D-MATH
Dr. Vasile Gradinaru
Alexander Dabrowski

Serie 7

Best Before: 3.5/4.5, in den Übungsgruppen (1 woche)

Koordinatoren: Alexander Dabrowski, HG G 52.1, `alexander.dabrowski@sam.math.ethz.ch`

Webpage: http://www.math.ethz.ch/education/bachelor/lectures/fs2016/other/nm_pc

Version: 1.0

1. *Gram-Schmidt-Verfahren und Householder Transformation*

In der Vorlesung haben wir die Householder Transformation verwendet um die **QR**-*Zerlegung* einer Matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ zu bestimmen. Ein weiteres, sehr intuitives Verfahren, das sukzessive die Spalten $\underline{a}_1, \ldots, \underline{a}_n$ mit $\underline{a}_i \in \mathbb{R}^m$, von $\mathbf{A}$ orthogonalisiert, ist das Gram-Schmidt-Verfahren. Das Gram-Schmidt-Verfahren ist ein Standardwerkzeug in Beweisen der Linearen Algebra. Die folgenden Algorithmen (in Pseudo-Code) liefern eine **QR**-Zerlegung nach dem *Gram-Schmidt-Verfahren* und dem *modifizierten Gram-Schmidt-Verfahren*:

Gram-Schmidt:

for $j = 1, \ldots, n$ **do**
  $\mathbf{v_j} = \mathbf{A}_{:j}$
  for $i = 1, \ldots, j-1$ **do**
    $\mathbf{R}_{ij} = \mathbf{q_i}^{\mathrm{T}}\mathbf{a_j}$
    $\mathbf{v_j} = \mathbf{v_j} - \mathbf{R}_{ij}\mathbf{q_i}$
  end for
  $\mathbf{R}_{jj} = \|\mathbf{v_j}\|_2$
  $\mathbf{Q}_{:j} = \frac{\mathbf{v_j}}{\mathbf{R}_{jj}}$
end for

Modifiziertes Gram-Schmidt:

for $i = 1, \ldots, n$ **do**
  $\mathbf{V}_{:i} = \mathbf{A}_{:i}$
end for
for $i = 1, \ldots, n$ **do**
  $\mathbf{R}_{ii} = \|\mathbf{v_i}\|_2$
  $\mathbf{q_i} = \frac{\mathbf{v_i}}{\mathbf{R}_{ii}}$
  for $j = i+1, \ldots, n$ **do**
    $\mathbf{R}_{ij} = \mathbf{q_i}^{\mathrm{T}}\mathbf{v_j}$
    $\mathbf{v_j} = \mathbf{v_j} - \mathbf{R}_{ij}\mathbf{q_i}$
  end for
end for

a) Implementieren Sie die beiden Gram-Schmidt-Verfahren in `ortho_Template.py` und verwenden Sie beide Verfahren, um die **QR**-Zerlegung der Matrix $\mathbf{Z} \in \mathbb{R}^{50\times 50}$ mit den Einträgen:

$$\mathbf{Z}_{ij} = 1 + \min(i,j), \quad 0 \le i, j < 50$$

zu bestimmen.

b) Vergleichen Sie die Güte der beiden Gram-Schmidt-Verfahren in Bezug auf die Orthogonalität der Spalten von **Q**.

c) Warum sind die Gram-Schmidt-Verfahren im Gegensatz zur Householder-Transformation (siehe `ortho_Template.py`) ungeeignete numerische Methoden zur Berechnung von **QR**-Zerlegungen?

Solution:
Die Implementierung ist in Listing 1 gezeigt.

Listing 1: Aufgabe 1):

```
from matplotlib.pylab import *
from numpy import *
from numpy.linalg import qr
# verwendet fuer plots
from mpl_toolkits.axes_grid1 import make_axes_locatable


def GR(a):
    """
    Gram-Schmidt Verfahren

    Keyword Arguments:
    a -- (m x n) Matrix

    Returns: q, r
    q -- Matrix Q
    r -- Matrix R
    """
    m, n = a.shape
    q = zeros((m, n))
    r = zeros((n, n))
    for j in xrange(n):
        vj = a[:, j].copy()
        for i in xrange(j):
            r[i, j] = dot(q[:, i], a[:, j])
            vj = vj - r[i, j] * q[:, i]
        r[j, j] = linalg.norm(vj)
        q[:, j] = vj / r[j, j]
    return q, r


def GRmod(a):
    """
    Modifiziertes Gram-Schmidt Verfahren
    Keyword Arguments:
    a -- (m x n) Matrix

    Returns: q, r
    q -- Matrix Q
    r -- Matrix R
    """
    m, n = a.shape
    q = zeros((m, n))
    r = zeros((n, n))
    v = a.copy()
    for i in xrange(n):
        r[i, i] = linalg.norm(v[:, i])
        q[:, i] = v[:, i] / r[i, i]
        for j in xrange(i + 1, n):
            r[i, j] = dot(q[:, i], v[:, j])
            v[:, j] = v[:, j] - r[i, j] * q[:, i]
    return q, r


# Matrix Definition
n = 50
m = 50
Z = zeros((m, n))
for i in xrange(m):
    for j in xrange(n):
        Z[i, j] = 1 + min(i, j)

# numpy QR-implementation (als Vergleich)
```

