

ETHZ, D-MATH
Prüfung
Numerische Methoden D-PHYS, WS 2016/17
Dr. V. Gradinaru
02.02.2017

1 Aufgabe: 6 Punkte

Die Lösung des nicht-linearen Gleichungssystems

$$\begin{aligned}x_1 &= \frac{1}{4}(\cos(x_1) - \sin(x_2)) \\x_2 &= \frac{1}{4}(\cos(x_1) - 3\sin(x_2))\end{aligned}$$

soll im Gebiet $[0, 1]^2$ mit Hilfe des Newton-Verfahrens bestimmt werden.
Arbeiten Sie in den folgenden Teilaufgaben mit dem Template `newton.py`.

- a) (1 punkte) Schreiben Sie auf papier eine Funktion $\mathbf{F}(x_1, x_2)$, welche als Nullstelle die Lösung des obigen nicht-linearen Gleichungssystems besitzt.
Schreiben Sie auf papier weiter die Jacobi Matrix sowie deren Inverse Ihrer Funktion \mathbf{F} .

Solution:

(+.5 for writing F and dF, +.5 for writing dFinverse) Die Jacobi Matrix Determinante ist

$$\det(\mathbf{F}') = 1 + \frac{1}{4}\sin(x_1) + \frac{3}{4}\cos(x_2) + \frac{1}{8}\sin(x_1)\cos(x_2) > 0 \forall x_1, x_2 \in \mathbb{R}.$$

Die Inverse der Jacobi Matrix ist dann einfach

$$\mathbf{F}'^{-1}(x_1, x_2) = \frac{1}{\det(\mathbf{F}')} \begin{pmatrix} 1 + \frac{3}{4}\cos(x_2) & -\frac{1}{4}\cos(x_2) \\ -\frac{1}{4}\sin(x_1) & 1 + \frac{1}{4}\sin(x_1) \end{pmatrix}.$$

- b) (4 punkte) Implementieren Sie das Newton-Verfahren in der Funktion `Newton(F, dFInv, x0, tol, maxit)` innerhalb von `newton.py`.

Solution:

(+.5 for correct implementation of F, +.5 for correct implementation of dF, +1 for implementing dFinverse, +1 correct iteration scheme, +1 correct stopping criteria)

- c) (1 punkte) Verwenden Sie Ihre Funktion `Newton` und einem geeigneten Startwert um die Lösung des Gleichungssystems zu bestimmen. Benutzen Sie als Abbruchkriterien `tol = 10-13` und `maxit = 100`.

Drucken Sie die fünf ersten Iterationswerte.

Solution:

Listing 1.1: newton_Solution.py

```
1 import numpy as np
2
3 def F(x):
4     """
5     Funktion deren Nullstelle gesucht wird
6
7     Input: x ... Punkt an welchem die Funktion F ausgewertet soll
8
9     Output: F ... Funktionswert am Punkt x
10
11     """
12
13     # Werte Funktion F aus am Punkt x
14     F = np.zeros(2)
15     F[0] = x[0] - 0.25*(np.cos(x[0]) - np.sin(x[1]))
16     F[1] = x[1] - 0.25*(np.cos(x[0]) - 3.*np.sin(x[1]))
17
18     # Rueckgabe
19     return F
20
21 def dF(x):
22     """
23     Jacobi Matrix der Funktion F
24
25     Input: x ... Punkt an welchem die Jacobi Matrix der Funktion F
26            ausgewertet soll
27
28     Output: dF ... Jacobi Matrix von F am Punkt x
29
30     """
31
32     # Werte Jacobie Matrix der Funktion F aus am Punkt x
33     dF = np.zeros((2,2))
34     dF[0,0] = 1 + 0.25*np.sin(x[0])
35     dF[0,1] = 0.25*np.cos(x[1])
36     dF[1,0] = 0.25*np.sin(x[0])
37     dF[1,1] = 1 + 0.50*np.cos(x[1])
38
39     # Rueckgabe
40     return dF
41
42 def dFInv(x):
43     """
44     Inverse der Jacobi Matrix der Funktion F
45
46     Input: x ... Punkt an welchem die Inverse der Jacobi Matrix der
47            Funktion F ausgewertet soll
48
49     Output: dFInv ... Jacobi Matrix von F am Punkt x
50
51     """
52
53     # Werte Inverse der Jacobie Matrix der Funktion F aus am Punkt x
54     dFInv = np.zeros((2,2))
55     det_dF = 1. + 0.25*np.sin(x[0]) + 0.50*np.cos(x[1]) \
56              + 1./16.*np.sin(x[0])*np.cos(x[1])
57     dFInv[0,0] = 1 + 3/4*np.cos(x[1])
58     dFInv[0,1] = - 0.25*np.cos(x[1])
59     dFInv[1,0] = - 0.25*np.sin(x[0])
60     dFInv[1,1] = 1 + 0.25*np.sin(x[0])
61     dFInv = dFInv/det_dF
62
63
64
65
66
```

```

67     # Rueckgabe
68     return dFInv
69
70 def Newton(F,dFInv,x0,tol,maxit):
71
72     """
73     Newton-Verfahren
74
75     Input: F      ... Funktion des Nullstelle gesucht wird
76            dFInv ... Inverse der Jacobi Matrix von F
77            x0    ... Startwert
78            tol   ... Genauigkeit
79            maxit ... maximale Anzahl Iterationen
80
81     Output: x      ... Vektor mit allen Iterationswerten (inklusive x0)
82            Converged ... Konvergenzflag (Converged = True falls konvergiert
83                                     Converged = False sonst)
84            i       ... Anzahl benoetigter Iterationen bis zur Genauigkeit
85                    tol
86
87     """
88
89     x = np.zeros((len(x0),maxit)) # initialisiere Iterations-Sequence
90                                     # mit Nulls
91     Converged = False             # Konvergenz Flag
92     x[:,0] = x0                   # Startwert
93     i = 0                          # initialisiere Iterations-Zaehler
94     while ( i < maxit - 1 ):
95         # Iterations-Schritt
96         x[:,i+1] = x[:,i] - np.dot(dFInv(x[:,i]),F(x[:,i]))
97         # Untersuche auf Konvergenz
98         if ( np.linalg.norm(x[:,i+1]-x[:,i]) < tol ):
99             Converged = True
100            i = i + 1
101            break
102            # erhaeche Iterations Zaehler
103            i = i + 1
104
105     # Rueckgabe
106     return x,Converged,i
107
108 # Bestimme die Nullstelle mit dem Newton-Verfahren
109 x0 = np.array([0.,0.]) # Startwert
110 tol = 1.e-13          # Toleranz
111 maxit = 100           # Maximale Anzahl Iterationen
112 [x,Converged,it] = Newton(F,dFInv,x0,tol,maxit)
113 print 'Iterationen i=1,...,5:',x[:,0:5]
114 print 'Die Nullstelle ist:',x[:,it]

