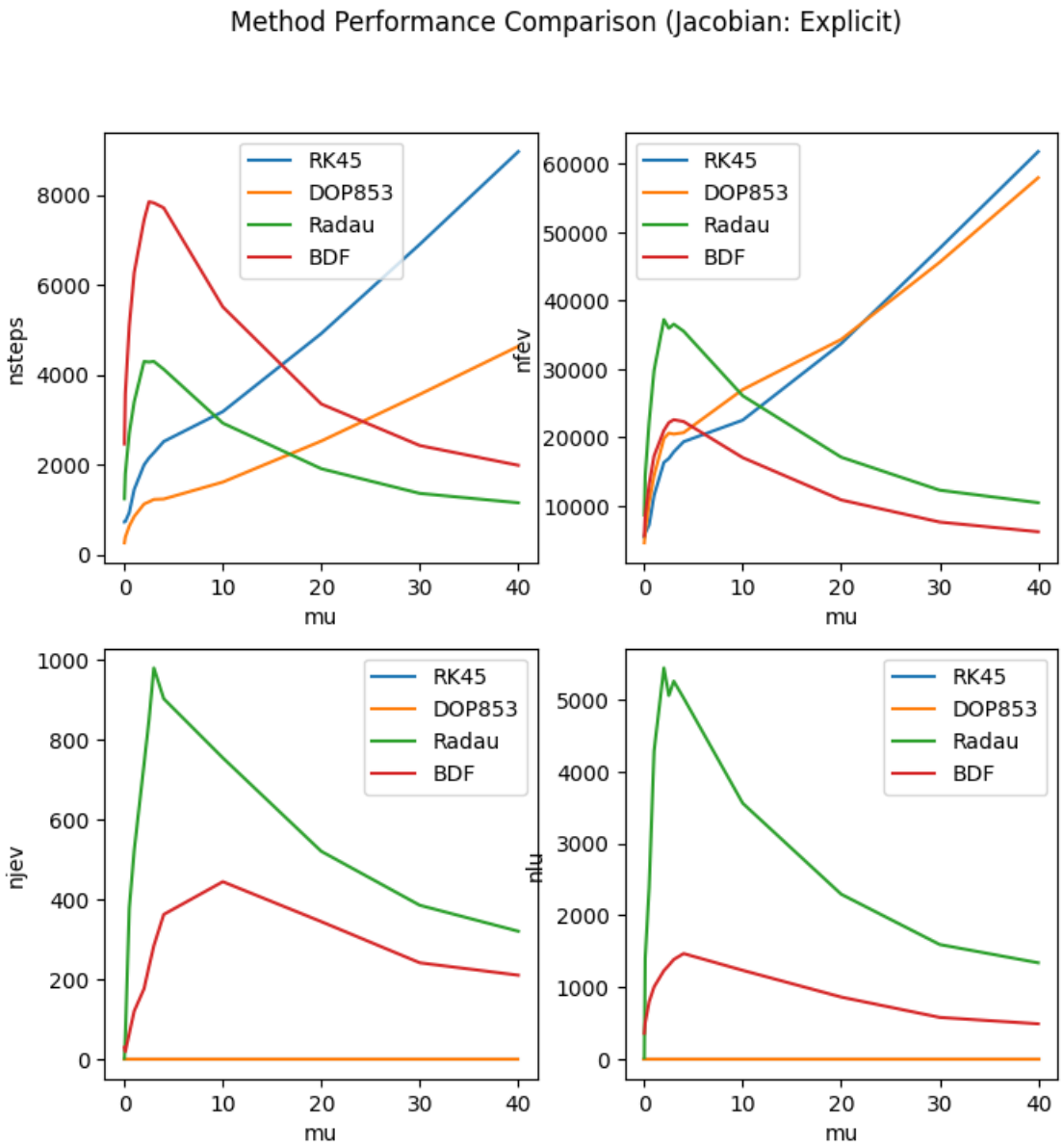
Die geringe Präzision Lösung von `RK45` macht bei `t=150` einen Sprung, die hohe Präzision Lösung von `RK45` ähnelt den Lösungen von `BDF`, also ist die Van der Pool DGL für große $\mu$ steif

