ETHZ, BSC RW/CSE

Prüfung
Numerische Mathematik für CSE/RW HS 07

Prof. D. Kressner

Wichtige Hinweise

- Entpacken Sie die Datei sources.tar im Home-Verzeichnis mit dem Befehl tar -xvf sources.tar.
- Prüfungsdauer: 180 Minuten
- Zugelassene Hilfsmittel: 6 beidseitig handgeschriebene A4 Blätter, MATLAB, Vorlesungsunterlagen unter ~/Dokumente/NCSE.pdf.
- Jede Aufgabe auf einer neuen Seite beginnen und jedes Blatt mit **Namen**, **Aufgabennummer** und **Rechnername** deutlich beschriften.
- Jede MATLAB-Programmdatei in der ersten Zeile mit einem Kommentar versehen, der Name und (Teil)Aufgabennummer enthält.
- Verwenden Sie die vorbereiteten MATLAB-Files/Gerüste in den Verzeichnissen ~/Aufgabe1, ... ~/Aufgabe6. Lösungen nur hier ablegen und keine weiteren Unterverzeichnisse anlegen!
- Ausdrucken der MATLAB-Programme am Prüfungsende mit dem Skript *./print_ncse_files.bsh*. Das Skript sammelt alle *.m-Files und *.eps-Files ein und schickt diese an den Drucker.
- Nicht durch Hardwareversagen entstandener Dateiverlust ist vom Prüfungskandidaten zu verantworten.

---

1. **(6 Punkte) Zusammengesetzte Weddle-Regel**

   Das Integral

   $$I = \int_0^2 (1 - \sqrt{x}\cos x)\mathrm{d}x \tag{1}$$

   soll mit der auf der Weddle-Regel basierenden zusammengesetzten Quadraturregel bestimmt werden. Die Weddle-Regel

   $$\int_0^h f(x)\mathrm{d}x = \frac{h}{840}\left(41f(0) + 216f\left(\frac{1}{6}h\right) + 27f\left(\frac{2}{6}h\right) + 272f\left(\frac{3}{6}h\right) + 27f\left(\frac{4}{6}h\right) + 216f\left(\frac{5}{6}h\right) + 41f(h)\right)$$

   verwendet auf jedem Teilintervall 7 Stützstellen.

   a) Implementieren Sie eine MATLAB-Routine

   ```
   function [Q]=weddle_quad(n,f),
   ```

die (ohne for-Schleife) auf n Teilintervallen von $[0, 2]$ der Länge $h = \frac{2}{n}$ die Weddle-Quadratur einer Funktion $f(x)$ berechnet und die Resultate summiert.

b) Implementieren Sie eine MATLAB-Routine

`function visualize_weddle_error()`,

die mit der MATLAB-Funktion **quad** eine Referenzlösung für I (siehe (1)) berechnet und den Fehler der zusammengesetzten Weddle-Regel für das gegebene Beispiel in Abhängigkeit von $h = \frac{2}{n}$ in geeignetem Massstab darstellt. Wählen Sie $n = 10^i, i = 1, 2, \ldots 5$ und speichern Sie das Ergebnis unter *weddle_rate.eps* ab.

c) Bestimmen Sie numerisch die Konvergenzordnung des zusammengesetzten Verfahrens zur Berechnung von I wie in (1). Benutzen Sie dazu die MATLAB-Funktion aus Teilaufgabe b).

2. **(8 Punkte) Runge-Kutta-Schema**

Zur Approximation einer gewöhnlichen Differentialgleichung der Form

$$y' = f(t, y), \quad y(t_0) = y_0$$

soll das durch den reellen Wert $c \neq 0$ parametrisierte Runge-Kutta-Verfahren

$$y_{n+1} = y_n + h\left((1-c)f(t_n, y_n) + cf(t_n + \frac{h}{2c}, y_n + \frac{h}{2c}f(t_n, y_n))\right)$$

verwendet werden.

a) Wieviele Stufen hat das Verfahren und wie sieht die entsprechende Butcher-Tabelle aus? Ist es ein explizites oder implizites Verfahren?