```

2 Aufgabe: 6 Punkte

Seien $\mathbf{z}, \mathbf{c} \in \mathbb{R}^n$, Vektoren von gemessenen Daten. Die zwei reellen Zahlen α^* und β^* seien definiert als

$$(\alpha^*, \beta^*) = \min_{\alpha, \beta \in \mathbb{R}} \|\mathbf{T}_{\alpha, \beta} \mathbf{z} - \mathbf{c}\|_2, \quad (2.1)$$

wobei $\mathbf{T}_{\alpha, \beta}$ eine tridiagonale Matrix

$$\mathbf{T}_{\alpha, \beta} = \begin{pmatrix} \alpha & \beta & 0 & \cdots & 0 \\ \beta & \alpha & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \alpha & \beta \\ 0 & \cdots & 0 & \beta & \alpha \end{pmatrix} \in \mathbb{R}^{n, n} \quad (2.2)$$

ist.

- a) (4 punkte) Reformulieren Sie dieses Problem als lineares Ausgleichsrechnung-Problem der Form

$$\mathbf{x}^* = \min_{\mathbf{x} \in \mathbb{R}^k} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2$$

mit passenden $\mathbf{A} \in \mathbb{R}^{m, k}$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{x} \in \mathbb{R}^k$, $k, m \in \mathbb{N}$. Schreiben Sie hier $k, m, \mathbf{A}, \mathbf{b}, \mathbf{x}$:

$k =$

$m =$

$\mathbf{A} =$

$\mathbf{b} =$

$\mathbf{x} =$

Solution:

$k = 2$ (+.5 punkte)

$m = n$ (+.5 punkte)

$$A = \begin{pmatrix} z_1 & z_2 \\ z_2 & z_1 + z_3 \\ \vdots & \vdots \\ z_{n-1} & z_{n-2} + z_n \\ z_n & z_{n-1} \end{pmatrix} \quad (+2 \text{ punkte})$$

$\mathbf{b} = \mathbf{c}$ (+.5 punkte)

$$\mathbf{x} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad (+.5 \text{ punkte})$$

- b) (2 Punkte) Bestimmen Sie α^* und β^* für die in `leastqr.py` definierten Vektoren \mathbf{z} und \mathbf{c} .

Hinweis: verwenden Sie `linalg.lstsq` aus `numpy`.

Solution:

(+1 correct implementation of A , +1 correct usage of `lstsq`).