Untersuchen Sie wie sich die Anzahl der Zeitschritte, die Anzahl der Auswertungen der rechten Seite, die Anzahl der Auswertungen der Jacobi-Matrix und die Anzahl der LU-Zerlegungen verändert, wenn Sie verschiedene Parameter $\mu$ wählen. Nutzen Sie die Methoden `RK45`, `DOP853`, `Radau`, `BDF` und übergeben Sie diesmal kein `t_eval`. Ändert sich die Anzahl der Auswertungen der Jacobi-Matrix, wenn Sie die exakte Jacobi-Matrix übergeben? Wählen Sie `rtol=1e-4` und `atol=1e-7`. Stellen Sie die Anzahl der verschiedenen Auswertungen in einem Schaubild über $\mu$ dar.

[73]:
```
mus = np.array([0.01,0.1,0.5,1.0,2.0,2.5,3.0,4.0,10.0,20.0,30.0,40.0])

nsteps = np.zeros((4,len(mus)))
nfevs = np.zeros((4,len(mus)))
njevs = np.zeros((4,len(mus)))
nlus = np.zeros((4,len(mus)))

# rhs
rhs_vdP = lambda t,y,mu: [c := np.transpose(np.squeeze(
    np.matmul(
        [np.place(a := np.tile(np.array([[0, 1], [-1, 0]], dtype=np.float64),
                               (y.shape[1] if len(y.shape) > 1 else 1, 1, 1)),
                  np.tile(np.array([[False, False], [False, True]]),
                          (y.shape[1] if len(y.shape) > 1 else 1, 1, 1)),
                  mu * (1 - y[0,...]**2)), a][1],
        np.transpose(y)[:, :, np.newaxis] if y.ndim == 2 else y[np.newaxis, :,␣
 ↪np.newaxis]), axis=2)),
                      np.squeeze(c, axis=1) if len(y.shape) == 1 else c][1]

jac = lambda t,y,mu: np.array([
    [0, 1],
    [-2 * mu * y[0] * y[1] - 1, mu * (1 - y[0]**2)]
])

def method_performance_comparison(jac=None):
  methods = ["RK45", "DOP853", "Radau", "BDF"]

  for i, method in enumerate(methods):
    for j, mu in enumerate(mus):
      sol = scipy.integrate.solve_ivp(fun=lambda t,y: rhs_vdP(t, y, mu),␣
 ↪t_span=t_span, y0=y0, method=method, vectorized=i>1 and jac==None,␣
 ↪atol=1e-7, rtol=1e-4, jac=None if jac == None else lambda t,y: jac(t, y, mu))
      nsteps[i, j] = len(sol.t) - 1 # solution at t=0 is given
      nfevs[i, j] = sol.nfev
      njevs[i, j] = sol.njev
      nlus[i, j] = sol.nlu

  # plot
```

```
  fig, axs = plt.subplots(2, 2, figsize=(8, 8))
  ((ax_nsteps, ax_nfev), (ax_njev, ax_nlu)) = axs
  fig.suptitle(f"Method Performance Comparison (Jacobian: {'Finite Difference␣
 ↪Approximation' if jac == None else 'Explicit'})")

  for ax1 in axs:
    for ax in ax1:
      ax.set_xlabel("mu")

  ax_nsteps.set_ylabel("nsteps")
  ax_nfev.set_ylabel("nfev")
  ax_njev.set_ylabel("njev")
  ax_nlu.set_ylabel("nlu")

  for i, method in enumerate(methods):
    ax_nsteps.plot(mus, nsteps[i,:], "-", label=method)
    ax_nfev.plot(mus, nfevs[i,:], "-", label=method)
    ax_njev.plot(mus, njevs[i,:], "-", label=method)
    ax_nlu.plot(mus, nlus[i,:], "-", label=method)

  for ax1 in axs:
    for ax in ax1:
      ax.legend()

  plt.show()

method_performance_comparison()
method_performance_comparison(jac=jac)
```


Method Performance Comparison (Jacobian: Finite Difference Approximation)
RK45
DOP853
Radau
BDF
nsteps
nfev
njev
nlu
mu
0
2000
4000
6000
8000
10000
20000
30000
40000
50000
60000
200
400
600
800
1000
1000
2000
3000
4000
5000
0
10
20
30
40