Siehe nächstes Blatt!

```python
q0, r0 = qr(Z, mode='full')
# Guete des Gram-Schmidt Verfahren
q2, r2 = GR(Z)
# Guete des modifizierten Gram-Schmidt Verfahren
q3, r3 = GRmod(Z)
print("numpys qr liefert:         %.10e" % (dot(q0.T, q0) - eye(n)).max())
print("Gram-Schmidt liefert:      %.10e" % (dot(q2.T, q2) - eye(n)).max())
print("mod. Gram-Schmidt liefert: %.10e" % (dot(q3.T, q3) - eye(n)).max())


## Fehler plots
figure()
ax = subplot(131)
title(r'$\log(|Q^T Q - I|)$ ,Numpys qr')
im = imshow(log10(abs(dot(q0.T,q0)-eye(n))+1e-16),vmin=-16, vmax=1, 
    interpolation='nearest')
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)
colorbar(im, cax=cax)

ax = subplot(132)
title(r'$\log(|Q^T Q - I|)$, Gram-Schmidt')
im = imshow(log10(abs(dot(q2.T,q2)-eye(n))+1e-16), vmin=-16, vmax=1, 
    interpolation='nearest')
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)
colorbar(im, cax=cax)

ax = subplot(133)
title(r'$\log(|Q^T Q - I|)$, mod. Gram-Schmidt')
im = imshow(log10(abs(dot(q3.T,q3)-eye(n))+1e-16), vmin=-16, vmax=1, 
    interpolation='nearest')
divider = make_axes_locatable(ax)
cax = divider.append_axes("right", size="5%", pad=0.05)
colorbar(im, cax=cax)
savefig('qr-errors.png')
```

Listing 2: Aufgabe 1): Resultat zu dem Skript in Listing 1

```
    numpys qr liefert:         1.7763568394e-15
    Gram-Schmidt liefert:      9.9967692280e-01
    mod. Gram-Schmidt liefert: 2.5757607852e-13
```

Aufgrund von Rundungsfehlern ist das Gram-Schmidt-Verfahren numerisch instabil, insbesondere geht die Orthogonalität verloren. Das modifizierte Gram-Schmidt-Verfahren ist zwar stabiler, aber lange nicht so stabil wie das Householder-Verfahren.

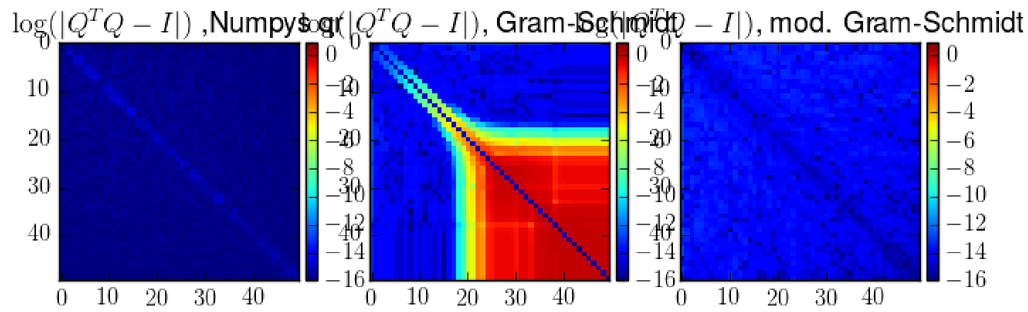

Abbildung 1: Aufgabe 1: $\log_{10}(|Q^T Q - I|)$

2. *Radioaktiver Zerfall*

In einem Gefäss befinden sich n verschiedene Elemente $Z_1, \ldots, Z_n$. Zum Zeitpunkt t sei $M_k(t)$ die Menge von Element Z_k. Die Elemente seien radioaktiv und die Zerfallsprodukte zerfallen selbst nicht weiter. Die Zerfallskonstanten $\lambda_1, \ldots, \lambda_n$ sind gegeben. Zu m $(m \geq n)$ Zeiten t_j erfolgt eine Messung der Aktivität $G(t_j)$.

Folgende physikalische Gesetze werden angenommen:

1. Zerfallsgesetz: $M_i(t) = M_i(0)\exp(-\lambda_i t)$, $t \geq 0$
2. Gesamtaktivität: $G(t) = \sum_{i=1}^{n} G_i(t) = \sum_{i=1}^{n} \lambda_i M_i(t)$

Formulieren Sie ein Ausgleichsproblem zur Bestimmung von $M_1(0), \ldots, M_n(0)$. Lösen Sie dieses Ausgleichsproblem für $n = 2, \lambda_1 = \frac{1}{10}, \lambda_2 = \frac{1}{100}$ und die Messdaten:

t_j	1	3	5	7
$G(t_j)$	24.04	20.64	17.84	15.53

Solution:
Wir formulieren das Ausgleichsproblem $\mathbf{A}\underline{x} = \underline{b}$ $(t_1, \ldots, t_4,\ \lambda_1, \lambda_2$ und $G(t_1), \ldots, G(t_4)$ gege-

Siehe nächstes Blatt!

ben):

$$\begin{pmatrix} \lambda_1 e^{-\lambda_1 t_1} & \lambda_2 e^{-\lambda_2 t_1} \\ \lambda_1 e^{-\lambda_1 t_2} & \lambda_2 e^{-\lambda_2 t_2} \\ \lambda_1 e^{-\lambda_1 t_3} & \lambda_2 e^{-\lambda_2 t_3} \\ \lambda_1 e^{-\lambda_1 t_4} & \lambda_2 e^{-\lambda_2 t_4} \end{pmatrix} \begin{pmatrix} M_1(0) \\ M_2(0) \end{pmatrix} = \begin{pmatrix} G(t_1) \\ G(t_2) \\ G(t_3) \\ G(t_4) \end{pmatrix}$$

Um die Lösung kleinster Quadrate zu finden, verwenden wir die Normalengleichungen:

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\underline{x} = \mathbf{A}^{\mathrm{T}}\underline{b}\,. \tag{1}$$