Listing 2.1: `leastqr_Solution.py`

```
1 import numpy as np
2
3 # Vektoren z & c
4 n = 10
5 z = np.arange(1,n+1)
6 c = np.arange(n,0,-1)
7
8 # Bilde Matrix A
9 A = np.zeros((n,2))
10 A[0,:] = np.array([z[0],z[1]])
11 for i in np.arange(1,n-1):
12     A[i,0] = z[i]
13     A[i,1] = z[i-1] + z[i+1]
14 A[-1,:] = np.array([z[-1],z[-2]])
15
16 # Loese Ausgleichsproblem
17 [x,res,rnk,s] = np.linalg.lstsq(A,c)
18 a = x[0]
19 b = x[1]
20
21 # Gebe die Loesung aus
22 print 'alpha_□=□', a
23 print 'beta_□=□', b
```

Listing 2.2: `leastqr_Solution.out`

```
1 alpha = -0.421052631579
2 beta = 0.578947368421
```

3 Aufgabe: 14 Punkte

Wir betrachten N Körper im dreidimensionalen Raum. Ihre Dynamik unterliegt einzig der Gravitation. Jeder Körper K_i hat eine Position $\vec{q}_i \in \mathbb{R}^3$ und einen Impuls $\vec{p}_i \in \mathbb{R}^3$ sowie eine Masse m_i . Wir fassen deren Positionen \vec{q}_i und Impulse \vec{p}_i in Vektoren zusammen:

$$\begin{aligned}\vec{q} &= [\vec{q}_1 \dots \vec{q}_N]^T \in \mathbb{R}^{3N} \\ \vec{p} &= [\vec{p}_1 \dots \vec{p}_N]^T \in \mathbb{R}^{3N}.\end{aligned}$$

Die Hamilton-Funktion $\mathcal{H}(\vec{q}, \vec{p})$ lautet dann:

$$\mathcal{H}(\vec{q}, \vec{p}) = \frac{1}{2} \sum_{i=1}^N \frac{1}{m_i} \vec{p}_i^T \vec{p}_i - G \sum_{1 \leq i < j \leq N} \frac{m_i m_j}{\|\vec{q}_j - \vec{q}_i\|}.$$

Schliesslich fasst man noch die Positionen \vec{q} und Impulse \vec{p} in einen einzigen grossen Vektor $\vec{y} = [\vec{q} \mid \vec{p}]^T$ zusammen. Die Bewegungsgleichungen ist gegeben bei ein System von gekoppelten Differentialgleichungen erster Ordnung:

$$\dot{\vec{y}}(t) = \begin{bmatrix} \dot{\vec{q}}(t) \\ \dot{\vec{p}}(t) \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathcal{H}}{\partial \vec{p}} \\ \frac{\partial \mathcal{H}}{\partial \vec{q}} \end{bmatrix} = f(t, \vec{y})$$

mit $f : \mathbb{R}^+ \times \mathbb{R}^{6N} \rightarrow \mathbb{R}^{6N}$ welches wir nun lösen wollen.

- a) (5 punkte) Implementieren Sie in `nbody.py` mithilfe obiger Formeln die Funktion $f(\vec{y})$ korrekt für N Körper.

Hinweis:

$$\begin{aligned}\frac{\partial \mathcal{H}}{\partial \vec{p}_i} &= \frac{\vec{p}_i}{m_i}, \\ \frac{\partial \mathcal{H}}{\partial \vec{q}_i} &= G \sum_{j \neq i} m_i m_j \frac{\vec{q}_i - \vec{q}_j}{\|\vec{q}_i - \vec{q}_j\|^3}.\end{aligned}$$

Solution:

(+2 for correct kinetic term, +3 for correct potential term)

- b) Implementieren Sie:

- (2 punkte) das explizite Eulerverfahren,
- (2 punkte) das implizite Eulerverfahren,
- (2 punkte) die implizite Mittelpunktsregel.

Hinweis: Benutzen Sie die Funktion `fsolve` aus `scipy.optimize`.

- c) (3 Punkte) Implementieren Sie das velocity-Verlet Verfahren und eine kompatible Version der rechten Seite f . Achten Sie dabei auf die Verwendung korrekter Input-Werte.

Solution:

(+1 for correct rhs, +2 for correct velocity-verlet method)

Ihre Implementierung wird für zwei Körper mit den in der Vorlage angegebenen Anfangsbedingungen getestet.