<– Platz für Ihre Antwort –>

Die Angabe der exakten Jacobi Matrix hat keine sichtbaren Auswirkungen auf die Anzahl der Auswertungen der Jacobi Matrix bei `BDF` und `Radau` Die Anzahl der Auswertungen der Jacobi Matrix ist kleiner als die Anzahl der LU Zerlegungen, es wird also wahrscheinlich das vereinfachte Newton Verfahren verwendet bei der Implementierung des impliziten Runge Kutta Verfahrens `Radau`

Die Anzahl der Schritte nimmt bei den expliziten Verfahren `RK45` und `DOP853` mit $\mu$ zu, die Gleichung wird steifer Die Anzahl der Schritte nimmt bei den impliziten Verfahren `BDF` und `Radau` mit hinreichend großem $\mu$ ab, da implizite Verfahren auch für steife Gleichungen funktionieren und das Beispiel $\mu = 100$ nahe legt, dass die Lösung mit höherem $\mu$ in großen Bereichen glatter wird

Die expliziten Runge Kutta Verfahren `RK45` und `DOP853` verwenden keine Jacobi Matrix

## 4 Aufgabe 3: chemische Reaktion von Robertson (1966) 3,5/4

Als nächstes betrachten wir ein Beispiel einer chemischen Reaktion von Robertson (vgl. Abschnitt 11.1 im Skript). Die Konzentartion dreier Substanzen $A, B, C$ erfüllt die folgende Differentialgleichung

$$\begin{aligned} A &: y_1' = -0.04y_1 + 10^4 y_2 y_3 \\ B &: y_2' = 0.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2 \\ C &: y_3' = 3 \cdot 10^7 y_2^2 \end{aligned}$$

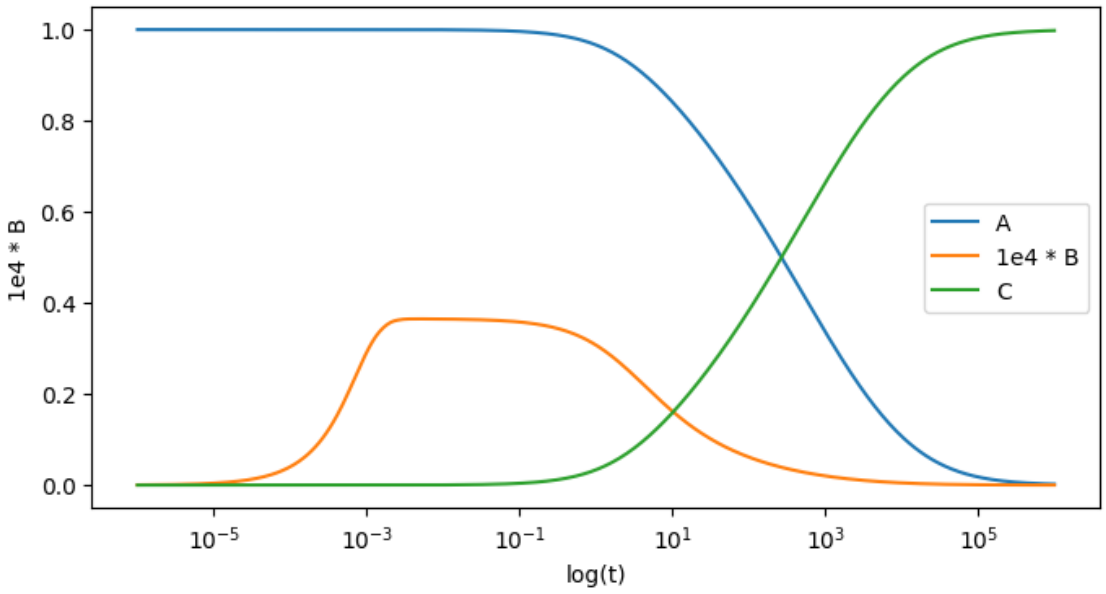
mit Anfangswerten $y_1(0) = 1, y_2(0) = 0, y_3(0) = 0$.

Schreiben Sie eine Funktion `rhs_Robertson(t,y)`, welche die rechte Seite dieser Differentialgleichung beschreibt. Lösen Sie anschließend die Differentialgleichung auf dem Zeitinterval $t \in [0, 10^6]$ mit dem Radau-Verfahren. Wählen Sie als Auswertungspunkte `t_eval=np.geomspace(1e-6,1e6,501)`. Was ist der Unterschied von `np.geomspace` zu `np.linspace`?