Wir erhalten mit dem `Python`-Skript in Listing 3 die Lösungen:

$$M_1(0) \approx 199.97$$
$$M_2(0) \approx 600.47\,.$$

Listing 3: Aufgabe 2): Ein einfaches `Python`-Skript zu Aufgabe 2)

```
from numpy import *
from numpy.linalg import qr, solve

# Daten
t = array([1., 3., 5., 7.])              # Zeit t_j
G = array([24.04, 20.64, 17.84, 15.53])  # G(t_j)

# Zerfallskonstanten
lmbda = array([0.1, 0.01])

# bilde Matrix A
A = zeros((size(t), size(lmbda)))
for it in range(size(t)):
    for il in range(size(lmbda)):
        A[it, il] = lmbda[il] * exp(-lmbda[il] * t[it])

# bilde Vektor b (= G)
b = G

# Variante 1: direkte Loesung
# loese die Normalengleichungen A^T A x = A^T b
AT = A.T  # A^T, aequivalent zu tranpose(A)
x = solve(dot(AT, A), dot(AT, b))

# ausgabe
print('Variante 1:')
print('x: %s' % str(x) )
print('Residuum: %s' % str(dot(A, x) - b) )

# Variante 2: Loesung mittels QR-Zerlegung
Q, R = qr(A)  # QR-Zerlegung von A
x = solve(R, dot(transpose(Q), b))  # Loese R x = Q b

# ausgabe
print('Variante 2:')
print('x: %s' % str(x))
print('Residuum: %s' % str(dot(A, x) - b) )
```

Listing 4: Aufgabe 2): Resultat zu dem Skript in Listing 3

```
    Variante 1:
    x: [ 199.97073521  600.4654466 ]
    Residuum: [-0.00099247  0.00138653  0.00064221 -0.00104446]
    Variante 2:
    x: [ 199.97073521  600.4654466 ]
    Residuum: [-0.00099247  0.00138653  0.00064221 -0.00104446]
```

Bitte wenden!

Bemerkungen:

- Anstelle der Normalengleichungen kann man die **QR**-Zerlegung benützen: $\mathbf{A}\underline{x} = \mathbf{QR}\underline{x} = \underline{b}$, bzw. $\mathbf{R}\underline{x} = \mathbf{Q}^{\mathrm{T}}\underline{b}$. Wir brauchen hier von **Q** nur die ersten zwei Zeilen, da Zeilen 3 und 4 nicht zur Lösung beitragen.

Siehe nächstes Blatt!

3. *Die Normalengleichungen sind schlecht konditioniert*

Wir betrachten die Matrix:

$$\mathbf{A} = \begin{pmatrix} 1+\varepsilon & 1 \\ 1-\varepsilon & 1 \\ \varepsilon & \varepsilon \end{pmatrix}. \tag{2}$$

In exakter Arithmetik ist die Normalengleichung:

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\underline{x} = \mathbf{A}^{\mathrm{T}}\underline{b} \tag{3}$$

äquivalent zu

$$\mathbf{B}_\alpha \begin{pmatrix} \underline{r} \\ \underline{x} \end{pmatrix} := \begin{pmatrix} -\alpha\mathbf{I} & \mathbf{A} \\ \mathbf{A}^{\mathrm{T}} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \underline{r} \\ \underline{x} \end{pmatrix} = \begin{pmatrix} \underline{b} \\ \underline{0} \end{pmatrix}. \tag{4}$$

Schreiben Sie ein `Python`-Skript, das die Kondition von $\mathbf{A}$, $\mathbf{A}^{\mathrm{T}}\mathbf{A}$, $\mathbf{B}_1$ und $\mathbf{B}_\alpha$ mit $\alpha = \varepsilon\|\mathbf{A}\|_2/\sqrt{2}$ für $10^{-5} < \varepsilon < 1$ plottet. Das `Python`-Modul `numpy.linalg` hat eine Funktion `cond`.

Solution:
Das `Python`-Skript in Listing 5 erzeugt die Abbildung 2. Im Gegensatz zur Matrix $\mathbf{B}$ ist die Matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ deutlich schlechter konditioniert als $\mathbf{A}$.

Listing 5: Aufgabe 3): Ein einfaches `Python`-Skript zu Aufgabe 3)

```
from numpy import *
from matplotlib.pyplot import *
from numpy.linalg import cond, norm

# bilde matrix A = A1 + epsilon*Aeps
A1 = array([[1, 1],
            [1, 1],
            [0, 0]])
Aeps = array([[+1, 0],
              [-1, 0],
              [+1, +1]])

# berechne Kondition von A, A^T A, B_1 und B_alpha mit
#   alpha = epsilon*||A||_2/sqrt(2) fuer 1.e-5 <= epsilon <= 1
Nepsilon = 20  # anzahl epsilons im intervall 1.e-5 < epsilon <= 1
epsilons = logspace(0, -5, num=Nepsilon)

# erstelle speicher fuer die verschiedenen Kondition s Werte
kond_A    = zeros(Nepsilon)
kond_ATA = zeros(Nepsilon)
kond_B1  = zeros(Nepsilon)
kond_Balpha = zeros(Nepsilon)

# berechne Kondition(en) fuer alle epsilons
for ieps in range(Nepsilon):
    print('Berechne fuer %.8e' % epsilons[ieps])
    # bilde Matrix A und berechne Kondition
    A = A1 + Aeps * epsilons[ieps]
    kond_A[ieps] = cond(A)
    # bilde Matrix AT A und berechne Kondition
    ATA = dot(A.T, A)
    kond_ATA[ieps] = cond(ATA)
    # bilde Matrix B1 und berechne Kondition
    alpha = 1.
    I = eye(shape(A)[0])
    ZeroM = zeros((shape(A)[1], shape(A)[1]))
    B1 = vstack((hstack(( -alpha * I, A    )),
```

```
                  hstack(( A.T       , ZeroM))))
    # Beachte: hier verwenden wir die nuetzlichen funktionen hstack & vstack
    #          in numpy...
    kond_B1[ieps] = cond(B1)
    # bilde Matrix Balpha und berechne Kondition
    alpha = epsilons[ieps] * norm(A, ord=2) / sqrt(2)
    I = eye(shape(A)[0])
    Balpha = vstack((hstack(( -alpha * I, A    )),
                     hstack(( A.T       , ZeroM))))
    kond_Balpha[ieps] = cond(Balpha)

# zeichne die Konditions Zahlen als Funktion von epsilon
loglog(epsilons, kond_A,   label=r'cond$_2({\bf A})$')
loglog(epsilons, kond_ATA, label=r'cond$_2({\bf A}^{T}{\bf A})$')
loglog(epsilons, kond_B1,  label=r'cond$_2({\bf B}_1)$')
loglog(epsilons, kond_Balpha, label=r'cond$_2({\bf B}_{\alpha})$')
grid(True)
legend()
xlabel(r'$\epsilon$')
savefig('condiLSP.pdf')
show()
```