Solution:

Listing 3.1: nbdoxy

```
1  #!/usr/bin/env python
2  from numpy import *
3  from numpy.linalg import *
4  import scipy.optimize
5  from matplotlib.pyplot import *
6  from time import time
7
8
9  # Unteraufgabe a)
10
11 def F(q, p, m, D=3):
12     """The function f(y)
13
14     Input: q ... array with positions
15           p ... array with momenta
16           m ... array with masses
17           D ... space dimension, default equals 3.
18
19     Output: dq ... time-derivative of the positions
20            dp ... time-derivative of the momenta
21     """
22     N = q.size // D
23
24     dq = zeros_like(p)
25     for k in xrange(N):
26         mk = m[k]
27         pk = p[D*k:D*(k+1)]
28         dq[D*k:D*(k+1)] = 1.0/mk * pk
29
30     dp = zeros_like(q)
31     for k in xrange(N):
32         mk = m[k]
33         qk = q[D*k:D*(k+1)]
34         dpk = zeros_like(qk)
35         for i in xrange(N):
36             if i != k:
37                 mi = m[i]
38                 qi = q[D*i:D*(i+1)]
39                 no = norm(qi - qk)
40                 dpk += mk*mi/no**3 * (qi - qk)
41         dp[D*k:D*(k+1)] = G * dpk
42
43     return dq, dp
44
45
46 def rhs(y):
47     """Right hand side
48
49     Input: y ... array of q and p: y = (q, p)
50
51     Output: dy ... time-derivative of y
52     """
53     q, p = hsplit(y, 2)
54     dy = hstack(F(q, p, m))
55     return dy
```

```

56
57
58
59 # Unteraufgabe b)
60
61 def integrate_EE(y0, xStart, xEnd, steps, flag=False):
62     r"""Integrate ODE with explicit Euler method
63
64     Input: y0...initial condition
65            xStart...start x
66            xEnd...end x
67            steps...number of steps (h=(xEnd-xStart)/N)
68            flag==False return complete solution: (phi, phi', t)
69            flag==True return solution at endtime only: phi(tEnd)
70
71     Output: x...variable
72            y...solution
73     """
74     h = (xEnd - xStart) / (1.0*steps)
75     x = zeros(steps+1)
76     y = zeros((steps+1, size(y0)))
77
78     x[0] = xStart
79     y[0, :] = y0.reshape(size(y0),)
80     for i in xrange(steps):
81         x[i+1] = xStart + (i + 1)*h
82         y[i+1, :] = y[i, :] + h*rhs(y[i, :])
83
84     if flag:
85         return x[-1], y[-1][:]
86     else:
87         return x, y
88
89
90 def integrate_IE(y0, xStart, xEnd, steps, flag=False):
91     r"""Integrate ODE with implicit Euler method
92
93     Input: y0...initial condition
94            xStart...start x
95            xEnd...end x
96            steps...number of steps (h=(xEnd-xStart)/N)
97            flag==False return complete solution: (phi, phi', t)
98            flag==True return solution at endtime only: phi(tEnd)
99
100    Output: x...variable
101            y...solution
102    """
103    h = (xEnd - xStart) / (1.0*steps)
104    x = zeros(steps+1)
105    y = zeros((steps+1, size(y0)))
106
107    x[0] = xStart
108    y[0, :] = y0
109    for i in xrange(steps):
110        x[i+1] = xStart + (i + 1)*h
111        F = lambda yph: yph - y[i, :] - h*rhs(yph)
112        y[i+1, :] = scipy.optimize.fsolve(F, y[i, :]+h*rhs(y[i, :]))
113
114    if flag:
115        return x[-1], y[-1][:]
116    else:
117        return x, y
118
119
120 def integrate_IM(y0, xStart, xEnd, steps, flag=False):
121     r"""Integrate ODE with implicit midpoint rule
122
123     Input: y0...initial condition
124            xStart...start x
125            xEnd...end x
126            steps...number of steps (h=(xEnd-xStart)/N)