b) Schreiben Sie eine MATLAB-Funktion

`function [t,y]=runge_kutta(t0,y0,T,N,c,f)`,

die $y(t)$ in N-Schritten approximiert. $y0$ ist der Anfangswert $y(t0)$ zu $t0$. T ist Stoppzeit, N die Anzahl der Schritte, c die Verfahrenskonstante und f der Zeiger auf $f(t, y)$. Die Rückgabewerte $t = [t0, \ldots tN]$ und $y = [y0, \ldots yN]$ sind Vektoren mit diskreten Zeitpunkten und den approximierten Funktionswerten $y_i \approx y(t_i)$.

c) Sei nun

$$f(t, y) = -4y, \quad t_0 = 0, \quad y_0 = 10, \quad T = 10. \tag{2}$$

Wie lautet die exakte Lösung y? Bestimmen Sie für $N = 10, 11, \ldots 30$ die approximierten Funktionswerte $y_N \approx y(10)$ am Intervallende und stellen Sie die Abhängigkeit des Fehlers $|y(10) - y_N|$ von N bei festem $c = 1/2$ graphisch dar. Beschriften Sie die Achsen und speichern Sie das Ergebnis unter dem Namen *runge_kutta_stab_1.eps* (benutzen Sie `plot_runge_kutta_c()`). Ist das Stabilitätsgebiet beschränkt?

d) Stellen Sie nun für das Problem (2) aus Aufgabenteil c) die Abhängigkeit der Fehler $|y(10) - y_{30}|$ und $|y(10) - y_{10}|$ vom Verfahrensparameter c im Intervall $c \in [0.1, 1]$ dar (benutzen Sie `plot_runge_kutta_d()`). Beschriften Sie die Achsen und speichern Sie die Ergebnisse unter den Namen *runge_kutta_stab_2.eps* und *runge_kutta_stab_3.eps*. Was beobachten Sie?

e) Wie lautet das Stabilitätsgebiet S des Verfahrens? Welches ist der kleinste Wert $\mu_0 \in \mathbb{R}$, so dass das Intervall $[\mu_0, 0]$ im Stabilitätsgebiet liegt? *Hinweis: Bestimmen Sie den Schnittpunkt der negativen reellen Achse mit dem Rand von S.*

3. **(8 Punkte) Periodische Spline-Interpolation**

Zu einer T-periodischen Funktion $f : \mathbb{R} \to \mathbb{R}$ seien $n+1$ Funktionswerte im Interval $[0, T]$ gegeben:

$$t_j = jh, \quad y_j = f(t_j), \quad j = 0, \ldots, n$$

wobei $h = T/n$. Hierbei gilt wegen der Periodizität $y_0 = y_n$. Der interpolierende *periodische kubische Spline* s hat die Darstellung

$$\begin{aligned} s_{[t_{j-1},t_j]}(t) &= y_{j-1}(1 - 3\tau^2 + 2\tau^3) + y_j(3\tau^2 - 2\tau^3) \\ &\quad + hc_{j-1}(\tau - 2\tau^2 + \tau^3) + hc_j(-\tau^2 + \tau^3), \end{aligned} \tag{3}$$

auf dem Intervall $[t_{j-1}, t_j]$ mit $\tau = (t - t_{j-1})/h$ und zu bestimmenden Koeffizienten $c_0, c_1, \ldots, c_n$.

(a) Erstellen Sie eine MATLAB-Funktion `pspline_coeff`, die für gegebene $y_0, \ldots, y_{n-1}$ und T die Koeffizienten $c_0, \ldots, c_{n-1}$ bestimmt.
Das entsprechende $n \times n$ Gleichungssystem

$$\begin{pmatrix} 4 & 1 & 0 & \cdots & 0 & 1 \\ 1 & 4 & 1 & & & 0 \\ 0 & 1 & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & & & \ddots & \ddots & 1 \\ 1 & 0 & \cdots & 0 & 1 & 4 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-2} \\ c_{n-1} \end{pmatrix} = \frac{3}{h} \begin{pmatrix} y_1 - y_{n-1} \\ y_2 - y_0 \\ \vdots \\ y_{n-1} - y_{n-3} \\ y_0 - y_{n-2} \end{pmatrix}$$

soll mittels Sherman-Morrison-Woodbury-Formel in $O(n)$ Rechenaufwand gelöst werden. Dabei können Sie zur Lösung von tridiagonalen Gleichungssystemen auf den MATLAB-Operator \ für schwachbesetzte Matrizen zurückgreifen.
Wie ist c_n zu wählen?