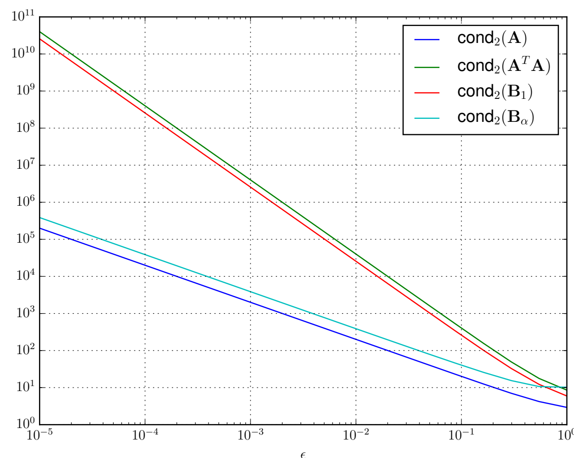

Abbildung 2: Aufgabe 3)

Siehe nächstes Blatt!

4. Kernaufgabe: Adaptive Methoden für steife Systeme

Solution:
12 Punkte insgesamt.

a) Implementieren Sie die Rosenbrock-Wanner Methoden der Ordnung 2 und 3. Es sollen Funktionen `row_2_step(f, Jf, yi, h)` und `row_3_step(f, Jf, yi, h)` geschrieben werden, die ausgehend vom Wert $y_i(t_i)$ genau einen Zeitschritt h der entsprechenden Methode berechnen und die Propagierte $y_{i+1}(t_i + h)$ zurück geben.

Hinweis: Die Parameter sind im Template `stiff_row_Template.py` erklärt.

Solution:
4 Punkte. (2 pro Methode)

b) Lösen Sie die logistische Differentialgleichung:

$$\dot{y}(t) = \lambda y(t)(1 - y(t))$$

mit dem Anfangswert $y(0) = c = 0.01$ und $\lambda = 25$ bis zum Zeitpunkt $T = 2$. Benutzen Sie $N = 100$ Zeitschritte. Plotten Sie die numerischen Lösungen $y(t)_{\texttt{ROW}}$ sowie die Fehler $y(t)_{\texttt{ROW}} - y(t)$ beider Methoden gegen die Zeit. Wie gross kann λ sein, bevor der Fehler der `ROW-2` Methode einen maximalen Wert von 0.05 überschreitet?

Solution:
1 Punkt. Für $\lambda = 56$ überschreitet der absolute Fehler die gegebene Grenze.

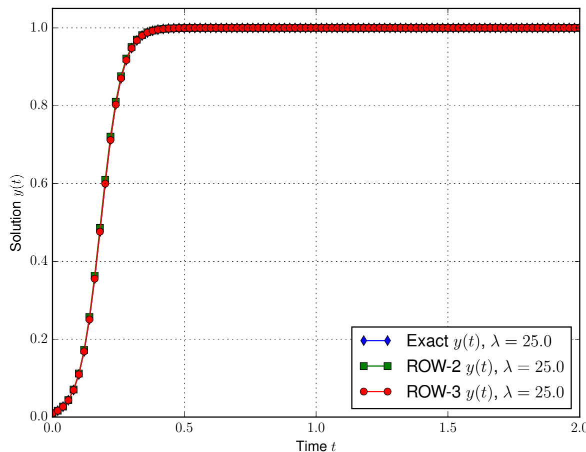

Abbildung 3: Lösung der logistischen Differentialgleichung mithilfe der ROW Methoden.

c) Messen Sie die Konvergenzordnung beider Methoden. Benutzen Sie hierfür obige Gleichung und Anfangswerte mit $\lambda = 10$. Wählen Sie $N = [2^4, \ldots, 2^{12}]$ und berechnen Sie den Fehler zum Endzeitpunkt $T = 2$ gegenüber der exakten Lösung:

$$y(t) = \frac{ce^{\lambda t}}{1 - c + ce^{\lambda t}}$$

Plotten Sie den Fehler gegen die Anzahl Schritte doppelt logarithmisch.

Solution:
1 Punkt.

Bitte wenden!

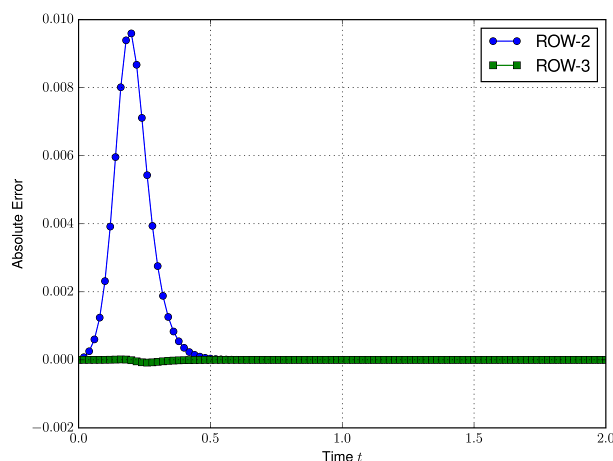

Abbildung 4: Fehler der Lösung der logistischen Differentialgleichung.