```

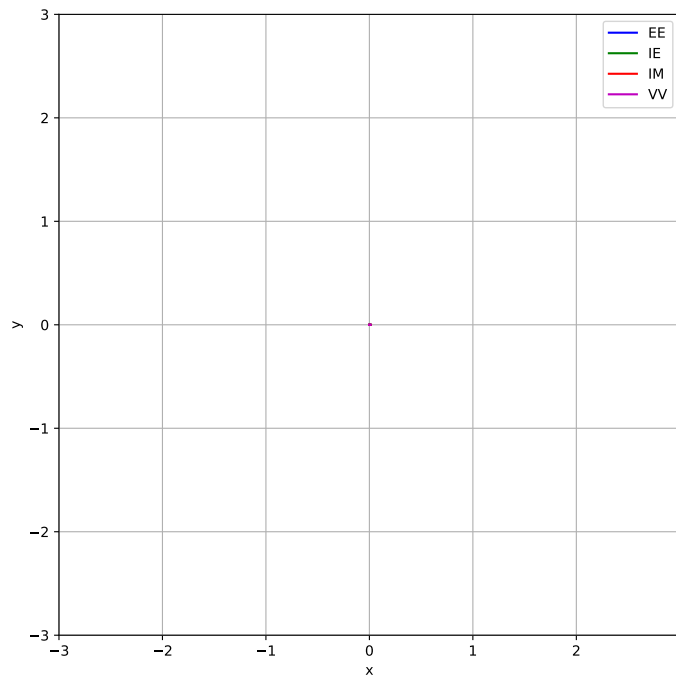


```

198     rz = rhs_vv(z)
199     z1 = z + h*v+0.5*h*h*rz
200     v1 = v + 0.5*h*(rz + rhs_vv(z1))
201     y[k+1,:] = 1.*hstack((z1,v1))
202     z = 1.0*z1
203     v = 1.0*v1
204     t[k+1] = k*h
205
206     if flag:
207         return x[-1], y[-1][:]
208     else:
209         return x, y
210
211
212 # Tests for two bodies
213 G = 1.0
214 m = array([500.0, 1.0])
215 q0 = hstack([0,0,0, 2,0,0])
216 p0 = hstack([0,0,0, 0,sqrt(m[0]/q0[3]),0])
217 y0 = hstack([q0, p0])
218
219 # Compute
220 T = 3
221 nrsteps = 5000
222
223 t_ee, y_ee = integrate_EE(y0, 0, T, nrsteps, False)
224 t_ie, y_ie = integrate_IE(y0, 0, T, nrsteps, False)
225 t_im, y_im = integrate_IM(y0, 0, T, nrsteps, False)
226 t_vv, y_vv = integrate_VV(y0, 0, T, nrsteps, False)
227
228 # Plot
229 fig = figure(figsize=(12,8))
230 ax = fig.gca()
231 ax.set_aspect("equal")
232 ax.plot(y_ee[:,0], y_ee[:,1], "b-")
233 ax.plot(y_ee[:,3], y_ee[:,4], "b-", label="EE")
234
235 ax.plot(y_ie[:,0], y_ie[:,1], "g-")
236 ax.plot(y_ie[:,3], y_ie[:,4], "g-", label="IE")
237
238 ax.plot(y_im[:,0], y_im[:,1], "r-")
239 ax.plot(y_im[:,3], y_im[:,4], "r-", label="IM")
240
241 ax.plot(y_vv[:,0], y_vv[:,1], "m-")
242 ax.plot(y_vv[:,3], y_vv[:,4], "m-", label="VV")
243
244 ax.grid(True)
245 ax.set_xlim(-3,3)
246 ax.set_ylim(-3,3)
247 ax.legend(loc="upper_right")
248 ax.set_xlabel("x")
249 ax.set_ylabel("y")
250 fig.savefig("zwei.pdf")

```

Solution:



4 Aufgabe: 6 Punkte

Wir betrachten die Funktion

$$\int_{\vec{x} \in [0,1]^d} |x_1 + \dots + x_d|^2 d\vec{x},$$

- a) (1 punkt) Implementieren Sie in `MonteCarlo.py` eine Python-Funktion `mcquad(d,k)`, die das obige Integral mit der Monte-Carlo-Methode mit $N = 10^k$ Zufallsvektoren numerisch berechnet.
- b) (3 punkte) Verwenden Sie diese Funktion mit $k = 4$ und $d = 10$ für $M = 100$ verschiedene Experimente. Bestimmen Sie Mittelwert und

$$\tilde{\sigma}_M = \sqrt{\frac{\frac{1}{M} \sum_{i=1}^M z(t_i)^2 - \left(\frac{1}{M} \sum_{i=1}^M z(t_i)\right)^2}{M-1}}.$$

Welche Aussage lässt sich daraus über die Genauigkeit der Approximation ableiten?

Solution:

(+1 for Mittelwert, +1 for sigma, +1 for it's the 68% confidence interval for the approximate integral value")

- c) (2 punkte) Bestimmen und printen Sie das Vertrauensintervall für $d = 2, \dots, 10$ und $k = 4$ und $M = 100$.

Solution:

(+2 for correct Vertrauensintervall)

Solution:

Listing 4.1: Multidimensionale Monte-Carlo Integration.

```
1 from numpy import array, sqrt, sum
2 from numpy.random import rand
3
4 func = lambda x: sum(x, axis=1)**2
5
6 # Unteraufgabe a)
7
8 def mcquad(d, k):
9     "Argumente d und k wie in der Aufgabenstellung"
10    N = 10 ** k
11    # Sample
12    x = rand(N, d)
13    x = func(x)
14    # Integral
15    I = sum(x) / N
16    return I
17
18
19 # Unteraufgabe b)
20 k = 4
21 d = 10
22 M = 100
23 exval = d/3.0 + 0.25*d*(d - 1)
24 results = []
25
26 for m in xrange(M):
27     ex = mcquad(d, k)
28     results.append(ex)
29 ex = array(results)
30 ev = sum(ex, axis=0) / M
31 sgm = sqrt( (sum(ex**2)/M - ev**2) / (M-1.) )
32
33 print("Dimension: %d" % d)
34 print('Exakt Wert: %f' % exval)
35 print('Mittelwert: %f' % ev)
36 print('Sigma: %f' % sgm)
37 print('-'*16)
38
39
40 # Unteraufgabe c)
41 k = 4
42 M = 100
43
44 for d in xrange(2, 11):
45     print("Dimension: %d" % d)
46     exval = d/3.0 + 0.25*d*(d - 1)
47     print('Exakt Wert: %f' % exval)
48     results = []
49
50     for m in xrange(M):
51         ex = mcquad(d, k)
52         results.append(ex)
53
54     ex = array(results)
55     ev = sum(ex, axis=0) / M
56     sgm = sqrt( (sum(ex**2)/M - ev**2) / (M-1.) )
57     a = ev - sgm
58     b = ev + sgm
59     print('Vertrauensintervall: [%f, %f]' % (a, b))
60     print('-'*16)
```