(b) Erstellen Sie eine MATLAB-Funktion `pspline_eval`, die für gegebenes T, y, und c, den Spline s an k Punkten $\tilde{t}_1, \ldots, \tilde{t}_k \in [0, T]$ auswertet.

(c) Für eine genügend glatte T-periodische Funktion $f : \mathbb{R} \to \mathbb{R}$ ist der maximale Fehler $|s - f|$ nahezu proportional zu h^p für genügend kleine h.
Bestimmen Sie p experimentell jeweils für

$$f(t) = \sin(t), \quad T = 2\pi,$$

(benutzen Sie `pspline_sin()`) sowie für

$$f(t) = |1/2 - t| \text{ für } t \in [0, 1], \quad T = 1$$

(benutzen Sie `pspline_hat()`). Begründen Sie die unterschiedlichen Werte für p.

4. **(6 Punkte) Steifes Anfangswertproblem**

Gegeben sei das Anfangswertproblem

$$y''(t) = -201y'(t) - 200y(t) + 2, \quad 0 \le t \le 20, \quad y(0) = 0.99, \quad y'(0) = 1. \tag{4}$$

a) Schreiben Sie (4) in ein lineares Differentialgleichungssystem erster Ordnung um:

$$\boldsymbol{y}'(t) = \boldsymbol{A}\boldsymbol{y}(t), \quad \boldsymbol{y}(0) = \boldsymbol{y}_0. \tag{5}$$

b) Berechnen Sie die Eigenwerte der Matrix $\boldsymbol{A}$. Sie dürfen dazu den MATLAB-Befehl `eig` verwenden.

c) Lösen Sie das Anfangswertproblem (5) mit den MATLAB-Integratoren `ode45` und `ode23s` auf dem Zeitintervall [0, 20] und stellen Sie die Lösungen graphisch dar. Beschriften Sie die Achsen und speichern Sie die Ergebnisse unter *AWP_45.eps* und *AWP_23s.eps* (benutzen Sie `AWP()`). Beide Integratoren bestimmen adaptiv die Schrittweite. Wieviele Schritte benötigt jeder Integrator für dieses Problem? Warum verwendet der eine Integrator so viel mehr Schritte als der andere? (*Hinweis: Benutze b) und die Hilfe von* MATLAB)

5. **(6 Punkte) Fixpunktiteration**

Das lineare Gleichungssystem

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$$

mit $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ und $\boldsymbol{b} \in \mathbb{R}^n$, soll mit der folgenden Fixpunktiteration gelöst werden:

$$\boldsymbol{x}^{k+1} = (\boldsymbol{D} - \omega\boldsymbol{L})^{-1}((1-\omega)\boldsymbol{D} + \omega\boldsymbol{R})\boldsymbol{x}^k + (\boldsymbol{D} - \omega\boldsymbol{L})^{-1}\omega\boldsymbol{b}$$

Dabei ist $\boldsymbol{A} = \boldsymbol{D} - \boldsymbol{L} - \boldsymbol{R}$ die additive Zerlegung der Matrix $\boldsymbol{A}$ in eine Diagonalmatrix $\boldsymbol{D} = \mathrm{diag}(\boldsymbol{A})$, die strikte untere Dreiecksmatrix $-\boldsymbol{L}$ sowie die strikte obere Dreiecksmatrix $-\boldsymbol{R}$.

a) Zeigen Sie, dass der Fehler $\boldsymbol{e}^k := \boldsymbol{x}^k - \boldsymbol{x}$ im k-ten Schritt durch

$$\boldsymbol{e}^{k+1} = \boldsymbol{B}_\omega \boldsymbol{e}^k$$

mit

$$\boldsymbol{B}_\omega = (\boldsymbol{D} - \omega\boldsymbol{L})^{-1}((1-\omega)\boldsymbol{D} + \omega\boldsymbol{R})$$

gegeben ist.