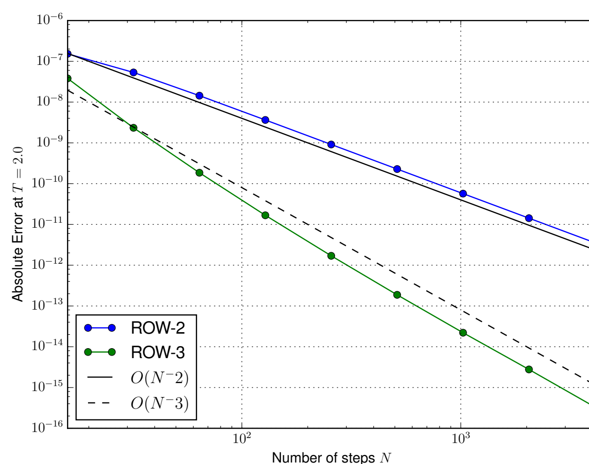

Abbildung 5: Konvergenzverhalten der ROW Methoden in Bezug auf die Zeitschrittgrösse.

d) Implementieren Sie eine adaptive Strategie basierend auf den `ROW-2` und `ROW-3` Methoden. Verwenden Sie als Fehlerschätzer die Norm:

$$\varepsilon_i := \|y(t_i)_{\texttt{ROW-2}} - y(t_i)_{\texttt{ROW-3}}\|_2$$

Wählen Sie den initialen Zeitschritt als $h_0 = T/(100\,(\|f(y_0)\|_2 + 0.1))$ und passen Sie die Grösse des nächsten Zeitschritts durch Verkleinern ($h_{j+1} = \frac{h_j}{2}$) oder Vergrössern ($h_{j+1} = 1.1h_j$) an.

Solution:
3 Punkte. (1 für Fehlerschätzer, 1 für Variation des Schrittes, 1 sonst)

e) Testen Sie die Implementation wiederum an der logistischen Differentialgleichung mit $\lambda = 50$. Wie viele Zeitschritte werden insgesamt zur Lösung benötigt? Plotten Sie die numerische Lösung $y(t)_{\texttt{ADA}}$ sowie die Fehler $y(t)_{\texttt{ADA}} - y(t)$ gegen die Zeit.

Solution:
1 Punkt. Es werden 71 Zeitschritte benötigt.

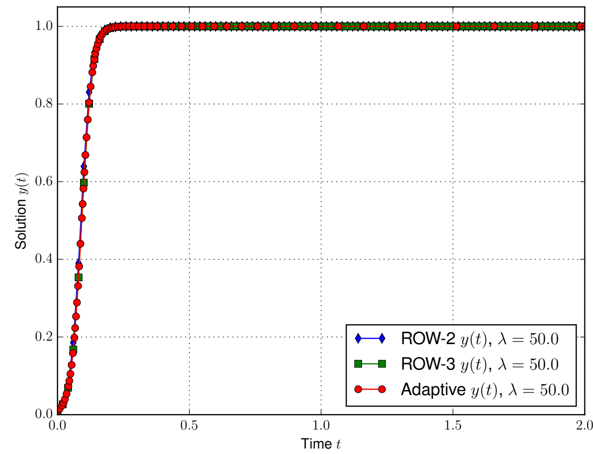

Abbildung 6: Lösung der logistischen Differentialgleichung mithilfe einer adaptiven Methode.

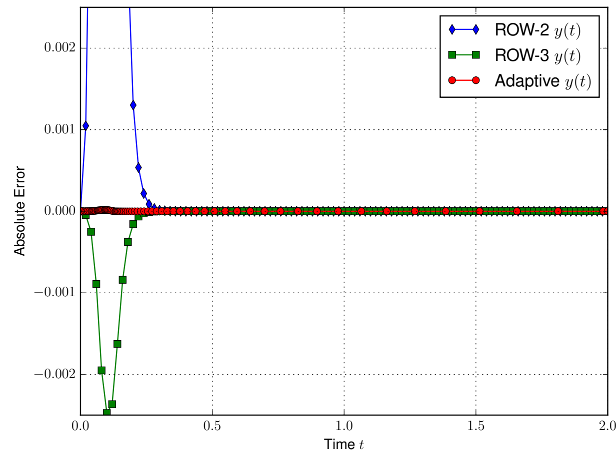

Abbildung 7: Fehler der Lösung der logistischen Differentialgleichung.

f) Lösen Sie das folgende gekoppelte System:

$$\dot{y}_0(t) = -76\, y_0(t) - 25\sqrt{3}\, y_1(t)$$
$$\dot{y}_1(t) = -25\sqrt{3}\, y_0(t) - 26\, y_1(t)$$

mit Anfangswerten $y_0(0) = 1$ und $y_1(0) = 1$ bis zum Zeitpunkt $T = 1$ mit dem adaptiven Verfahren und einer Anfangsschrittweite von $h = 0.1$, plotten Sie $y(t)$.

Solution:
1 Punkt.

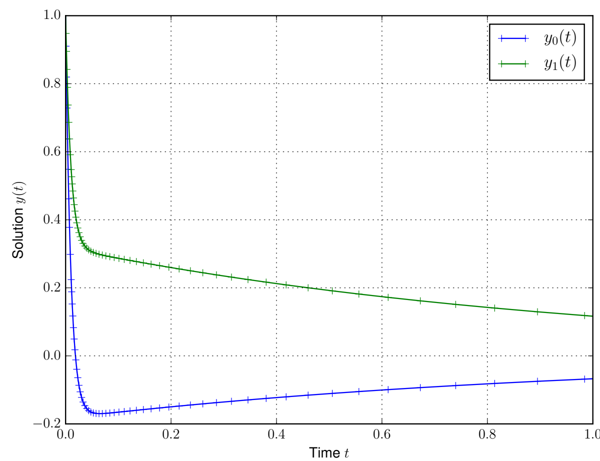

Abbildung 8: Lösungen $y_0(t)$ und $y_1(t)$ des Gleichungssystems.

g) Lösen Sie die folgende sehr steife Gleichung:

$$\dot{y}(t) = \lambda y^2(t)(1 - y^2(t))$$

mit dem Anfangswert $y(0) = 0.01$ und $\lambda = 500$. Plotten Sie die numerische Lösung $y(t)_{\texttt{ADA}}$ sowie die Grösse der Zeitschritte gegen die Zeit. Wie viele Zeitschritte benötigt dieses Verfahren und was ist der kleinste Zeitschritt? Wie viele Zeitschritte dieser Grösse würde ein nicht-adapives Verfahren benötigen?