5 Aufgabe: 16 Punkte

- a) (2 Punkte) Implementieren Sie in `RungeKutta.py` ein Runge-Kutta-Schritt mit dem Butcher Tableau der so genannten *England Formel*:

$$\begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & \\ 1 & 0 & -1 & 2 \\ \hline & \frac{1}{6} & 0 & \frac{2}{3} & \frac{1}{6} \end{array}$$

- b) (3 Punkte) Finden Sie empirisch die Konvergenzordnung dieses Verfahrens indem Sie es für die Lösung der Gleichung $\ddot{y} = -y(t)$ auf $[0, 10]$ verwenden.

Solution:

(of these points, 1 for saying it is order 4).

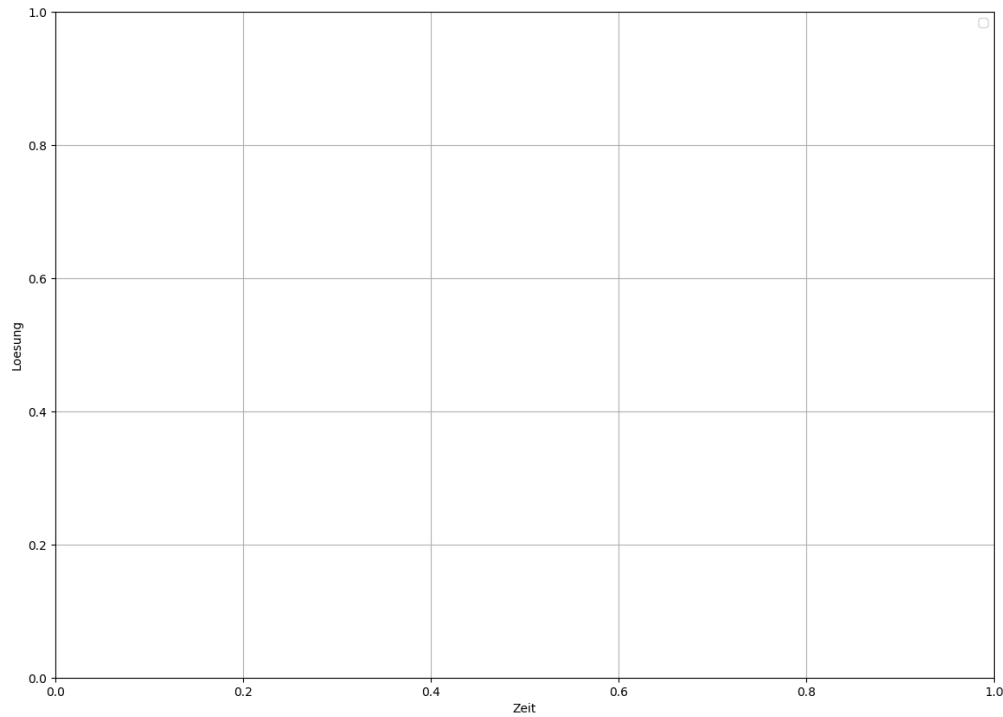


- c) (2 Punkte) Verwenden Sie diese Methode, um die Gleichung

$$\begin{cases} \ddot{y} = 10025 - 10025y - 10\dot{y} \\ y(0) = 2 \\ \dot{y}(0) = 95 \end{cases} \quad (5.1)$$

auf $[0, 1]$ mit 2^{12} Zeitschritten zu lösen. Plotten Sie die numerische Lösung.

Solution:
2 Punkte.



- d) (2 Punkte) Kann man mit dieser Methode in 5 Zeitschritten eine glaubwürdige Annäherung der Lösung der letzten Gleichung zur Zeit $T = 1$ erhalten? Warum?

Solution:

2 Punkt fuer "Nein, Stabilitaetsgebiet/Problem".

- e) (7 Punkte) Implementieren Sie die Stabilitätsfunktion $S : \mathbb{C} \rightarrow \mathbb{C}$. Das Template verwendet dieses $S(z)$ um den Stabilitätsbereich der Runge-Kutta-Methode zu plotten. Bestimmen Sie durch Ablesen aus diesem Bild eine maximale Zeitschrittweite h , die eine stabile Lösung für letztgenannte Differentialgleichung garantiert. Notieren Sie die Herleitung und den Wert auf Papier.