b) Das Iterationsverfahren konvergiert genau dann wenn der Spektralradius $\rho(\boldsymbol{B}_\omega)$ der Matrix $\boldsymbol{B}_\omega$ kleiner als 1 ist. Schreiben Sie eine MATLAB-Routine,

```
function power_iter(M, tol)
```

die den Spektralradius einer Matrix $\boldsymbol{M}$ mit der Potenzmethode bestimmt.

c) Sein nun $\boldsymbol{A} \in \mathbb{R}^{n\times n}, n = 100$ mit

$$a_{ij} = \begin{cases} 4 & i = j \\ 1 & i = j-1 \\ 1 & j = i-1 \\ 0 & \text{sonst.} \end{cases}$$

Schreiben Sie eine MATLAB-Routine

```
function visualize_spec_radius(),
```

die für $\omega = [0.01, 0.02 \ldots 1.99]$ den Spektralradius $\rho(\boldsymbol{B}_\omega)$ einmal mit der Funktion aus b) und einmal mit der MATLAB-Funktion `eig` bestimmt und $\rho(\boldsymbol{B}_\omega)$ als Funktion von ω graphisch darstellt. Beschriften Sie die Achsen und speichern Sie das Ergebnis unter *sor_radius.eps*. Schätzen Sie den optimalen Wert für den Parameter ω. Worauf sind die Unterschiede zwischen `eig` und der Potenzmethode zurückzuführen.

6. **(6 Punkte) Nichtlineares Ausgleichsproblem**

Sei $f(t) := Ce^{\lambda t} \cos 2\pi t$. Um die unbekannten Parameter λ und C zu bestimmen stehen die folgenden Messwerte $b_i \approx f(t_i)$ zur Verfügung:

t_i	0.1	0.2	0.3	0.05	0.35	0.25
b_i	0.395	0.134	−0.119	0.6	−0.2	0

Formulieren Sie die Aufgabe, die Parameter λ und C zu ermitteln, als nichtlineares Ausgleichsproblem. Ergänzen Sie die MATLAB-Routine `fit()` um einen geeigneten Löser

```
function [l, c]=solver([t_1,.. t_n],[b_1,.. b_n],l,c,tol),
```

der aus gegebenen Daten $t_1, \ldots t_n$ und $b_1, \ldots b_n$ die Parameter λ und C mit Gauss-Newton berechnet und bestimmen Sie die Parameterwerte zu den Daten aus der Tabelle.

Viel Erfolg!

Musterlösung

1. a) Implementierung von summierter Weddle-Regel:

```
function [Q]=weddle_quad(n,f)
% Mein Name, Aufgabe 1
% n number of subintervals
% f function handle
a=0;                    % left boundary
b=2;                    % right boundary
h=(b-a)/(6*n);          % mesh width
y=f(a:h:b);             % data
                        % quadrature
Q=(41*y(1)+41*y(end)+...
    216*sum(y(2:6:end-5))+27*sum(y(3:6:end-4))+...
    272*sum(y(4:6:end-3))+27*sum(y(5:6:end-2))+...
    216*sum(y(6:6:end-1))+82*sum(y(7:6:end-6)))*h/140;
end
```

b) Konvergenzrate (Abb. 1):

```
function visualize_weddle_error()
% Mein Name, Aufgabe 1
f=@(x) 1-sqrt(x).*cos(x);      % function
Q_ex=quad(f,0,2,1e-12);        % almost exact value
Q_w=zeros(5,1);                % Quadrature
h=zeros(5,1);                  % meshwidth
n=10.^[1:5];                   % number gridpoints
for i=1:5                      % iteration
    h(i)=(2)/n(i);
    Q_w(i)=weddle_quad(n(i),f);
end
loglog(h,abs(Q_w-Q_ex));       % loglog plot
xlabel('h'); ylabel('error');
print -depsc weddle_rate.eps
% slope
a=diff(log(abs(Q_w-Q_ex)))./diff(log(h))
```

c) Die Ordnung ist ungefähr 1.6, wegen der beschränkten Glattheit des Integranden.