Hinweis: Schauen Sie sich `help(diff)` an.

Solution:
1 Punkt.

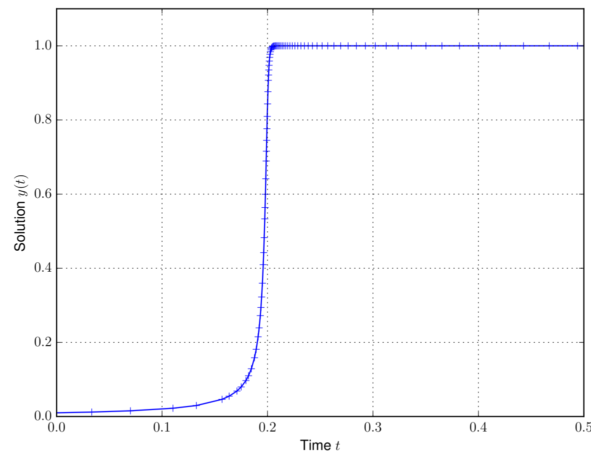

Abbildung 9: Lösung der steifen Gleichung.

```
Minimal steps size: 0.000208
```

Siehe nächstes Blatt!

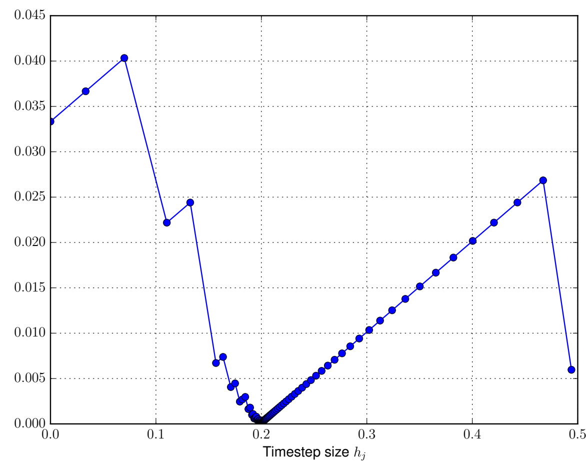

Abbildung 10: Zeitschrittgrösse gegenüber der Zeit.

```
        Number of adaptive steps: 88
        Number of non-adaptive steps: 2404.84
```

Solution:

Listing 6: Python-Skript

```
from numpy import *
from numpy.linalg import solve, norm
from matplotlib.pyplot import *


###################
# Unteraufgabe a) #
###################

def row_2_step(f, Jf, yi, h):
    r"""Rosenbrock-Wanner Methode der Ordnung 2

    Input:
    f:   Die rhs Funktion f(x)
    Jf:  Jacobi Matrix J(x) der Funktion: R^(nx1) -> R^(nxn)
    yi:  Aktueller Wert y_i zur Zeit ti
    h:   Schrittweite

    Output:
    yip1:  Zeitpropagierter Wert y(t+h): R^(nx1)
    """
    yi = atleast_2d(yi)
    n = yi.shape[0]
    a = 1.0 / (2.0 + sqrt(2.0))
    I = identity(n)
    J = Jf(yi)
    A = I - a*h*J
    # k1
    b1 = f(yi)
    k1 = solve(A, b1)
    # k2
    b2 = f(yi+0.5*h*k1) - a*h*dot(J,k1)
```

```
    k2 = solve(A, b2)
    # Advance
    yip1 = yi + h*k2
    return yip1


def row_3_step(f, Jf, yi, h):
    r"""Rosenbrock-Wanner Methode der Ordnung 3

    Input:
    f:   Die rhs Funktion f(x): R^(nx1) -> R^(nx1)
    Jf:  Jacobi Matrix J(x) der Funktion: R^(nx1) -> R^(nxn)
    yi:  Aktueller Wert y_i zur Zeit ti
    h:   Schrittweite

    Output:
    yip1:  Zeitpropagierter Wert y(t+h): R^(nx1)
    """
    yi = atleast_2d(yi)
    n = yi.shape[0]
    a = 1.0 / (2.0 + sqrt(2.0))
    d31 = - (4.0 + sqrt(2.0)) / (2.0 + sqrt(2.0))
    d32 = (6.0 + sqrt(2.0)) / (2.0 + sqrt(2.0))
    I = identity(n)
    J = Jf(yi)
    A = I - a*h*J
    # k1
    b1 = f(yi)
    k1 = solve(A, b1)
    # k2
    b2 = f(yi+0.5*h*k1) - a*h*dot(J,k1)
    k2 = solve(A, b2)
    # k3
    b3 = f(yi+h*k2) - d31*h*dot(J,k1) - d32*h*dot(J,k2)
    k3 = solve(A, b3)
    # Advance
    yip1 = yi + h/6.0*(k1 + 4*k2 + k3)
    return yip1


def constructor(stepalg):
    r"""
    Input:
    stepalg:  Methode um einen Zeitschritt zu machen

    Output:
    stepper:  Einen Integrator der die gegebene Methode anwendet.
    """

    def stepper(f, Jf, t0, y0, h, N):
        r"""
        Input:
        f:   Die rhs Funktion f(x): R^(nx1) -> R^(nx1)
        Jf:  Jacobi Matrix J(x) der Funktion: R^(nx1) -> R^(nxn)
        t0:  Startzeitpunt: R^1
        y0:  Anfangswert y(t0): R^(nx1)
        h:   Schrittweite
        N:   Anzahl Schritte

        Output:
        t:    Zeitpute ti: R^N
        sol:  Loesung y(ti): R^(nxN)
        """
        # Copy and reshape input
        ti = atleast_1d(t0).copy().reshape(1)
```