[74]:
```
def rhs_Robertson(t,y):
    y_v = y if len(y.shape) > 1 else y[:,np.newaxis]
    A = -0.04 * y_v[0,:] + 1e4 * y_v[1,:] * y_v[2,:]
    B = 0.04 * y_v[0,:] - 1e4 * y_v[1,:] * y_v[2,:] - 3 * 1e7 * (y_v[1,:] ** 2)
    C = 3 * 1e7 * (y_v[1,:] ** 2)
    dy_dt = np.stack((A, B, C), axis=0)
    return dy_dt if len(y.shape) > 1 else np.squeeze(dy_dt, axis=1)

t_span = (0, 1e6)
t_eval = np.geomspace(1e-6, 1e6, 501) # number from 1e-6 to 1e6 using geometric␣
 ↪sequence such that the first number is 1e-6 and the last number is 1e6
y0 = np.array([1, 0, 0])

sol_RADAU = scipy.integrate.solve_ivp(fun=rhs_Robertson, t_span=t_span, y0=y0,
                                      t_eval=t_eval, method="Radau",
                                      vectorized=True)
```

Stellen Sie die Lösung der Differentialgleichung über die Zeit in einem Schaubild dar. Skalieren Sie die $x$-Achse logarithmisch und skalieren Sie die Konzentration der Substanz $B$ mit $10^4$

[75]:
```
fig = plt.figure(figsize=(8, 4))
plt.xlabel("log(t)")
plt.ylabel("1e4 * B")
plt.xscale("log")
plt.plot(t_eval, sol_RADAU.y[0,:], "-", label="A")
plt.plot(t_eval, 1e4 * sol_RADAU.y[1,:], "-", label="1e4 * B")
plt.plot(t_eval, sol_RADAU.y[2,:], "-", label="C")
plt.legend()
plt.show()
```




In diesem Paper wird in Abschnitt 7.2 ein etwas komplexeres Beispiel dieser Differentialgleichung betrachtet. Dabei werden Lösungen $u(x,t), v(x,t), w(x,t)$ für $x \in [0,1]$ und $t \in [0,100]$ durch die partielle Differentialgleichung

$$
\begin{aligned}
u_t &= -0.04u + 10^4 vw + \alpha u_{xx} \\
v_t &= 0.04u - 10^4 vw - 3 \cdot 10^7 v^2 + \alpha v_{xx} \\
w_t &= 3 \cdot 10^7 v^2 + \alpha w_{xx}
\end{aligned}
$$

beschrieben. Als Anfangswerte werden $u(x,0) = 1 + \sin(2\pi x), v(x,0) = w(x,0) = 0$ und als Randwerte homogenene Neumann-Randdaten, also $u_x = v_x = w_x = 0$ gewählt. Für den Diffusionsparameter $\alpha$ wählen wir $\alpha = 0.02$. Im Ort wird die Differentialgleichung mit finiten Differenzen diskretisiert, siehe Abschnitt 12 im Skript.

Sie können die Funktion `scipy.sparse.linalg.LaplacianNd(...)` wie folgt benutzen um eine geeignete finite Differenzen Matrix zu assemblieren.

[76]:
```
import numpy as np
from scipy.sparse.linalg import LaplacianNd

n = 6
h = 1.0/(n-1)
xs = np.linspace(0.0,1.0, n)
grid_shape = xs.shape
lap = LaplacianNd(grid_shape, boundary_conditions='neumann',dtype=float)
dNM_Laplace = 1/(h**2)*lap.tosparse().toarray()
dNM_Laplace
```

[76]:
```
array([[-25.,  25.,   0.,   0.,   0.,   0.],
       [ 25., -50.,  25.,   0.,   0.,   0.],
       [  0.,  25., -50.,  25.,   0.,   0.],
       [  0.,   0.,  25., -50.,  25.,   0.],
       [  0.,   0.,   0.,  25., -50.,  25.],
       [  0.,   0.,   0.,   0.,  25., -25.]])
```

Legen Sie eine rechte Seite `rhs_RobertsonDiffusion(t,y)` an, die die rechte Seite der partiellen Differentialgleichung beschreibt. Nutzen Sie zur Diskretisierung im Ort `n=30` Punkte und verwenden Sie die Funktion `scipy.sparse.linalg.LaplacianNd(...)`.