2. a) Das Verfahren ist ein explizites zweistufiges Runge-Kutta-Verfahren mit dem folgenden Butcher-Schema:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2c} & \frac{1}{2c} & 0 \\ \hline & 1-c & c \end{array}$$

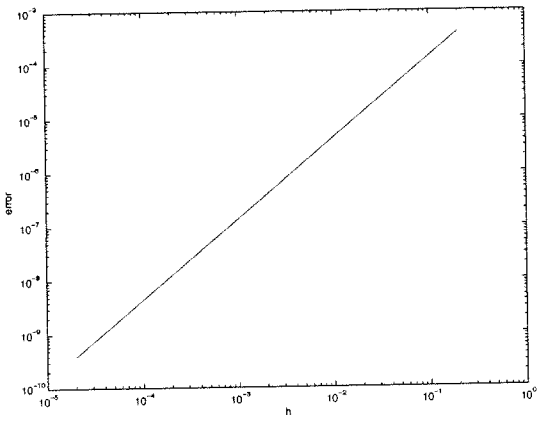

Figure 1: Konvergenzrate für summierte Weddel-Regel

b) MATLAB-Routine für Runge-Kutta aus a):

```
function [t,y]=runge_kutta(t0,y0,T,N,c,f)
% Mein Name, Aufgabe 2
% t0  initial time
% y0  initial value
% T   stop time
% N   Number of gridpoints
% c   Runge-Kutta-Parameter
% F   pointer to function
%
% t   array of time steps
% y   array ot values

t=zeros(N+1,1);       %allocate memory
y=zeros(N+1,1);
h = (T-t0)/N;         %step size
t(1) = t0;            %initial time
y(1) = y0;            %initial value
a=t(1);
for i = 1:N           %iteration
   k1 = h*(1-c)*f(t(i), y(i));
   k2 = h*c*f(t(i)+h/(2*c), y(i)+f(t(i), y(i))*h/(2*c));
   y(i+1) = y(i) + (k1+k2);
   t(i+1) = a + i*h;
end
t(end)
```

c) Der Graph in Abbildung 2 zeigt, dass die Lösung erst bei $N > 20$

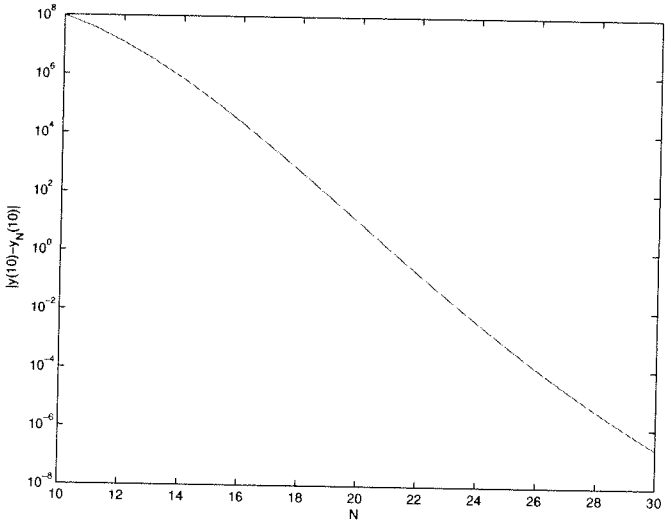

Figure 2: Abbildung zu Runge-Kutta c)

stabil ist. Die entspricht einer Gitterweite $h < 0.2$.

d) Die Graphen in Abbildung 3 zeigen, das der Stabilitätsbereich unabhängig von dem Parameter c ist.

e) Wir wenden das Verfahren auf das Modellproblem $\dot{y} = \lambda y$ mit $\lambda \in \mathbb{C}$ an und erhalten so

$$y_{n+1} = (1 + h\lambda + \frac{1}{2}h^2\lambda^2)y_n = \left(1 + \mu + \frac{\mu^2}{2}\right)^{n+1} y_0$$

für $\mu = h\lambda$. Folglich ist der Verstärkungsfaktor $V(\mu) = 1+\mu+\frac{\mu^2}{2}$ und das Stabilitätsgebiet ist $S = \{\mu \in \mathbb{C} : |1+\mu+\frac{\mu^2}{2}| < 1\}$. Insbesondere gilt $\partial S \cap \mathbb{R} = \{-2, 0\}$ und damit $(-2, 0) \subset S$.