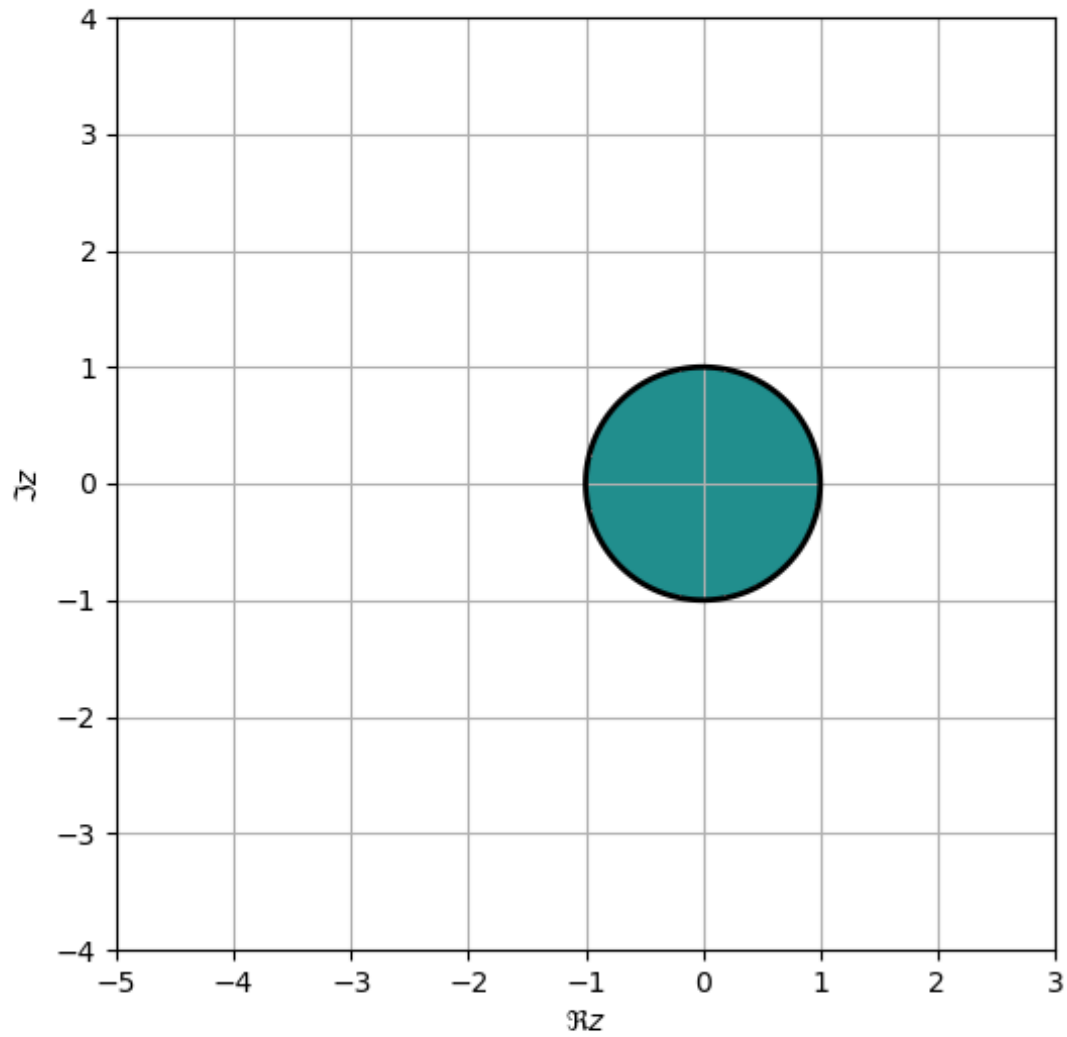
Solution:

(+2 for theoretical derivation of S, +1 for correct implementation of S in Python, +1 for correct linearized system, +1 for saying λ is one of the eigenvalues, +1 for correct value for λ , +1 for $h \simeq 0.0275$).

The matrix corresponding to the first order linearized system is

$$\begin{pmatrix} 0 & 1 \\ -10025 & -10 \end{pmatrix}$$

whose eigenvalues can be computed to be $\lambda = -5 \pm 100i$. Their norm is 100.125, thus (looking at the picture) the biggest h so that $|S(\lambda h)| < 1$ is approx. $2.75/100.125 \simeq 0.0275$.