Siehe nächstes Blatt!

```python
        yi = atleast_1d(y0).copy().reshape(-1,1)
        # Wrap function calls for shape consistency
        n = yi.shape[0]
        ff = lambda y: f(y).reshape(n,1)
        Jff = lambda y: Jf(y).reshape(n,n)
        # Collect results
        t = [ti]
        sol = [yi]
        # Time iteration
        for i in xrange(1,N+1):
            ti = ti + h
            yi = stepalg(ff, Jff, yi, h)
            t.append(ti)
            sol.append(yi)
        # Stack together results
        return hstack(t).reshape(-1), hstack(sol).reshape(n,-1)

    return stepper


# Construct integrators for ROW-2 and ROW-3
row_2 = constructor(row_2_step)
row_3 = constructor(row_3_step)


###################
# Unteraufgabe b) #
###################

def aufgabe_b():
    print(" Aufgabe b)")
    # Logistic ODE
    c = 0.01
    l = 25.0
    f = lambda y: l*y*(1 - y)
    Jf = lambda y: l - 2*l*y

    sol = lambda t: (c*exp(l*t)) / (1 - c + c*exp(l*t))
    t0 = 0.0
    y0 = c

    T = 2.0
    N = 100
    h = T/float(N)

    t2, s2 = row_2(f, Jf, t0, y0, h, N)
    t3, s3 = row_3(f, Jf, t0, y0, h, N)

    t = linspace(t0, T, N+1)
    y = sol(t)

    figure()
    plot(t, y, "-d", label="Exact $y(t)$, $\lambda = %3.1f$" % l)
    plot(t2, squeeze(s2), "-s", label="ROW-2 $y(t)$, $\lambda = %3.1f$" % l)
    plot(t3, squeeze(s3), "-o", label="ROW-3 $y(t)$, $\lambda = %3.1f$" % l)
    grid(True)
    xlim(t.min(), t.max())
    ylim(0, 1.05)
    legend(loc="lower right")
    xlabel(r"Time $t$")
    ylabel(r"Solution $y(t)$")
    savefig("solution_logistic_row.pdf")

    figure()
    plot(t, squeeze(s2) - y, "-bo", label="ROW-2")
```

Bitte wenden!

```
163     plot(t, squeeze(s3) - y, "-gs", label="ROW-3")
        grid(True)
165     xlim(t.min(), t.max())
        legend(loc="upper right")
167     xlabel(r"Time $t$")
        ylabel(r"Absolute Error")
169     savefig("error_logistic_row.pdf")

171
    ###################
173 # Unteraufgabe c) #
    ###################
175
    def aufgabe_c():
177     print(" Aufgabe c)")
        # Logistic ODE
179     c = 0.01
        l = 10.0
181     f = lambda y: l*y*(1-y)
        Jf = lambda y: l- 2*l*y
183
        sol = lambda t: (c*exp(l*t)) / (1 - c + c*exp(l*t))
185     t0 = 0.0
        y0 = c
187     T = 2.0

189     # Different number steps
        steps = 2**arange(4,13)
191
        # Storage for solution values
193     datae = []
        data2 = []
195     data3 = []

197     for N in steps:
            t, h = linspace(t0, T, N+1, retstep=True)
199         t2, sol2 = row_2(f, Jf, t0, y0, h, N)
            t3, sol3 = row_3(f, Jf, t0, y0, h, N)
201         sol2 = sol2.reshape(-1)
            sol3 = sol3.reshape(-1)
203
            datae.append(sol(t)[-1])
205         data2.append(sol2[-1])
            data3.append(sol3[-1])
207
        datae = array(datae)
209     data2 = array(data2)
        data3 = array(data3)
211
        err2 = abs(data2 - datae)
213     err3 = abs(data3 - datae)

215     figure()
        loglog(steps, err2, "b-o", label="ROW-2")
217     loglog(steps, err3, "g-o", label="ROW-3")
        loglog(steps, 1e-5*(float(T)/steps)**2, "-k", label="$O(N^-2)$")
219     loglog(steps, 1e-5*(float(T)/steps)**3, "--k", label="$O(N^-3)$")
        grid(True)
221     xlim(steps.min(), steps.max())
        legend(loc="lower left")
223     xlabel(r"Number of steps $N$")
        ylabel(r"Absolute Error at $T = %.1f$" % T)
225     savefig("convergence_row.pdf")

227
```

Siehe nächstes Blatt!

```
###################
# Unteraufgabe d) #
###################

def odeintadapt(Psilow, Psihigh, T, y0, fy0=None, h0=None, hmin=None, reltol=1e-2, abstol=1e-4):
    r"""Adaptive Integrator for stiff ODEs and Systems
    based on two suitable methods of different order.

    Input:
    Psilow:   Integrator of low order
    Psihigh:  Integrator of high order
    T:        Endtime: R^1
    y0:       Initial value: R^(nx1)
    fy0:      The value f(y0): R^(nx1) used to estimate initial timestep size
    h0:       Initial timestep (optional)
    hmin:     Minimal timestep (optional)
    reltol:   Relative tolerance (optional)
    abstol:   Absolute Tolerance (optional)

    Output:
    t:    Timesteps: R^K with K the number of steps performed
    y:    Solution at the timesteps: R^(nxK)
    rej:  Time of rejected timesteps: R^G with G the number of rejections
    ee:   Estimated error: R^K
    """
    # Heuristic choice of initial timestep size
    if h0 is None:
        h0 = T / (100.0*(norm(fy0) + 0.1))
    if hmin is None:
        hmin = h0 / 10000.0

    # Initial values
    yi = atleast_2d(y0).copy()
    ti = 0.0
    hi = h0
    n = yi.shape[0]

    # Storage
    t = [ti]
    y = [yi]
    rej = []
    ee = [0.0]

    while ti < T and hi > hmin:
        yh = Psilow(hi, yi)
        yH = Psihigh(hi, yi)
        est = norm(yh - yH)
        if est < min(reltol*norm(yi), abstol):
            # Timestep is too long and ends beyond T
            if T - ti < hi:
                hi = T - ti
                yi = Psihigh(hi, yi)
            else:
                yi = yH

            ti = ti + hi
            hi = 1.1*hi

            y.append(yi)
            t.append(ti)
            ee.append(est)
        else:
            hi = 0.5*hi