3. a) MATLAB-Routine zur Bestimmung der Koeffizienten:

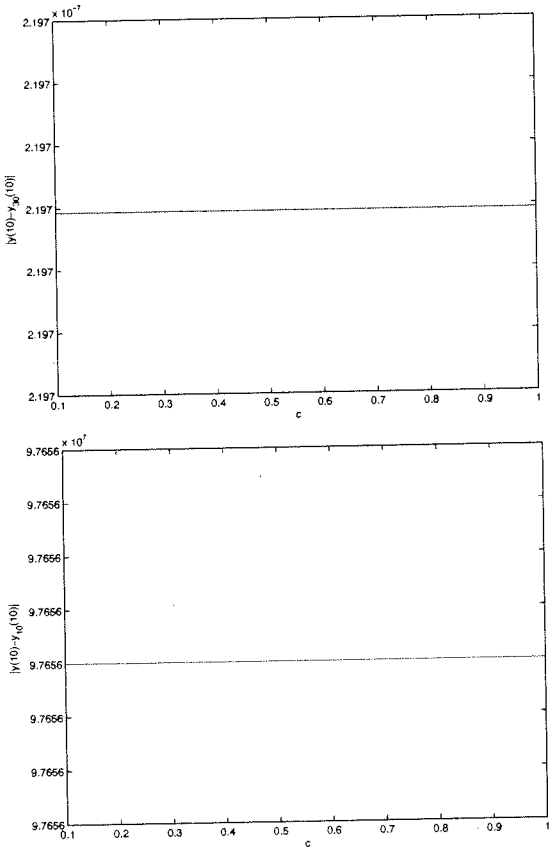

Figure 3: Abbildung zu Runge-Kutta d)

```
function c = pspline_coeff(T,y)
% Mein Name, Aufgabe 3
% T periodicity
% y data
n = length(y);

h = T/n;
e = ones(n,1);
A = spdiags([e 4*e e], -1:1, n, n);

e1 = eye(n,1);
U = [e1,e1(end:-1:1)];
V = [e1(end:-1:1),e1]';

c = 3/h * ( A \ ( y([2:end,1]) - y([end,1:end-1]) ) );

c = c - A \ ( U * ( ( eye(2) + V* ( A \ U ) ) \ V ) * c );
%A(1,n) = 1;
%A(n,1) = 1;
```

b) MATLAB-Routine zur Auswertung:

```
function s = pspline_eval(T,y,c,tt)
% Mein Name, Aufgabe 3
% T periodicity
% y data
% c coefficients
% tt evaluation points
n = length(c);
h = T/n;
c = [c;c(1)];
y = [y;y(1)];

% for-Schleife koennte vermieden werden, ...
% waere aber zu unuebersichtlich
s = zeros(1,length(tt));
for k = 1:length(tt),
   j = min(n,floor(tt(k)/h) + 1);
   tjm1 = (j-1)*h;

   tau = ( tt(k) - tjm1 ) / h;
   s(k) = y(j) * (1-3*tau^2 + 2*tau^3) + ...
       y(j+1) * (3*tau^2 - 2*tau^3) + ...
       h*c(j)*(tau-2*tau^2+tau^3)+...
       h*c(j+1)*(-tau^2+tau^3);
end
```

c) Für $f(t) = sin(t)$ ist $p = 4$ und für $f(t) = |1/2 - t|$ ist $p = 1$.

```
function pspline_sin()
% mein Name, Aufgabe 3
nind = 2:2:100;
err = [];
h=[];
for n = nind,
   y = sin(0:2*pi/n:2*pi);
   h=[h;pi/n];
   c = pspline_coeff(2*pi,y(1:end-1)');
   tt  = 0:2*pi/1000:2*pi;

   err = [err; ...
          max(abs(sin(tt)-...
      pspline_eval(2*pi,y(1:end-1)',c,tt))) ];
end

loglog(nind,err),
%diff(log(err))./diff(log(h))
p=polyfit(log(h),log(err),1);
p(1)
err(25) / err(50)
```

```
function pspline_hat()
% mein Name , Aufgabe 3
nind = 2:2:100;
err = [];
h=[];
for n = nind,
   x = 0:1/n:1;
   h=[h;1/n];
   y = abs(x-0.5);

   c = pspline_coeff(1,y(1:end-1)');
   tt  = 0:1/1000:1;

   err = [err; ...
          max(abs(abs(tt-0.5)-...
      pspline_eval(1,y(1:end-1)',c,tt))) ];
end

plot(pspline_eval(1,y(1:end-1)',c,tt)),

loglog(nind,err),
diff(log(err))./diff(log(h))
p=polyfit(log(h),log(err),1);
p(1)
err(25) / err(50),
```