Hinweis: Um die rechte Seite später in `scipy.integrate.solve_ivp` benutzen zu können, müssen Sie ein ein-dimensionales array zurückgeben. Nutzen Sie dazu slicing und die Funktion `np.concatenate(...)`.

[77]:
```
alpha = 0.02

n = 30
h = 1.0/(n-1)
xs = np.linspace(0.0,1.0, n)
grid_shape = xs.shape
lap = LaplacianNd(grid_shape, boundary_conditions='neumann',dtype=float)
dNM_Laplace = -1/(h**2)*lap.tosparse().toarray()

def rhs_RobertsonDiffusion(t,y):
    u = y[0:n]
    v = y[n:2*n]
    w = y[2*n:3*n]
    du_dxx = -np.matmul(dNM_Laplace, u)
    dv_dxx = -np.matmul(dNM_Laplace, v)
    dw_dxx = -np.matmul(dNM_Laplace, w)
    du_dt = -0.04 * u + 1e4 * v * w + alpha * du_dxx
    dv_dt = 0.04 * u - 1e4 * v * w - 3 * 1e7 * (v ** 2) + alpha * dv_dxx
    dw_dt = 3 * 1e7 * (v ** 2) + alpha * dw_dxx
    return np.concatenate((du_dt, dv_dt, dw_dt))
```

Schreiben Sie eine Funktion `jacobian_RobertsonDiffusion(...)`, welche die exakte Jacobimatrix der Reaktionsgleichung mit Diffusion zurückgibt.

[78]:
```
def jacobian_RobertsonDiffusion(t, y, alpha=alpha, dLapl=dNM_Laplace):
    u = y[0:n]
    v = y[n:2*n]
    w = y[2*n:3*n]
    dut_du = -0.04 * np.identity(n) - alpha * dNM_Laplace
    dut_dv = 1e4 * np.diag(w)
    dut_dw = 1e4 * np.diag(v)
    dvt_du = 0.04 * np.identity(n)
    dvt_dv = -1e4 * np.diag(w) - 6 * 1e7 * np.diag(v) - alpha * dNM_Laplace
```

```
    dvt_dw = -1e4 * np.diag(v)
    dwt_du = np.zeros_like(dut_du)
    dwt_dv = 6 * 1e7 * np.diag(v)
    dwt_dw = -alpha * dNM_Laplace

    dut_duvw = np.concatenate((dut_du, dut_dv, dut_dw), axis=1)
    dvt_duvw = np.concatenate((dvt_du, dvt_dv, dvt_dw), axis=1)
    dwt_duvw = np.concatenate((dwt_du, dwt_dv, dwt_dw), axis=1)
    return np.concatenate((dut_duvw, dvt_duvw, dwt_duvw), axis=0)

def jacobian_RobertsonDiffusion_approx(t, y, alpha=alpha, dLapl=dNM_Laplace):
    u = y[0:n]
    v = y[n:2*n]
    w = y[2*n:3*n]
    dut_du = np.zeros((n, n))
    dut_dv = 1e4 * np.diag(w)
    dut_dw = 1e4 * np.diag(v)
    dvt_du = np.zeros_like(dut_du)
    dvt_dv = -1e4 * np.diag(w) - 6 * 1e7 * np.diag(v)
    dvt_dw = -1e4 * np.diag(v)
    dwt_du = np.zeros_like(dut_du)
    dwt_dv = 6 * 1e7 * np.diag(v)
    dwt_dw = np.zeros_like(dut_du)

    dut_duvw = np.concatenate((dut_du, dut_dv, dut_dw), axis=1)
    dvt_duvw = np.concatenate((dvt_du, dvt_dv, dvt_dw), axis=1)
    dwt_duvw = np.concatenate((dwt_du, dwt_dv, dwt_dw), axis=1)
    return np.concatenate((dut_duvw, dvt_duvw, dwt_duvw), axis=0)
```

Legen Sie die Anfangswerte in einem array `y0` an und lösen Sie anschließend die partielle Differentialgleichung mit dem Radau-Verfahren von Ordnung 5. Übergeben Sie die Jacobi-Matrix, welche Sie zuvor bestimmt haben, an das `jac`-Argument. Wie wirkt sich eine Approximation der Jacobimatrix (z.B. wenn Sie nur Terme mit Vorfaktor $> 10^4$ betrachten) auf die Lösung und den zugehörigen Aufwand aus. Testen Sie auch eine Lösung ohne explizite Angabe der Jacobi-matrix.