Solution:

Listing 5.1: Runge-Kutta Verfahren

```
1  #!/usr/bin/env python
2  from numpy import *
3  from matplotlib.pyplot import *
4
5  # England Formel
6  bs_A = array([[0.0, 0.0, 0.0, 0.0],
7               [2.0, 0.0, 0.0, 0.0],
8               [1.0, 1.0, 0.0, 0.0],
9               [0.0, -4.0, 8.0, 0.0]]) / 4.0
10
11 bs_b = array([1.0, 0.0, 4.0, 1.0]) / 6.0
12
13 bs_c = array([0.0, 0.5, 0.5, 1.0])
14
15
16 # ex. a
17
18 def RK_step(rhs, A, b, c, t, y, h):
19     """
20     Makes a single Runge-Kutta step of size h.
21
22     Input:
23     rhs: right hand side of considered differential eq.
24     A, b, c: Butcher Schema
25     t: Current time t_i
26     y: Current solution y_i
27     h: Timestep size
28     """
29     tnew = t
30     ynew = y
31     s = A.shape[0]
32     k = zeros((2, s))
33
34     for i in xrange(s):
35         k[:,i] = rhs(t + c[i]*h, y + h*dot(A[i,:].T, k.T))
36
37     tnew = t + h
38     ynew = y + h * squeeze(dot(k, b))
39     return tnew, ynew
40
41
42 def RK(rhs, A, b, c, y0, tstart, tend, N):
43     """
44     Integrate the equation with a Runge-Kutta.
45
46     Input:
47     rhs: right hand side of considered differential eq.
48     A, b, c: Butcher Schema
49     y0: initial condition
50     tstart, tend: start, end time
51     N: number of steps
52     """
53     y = zeros((2, N+1))
54     t = zeros((N+1,))
55     t[0] = tstart
56     y[:,0] = y0
57     h = (tend - tstart) / float(N)
58     for n in xrange(N):
59         t[n+1], y[:,n+1] = RK_step(rhs, A, b, c, t[n], y[:,n], h)
60     return t, y
61
62
63 # ex b
64
65 IV = array([0.0, 1.0])
66 T = 10.0
```



```

67
68 Ns = 2**arange(2, 12)
69 hs = T / Ns
70 listErrors = []
71
72 rhs = lambda t, y: array([y[1], -y[0]])
73
74 for N in Ns:
75     t, y = RK(rhs, bs_A, bs_b, bs_c, IV, 0.0, T, N)
76     err = max(abs(y[0,:] - sin(t)))
77     listErrors.append(err)
78
79 k = 4 # Konvergenzordnung
80
81 figure(figsize=(14, 10))
82 loglog(hs, listErrors, 'x-', label='Fehler')
83 loglog(hs, hs**k, label='$h^{:d}$'.format(int(k)))
84 xlabel('Fehler')
85 xlabel('Schrittweite')
86 legend()
87 grid(True)
88 savefig('RK_EnglandFehler.png')
89
90
91 # ex c
92
93 IV = array([2.0, 95.0])
94 T = 1.0
95
96 rhs = lambda t, y: array([y[1],
97                          10025 - 10025*y[0] - 10.0*y[1]]) # === 0.5 Punkt
98
99 tex, yex = RK(rhs, bs_A, bs_b, bs_c, IV, 0.0, T, 2**12)
100 yex = yex[0,:]
101
102 t, y = RK(rhs, bs_A, bs_b, bs_c, IV, 0.0, T, 5) # === 0.5 Punkt
103 y = y[0,:]
104
105 figure(figsize=(14, 10))
106 plot(t, y, label='N=5') # === 0.5 Punkt
107 plot(tex, yex, label='N=2^{12}')
108 legend()
109 xlabel('Zeit')
110 ylabel('Loesung')
111 xlim(0.0, T)
112 grid(True)
113 savefig('RK_EnglandLoesung.png')
114
115
116 # ex e
117
118 from numpy.linalg import det
119
120
121 def S(z):
122     Id = identity(4)
123     oneT = ones(4)
124     num = det(Id - z*bs_A + z*outer(oneT, bs_b.T))
125     den = det(Id - z*bs_A)
126     result = num / den
127     return result
128
129
130 x1 = -5.0
131 xr = 3.0
132 y1 = -4.0
133 yr = 4.0
134 xd = linspace(x1, xr, 200)
135 yd = linspace(y1, yr, 200)
136 X, Y = meshgrid(xd, yd)
137 Z = X + 1j * Y

```

```
138 Sz = array(map(S, Z.flat)).reshape(Z.shape)
139
140 figure(figsize=(6, 6))
141 contourf(X, Y, where(abs(Sz) <= 1.0, 1.0, -inf), levels=linspace(0, 2, 51))
142 contour(X, Y, abs(Sz), levels=[1.0], colors='k', linewidths=[2])
143 grid(True)
144 xlim(-5, 3)
145 ylim(-4, 4)
146 xlabel(r'$\text{Re}_z$')
147 ylabel(r'$\text{Im}_z$')
148 savefig('RK_EnglandStabilityDomain.png')
```