4. a) Mit $y_1(t) = y(t), \quad y_2(t) = y'(t)$ gilt

$$\begin{aligned} y_1'(t) &= y_2(t), & y_1(0) &= 0.99 \\ y_2'(t) &= -201y_2(t) - 200y_1(t) + 2, & y_2(0) &= 1. \end{aligned}$$

b) Die Jacobimatrix lautet

$$D\boldsymbol{f}(\boldsymbol{y}) = \begin{pmatrix} 0 & 1 \\ -200 & -201 \end{pmatrix}$$

mit charakteristischem Polynom

$$\lambda^2 + 201\lambda + 200 = 0$$

$$\Rightarrow \lambda_{1,2} = -100.5 \pm \frac{\sqrt{40401 - 800}}{2}$$

$$\Rightarrow \lambda_1 = -1 \; \lambda_2 = -200$$

c) Lösungen der beiden Integratoren (Abb. 4): `ode45` benötigt 4875 Schritte, `ode23s` dagegen nur 68.

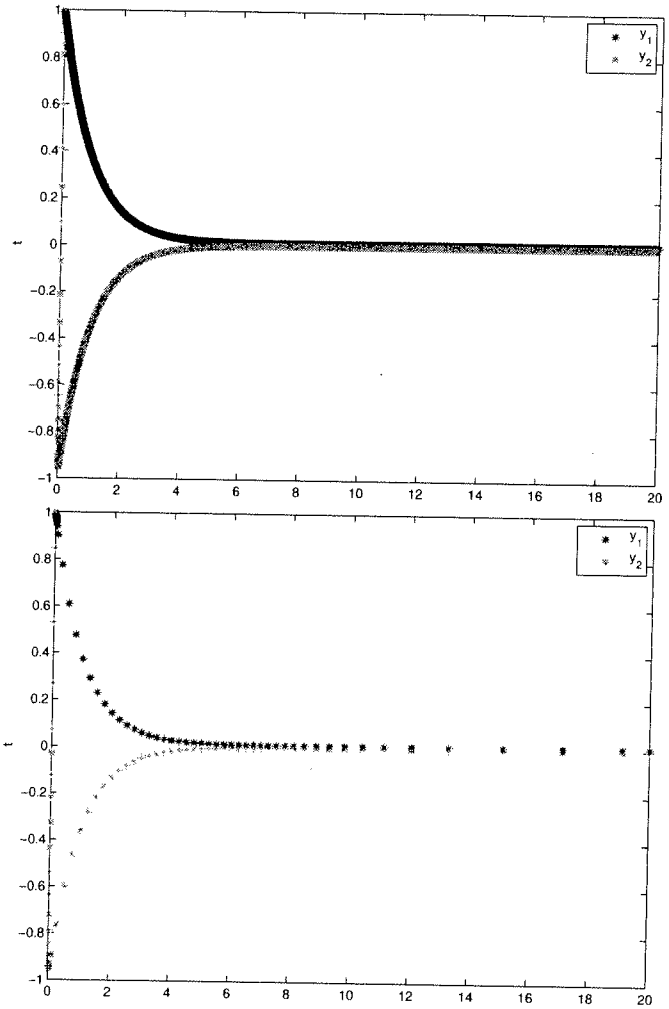

Figure 4: Abbildung zu Anfangswertproblem oben ode45

```
function  AWP()
% Mein Name, Aufgabe 4
f=@(t,y)[y(2); -200*y(1)-201*y(2)+2]; %function

[t,y]=ode45(f,[0 20],[0.99 1]);        %
steps45=size(t,1)
figure;
plot(t,y(:,1),'b*',t,y(:,2),'g*');
legend('y_1','y_2');
ylabel('t');
print -depsc AWP_45.eps

[t,y]=ode23s(f,[0 20],[0.99 1]);
steps23=size(t,1)
figure;
plot(t,y(:,1),'b*',t,y(:,2),'g*');
legend('y_1','y_2');
ylabel('t');
print -depsc AWP_23s.eps
function[f_]=f(t,y)
f_=[y(2);-200*y(1)-201*y(2)+2];
```

Wir haben hier ein steifes Anfangswertproblem und `ode23s` ist ein Integrator für steife Probleme.