```

Bitte wenden!

```
            rej.append(ti)

    return array(t).reshape(-1), array(y).T.reshape(n,-1), array(rej), array(ee)


###################
# Unteraufgabe e) #
###################

def aufgabe_e():
    print(" Aufgabe e)")
    # Logistic ODE
    c = 0.01
    l = 50.0
    f = lambda y: l*y*(1-y)
    Jf = lambda y: l- 2*l*y

    sol = lambda t: (c*exp(l*t)) / (1 - c + c*exp(l*t))
    y0 = sol(0.0)
    T = 2.0

    nsteps = 0

    # Discrete Evolution Operators
    Psilow =  lambda h, y: row_2_step(f, Jf, y, h)
    Psihigh = lambda h, y: row_3_step(f, Jf, y, h)

    # Adaptive timestepping
    ta, ya, rej, ee = odeintadapt(Psilow, Psihigh, T, y0, f(y0))
    nsteps = len(ta)

    print("Es werden %d Zeitschritte benoetigt" % nsteps)

    N = 100
    t2, s2 = row_2(f, Jf, array(0.0), y0, T/float(N), N)
    N = 100
    t3, s3 = row_3(f, Jf, array(0.0), y0, T/float(N), N)

    figure()
    plot(t2, squeeze(s2), "-bd", label=r"ROW-2 $y(t)$, $\lambda = %3.1f$" % l)
    plot(t3, squeeze(s3), "-gs", label=r"ROW-3 $y(t)$, $\lambda = %3.1f$" % l)
    plot(ta, squeeze(ya), "-ro", label=r"Adaptive $y(t)$, $\lambda = %3.1f$" % l)
    grid(True)
    xlim(0, T)
    ylim(0, 1.05)
    legend(loc="best")
    xlabel(r"Time $t$")
    ylabel(r"Solution $y(t)$")
    savefig("solution_logistic_adaptive.pdf")

    figure()
    plot(t2, squeeze(s2) - sol(t2), "-bd", label=r"ROW-2 $y(t)$")
    plot(t3, squeeze(s3) - sol(t3), "-gs", label=r"ROW-3 $y(t)$")
    plot(ta, squeeze(ya) - sol(ta), "-ro", label=r"Adaptive $y(t)$")
    grid(True)
    xlim(0, T)
    ylim(-0.0025, 0.0025)
    legend(loc="best")
    xlabel(r"Time $t$")
    ylabel(r"Absolute Error")
    savefig("error_logistic_adaptive.pdf")


###################
# Unteraufgabe f) #
```

Siehe nächstes Blatt!

```
###################

def aufgabe_f():
    print(" Aufgabe f)")
    # Test case
    tau = 4*pi/3.0
    R = array([[sin(tau), cos(tau)],[-cos(tau), sin(tau)]])
    D = array([[-101.0, 0.0],[0.0, -1.0]])
    A = dot(R.T, dot(D, R))

    f = lambda y: dot(A, y)
    Jf = lambda y: A
    y0 = array([[1.0],
                [1.0]])
    T = 1.0

    # Discrete Evolution Operators
    Psilow =  lambda h,y: row_2_step(f, Jf, y, h)
    Psihigh = lambda h,y: row_3_step(f, Jf, y, h)

    # Adaptive timestepping
    ta, ya, rej, ee = odeintadapt(Psilow, Psihigh, T, y0, h0=0.1)

    figure()
    plot(ta, ya[0,:], "-+", label=r"$y_0(t)$")
    plot(ta, ya[1,:], "-+", label=r"$y_1(t)$")
    grid(True)
    xlim(0, T)
    legend(loc="upper right")
    xlabel(r"Time $t$")
    ylabel(r"Solution $y(t)$")
    savefig("solution_system.pdf")


###################
# Unteraufgabe g) #
###################

def aufgabe_g():
    print(" Aufgabe g)")
    # Logistic ODE with y squared
    l = 500.0
    f = lambda y: l*y**2*(1-y**2)
    Jf = lambda y: l*(2*y-4*y**3)
    y0 = 0.01
    T = 0.5

    hmin = 1.0
    nrsteps = 0.0

    # Discrete Evolution Operators
    Psilow =  lambda h,y: row_2_step(f, Jf, y, h)
    Psihigh = lambda h,y: row_3_step(f, Jf, y, h)

    # Adaptive timestepping
    ta, ya, rej, ee = odeintadapt(Psilow, Psihigh, T, y0, f(y0))

    hmin = min(diff(ta))
    nrsteps = len(ta)

    figure()
    plot(ta, squeeze(ya), "+-")
    grid(True)
    xlim(0, T)
    ylim(0, 1.1)
```

```
    xlabel(r"Time $t$")
423 ylabel(r"Solution $y(t)$")
    savefig("solution_superstiff.pdf")
425
    figure()
427 plot(ta[:-1], diff(ta), "-o")
    grid(True)
429 xlim(0, T)
    xlabel(r"Time $t$")
431 xlabel(r"Timestep size $h_j$")
    savefig("timestepsize_superstiff.pdf")
433

435 print("Minimal steps size: %f" % hmin)
    print("Number of adaptive steps: %i" % nrsteps)
437 print("Number of non-adaptive steps: %.2f" % (T/hmin))

439


441
if __name__ == "__main__":
443 # Run all subtasks
    aufgabe_b()
445 aufgabe_c()
    aufgabe_e()
447 aufgabe_f()
    aufgabe_g()
```