[79]:
```
t_span = (0,100)
y0 = np.concatenate(
    (a := 1 + np.sin(2 * np.pi * xs),
     np.zeros_like(a),
     np.zeros_like(a)),
    axis=0)
sol_Radau = scipy.integrate.solve_ivp(fun=rhs_RobertsonDiffusion,
                                      t_span=t_span, y0=y0,
                                      t_eval=np.
 ↪linspace(t_span[0],t_span[-1],21),
                                      method="Radau",
                                      jac=jacobian_RobertsonDiffusion)
```

```
print("sol_Radau: ", sol_Radau)
sol_Radau_jac_approx = scipy.integrate.solve_ivp(fun=rhs_RobertsonDiffusion,
                                        t_span=t_span, y0=y0,
                                        t_eval=np.
 ↪linspace(t_span[0],t_span[-1],21),
                                        method="Radau",
                                        jac=jacobian_RobertsonDiffusion_approx)
print("sol_Radau_jac_approx: ", sol_Radau_jac_approx)
sol_Radau_jac_fd = scipy.integrate.solve_ivp(fun=rhs_RobertsonDiffusion,
                                        t_span=t_span, y0=y0,
                                        t_eval=np.
 ↪linspace(t_span[0],t_span[-1],21),
                                        method="Radau",
                                        jac=None)
print("sol_Radau_jac_fd: ", sol_Radau_jac_fd)
```

```
sol_Radau:    message: The solver successfully reached the end of the
integration interval.
  success: True
   status: 0
        t: [ 0.000e+00  5.000e+00 …  9.500e+01  1.000e+02]
        y: [[ 1.000e+00  1.182e+00 …  6.232e-01  6.174e-01]
            [ 1.215e+00  1.179e+00 …  6.232e-01  6.174e-01]
            …
            [ 0.000e+00  7.859e-02 …  3.768e-01  3.826e-01]
            [ 0.000e+00  7.827e-02 …  3.768e-01  3.826e-01]]
      sol: None
 t_events: None
 y_events: None
     nfev: 308
     njev: 15
      nlu: 72
sol_Radau_jac_approx:    message: The solver successfully reached the end of the
integration interval.
  success: True
   status: 0
        t: [ 0.000e+00  5.000e+00 …  9.500e+01  1.000e+02]
        y: [[ 1.000e+00  1.182e+00 …  6.232e-01  6.174e-01]
            [ 1.215e+00  1.179e+00 …  6.232e-01  6.174e-01]
            …
            [ 0.000e+00  7.859e-02 …  3.768e-01  3.826e-01]
            [ 0.000e+00  7.826e-02 …  3.768e-01  3.826e-01]]
      sol: None
 t_events: None
 y_events: None
     nfev: 99790
     njev: 5700
```

```
      nlu: 28498
sol_Radau_jac_fd:    message: The solver successfully reached the end of the
integration interval.
  success: True
   status: 0
        t: [ 0.000e+00  5.000e+00 …  9.500e+01  1.000e+02]
        y: [[ 1.000e+00  1.182e+00 …  6.232e-01  6.174e-01]
            [ 1.215e+00  1.179e+00 …  6.232e-01  6.174e-01]
            …
            [ 0.000e+00  7.859e-02 …  3.768e-01  3.826e-01]
            [ 0.000e+00  7.826e-02 …  3.768e-01  3.826e-01]]
      sol: None
 t_events: None
 y_events: None
     nfev: 317
     njev: 14
      nlu: 74
```

<– Platz für Ihre Antwort –>

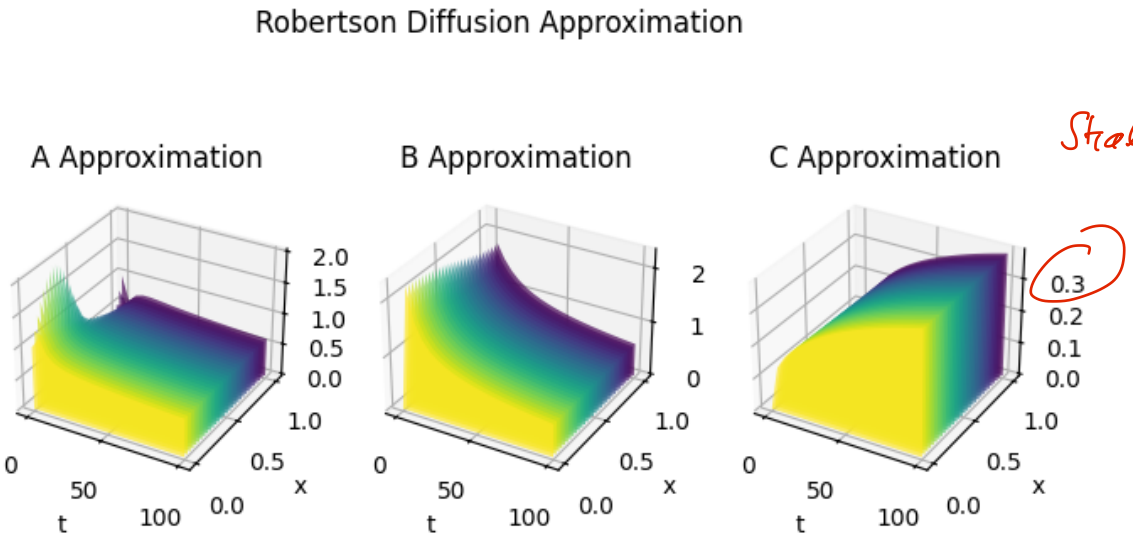
Die Approximation der Jacobi Matrix macht den Aufwand deutlich höher, wie zum Beispiel an `nfev` erkennbar, die Lösung sieht aber etwa gleich gut aus, siehe folgendes Diagramm

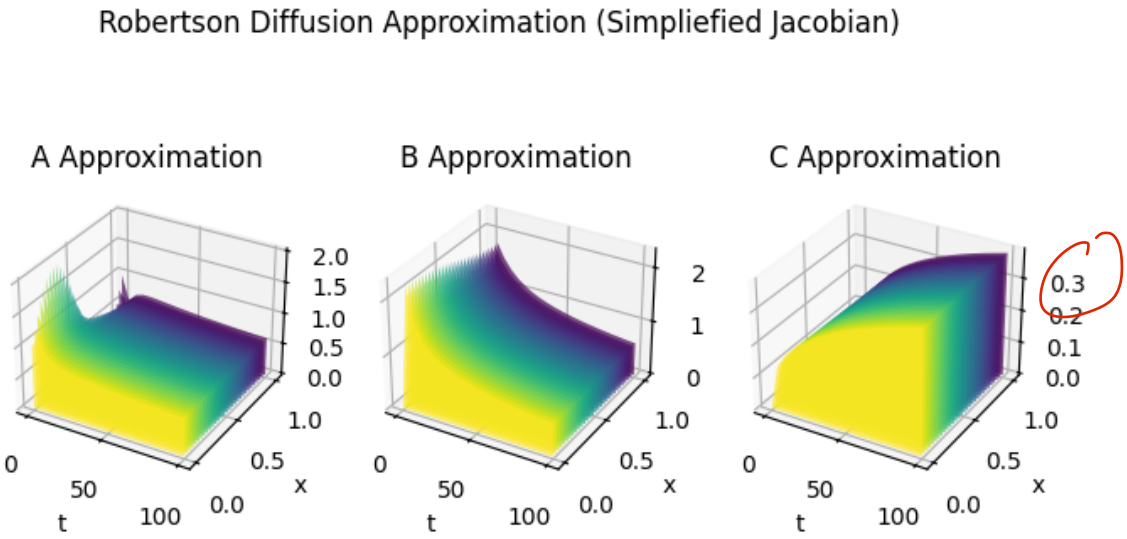
Die Approximation mit Finite Differenzen von der Jacobi Matrix macht den Aufwand nur geringfügig höher, wie zum Beispiel an `nfev` erkennbar, die Lösung sieht etwa gleich gut aus, siehe folgendes Diagramm

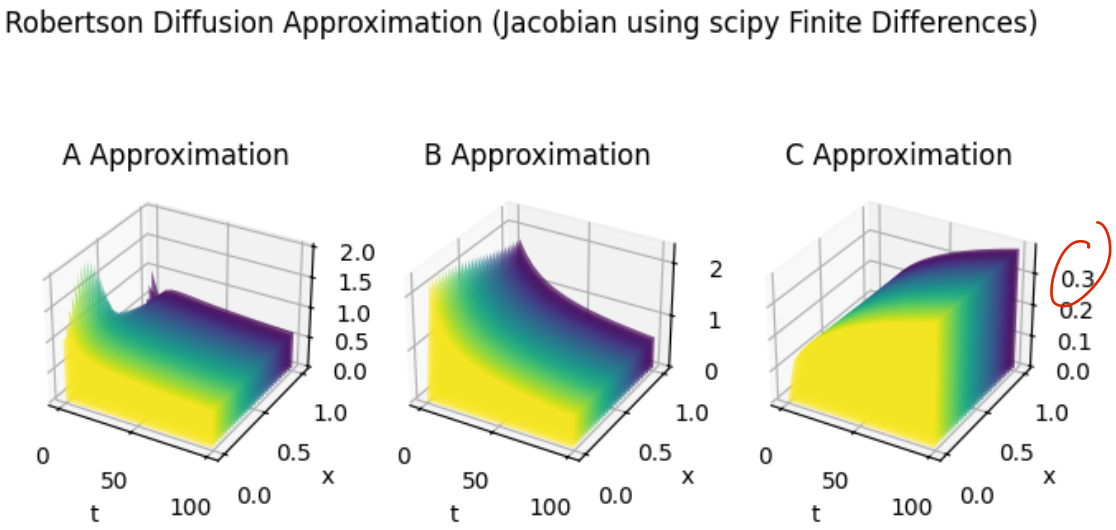
Visualisieren Sie die Lösen der Differentialgleichung komponentenweise. Dazu bietet sich zum Beispiel ein sogenannter "waterfall-plot" an, wie beispielsweise hier zu sehen. Skalieren Sie die zweite Komponente (Substanz B) wieder geeignet (z.B. mit $10^5$) und achten darauf die Komponenten mit den gleichen Achsenskalierungen darzustellen.

[80]:
```python
def plot_RobertsonDiffusion_solution(sol_Radau, title):
  u = sol_Radau.y[0:n,:]
  v = sol_Radau.y[n:2*n,:]
  w = sol_Radau.y[2*n:3*n,:]
  y = [u, v, w]
  ts = np.linspace(t_span[0],t_span[-1],21)

  # plot
  facecolors = plt.colormaps['viridis_r'](np.linspace(0, 1, n))
  titles = ["A Approximation", "B Approximation", "C Approximation"]
  z_labels = ["A", "1e5 * B", "C"]
  fig, axs = plt.subplots(1, 3, figsize=(8, 4), subplot_kw={"projection": "3d"})
  fig.suptitle(title)
  for i, ax in enumerate(axs):
    #ax = fig.add_subplot(projection="3d")
    ax.set_title(titles[i])
    ax.set_xlabel("t")
```

```
    ax.set_ylabel("x")
    ax.set_zlabel(z_labels[i])
    for j, x in enumerate(xs):
      ax.fill_between(ts, x, (1e5 if i == 1 else 1) * y[i][j], ts, x, 0,␣
 ↪facecolors=facecolors[j], alpha=0.7)
  plt.show()

plot_RobertsonDiffusion_solution(sol_Radau, "Robertson Diffusion Approximation")
plot_RobertsonDiffusion_solution(sol_Radau_jac_approx, "Robertson Diffusion␣
 ↪Approximation (Simpliefied Jacobian)")
plot_RobertsonDiffusion_solution(sol_Radau_jac_fd, "Robertson Diffusion␣
 ↪Approximation (Jacobian using scipy Finite Differences)")
```




Robertson Diffusion Approximation (Jacobian using scipy Finite Differences)
A Approximation
B Approximation
C Approximation
2.0
1.5
1.0
0.5
0.0
2
1
0
0.3
0.2
0.1
0.0
0
50
100
t
0.0
0.5
1.0
x