5. a) Wir benutzen, das x Fixpunkt ist.

$$\begin{aligned} e^{k+1} &= x^{k+1} - x \\ &= B_\omega x^k + (D - \omega L)^{-1}\omega b - x \\ &= B_\omega x^k + (D - \omega L)^{-1}\omega b - B_\omega x - (D - \omega L)^{-1}\omega b \\ &= B_\omega e^k \end{aligned}$$

b) Potenzmethode:

```
function e=power_iter(A,tol)
% Mein Name, Aufgabe 5
% A Matrix
% tol exit criterium
% e largest eigenvalue
n=size(A,1);
max=1000; k=1;
v_old=zeros(n,1);
v_new=v_old;
v_new(1)=1;
e=0;
while ((k<=max) && (norm(v_old-v_new)>tol))
    k=k+1;
    v_old=v_new;
    v_new=A*v_new;
    e=norm(v_new');
    if (norm(v_new)~=0)
        v_new=v_new/norm(v_new);
    end
end
```

c) Für $\omega \to 2$ und $\omega \to 0$ ρ geht gegen 1 (Abb. 5), d.h. langsame Konvergenz:

```
function visualize_spec_radius()
% Mein Name , Aufgabe 5
%
omega=[0.01:0.01:1.99];
%omega=1;
s=size(omega,2);
e=zeros(1,s);
e2=zeros(1,s);
tol=10^-15;
n=100;
for i=1:s
    d=[ones(n,1) 4*ones(n,1) ones(n,1)];
    L=-diag(ones(n-1,1),-1);
    D=4*diag(ones(n,1),0);
    R=-diag(ones(n-1,1),1);
    B=(D-omega(i)*L)\((1-omega(i))*D+omega(i)*R);
    e(i)=power_iter(B,tol);
    e2(i)=max(abs(eig(B)));
end
figure;
plot(omega,e,'b-',omega,e2,'r--');
xlabel('\omega');
ylabel('\rho(B_{\omega})');
legend('power iteration','matlab eig')
print -depsc sor_radius.eps
```

c) Für $\omega \to 1$ ist der Spektralradius klein, d.h. wir erwarten schnelle Konvergenz. Die Potenzmethode funktioniert nur dann gut, wenn der grösste Eigenwert die Vielfachheit 1 hat und alle anderen Eigenwerte deutlich kleiner sind.

6. Das nichtlineare Ausgleichsproblem lautet

$$min_{C,\lambda} \sum_{i=1}^{6} (f(t_i) - b_i)^2 \tag{1}$$

Die Routine `solver(t,b,l,c,tol)` ist der Gauss-Newton-Algorithmus und verwendet den MATLAB \ als Löser für die linearen Ausgleichprobleme die als Teilproblem auftreten. Er liefert $\lambda = -1.1866$ und $C = 0.5498$.

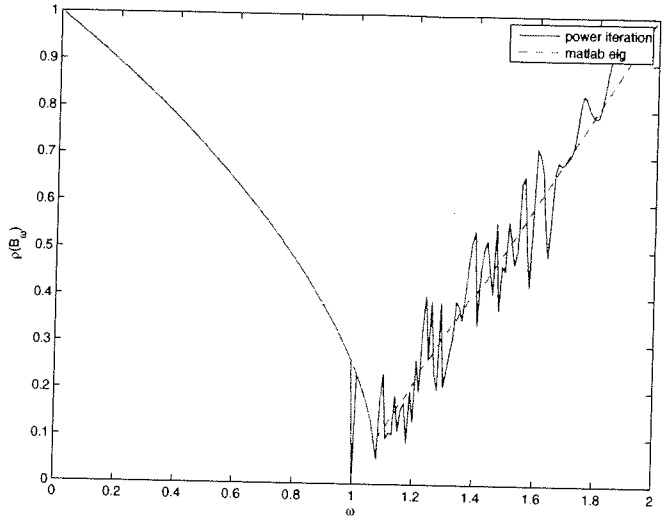

Figure 5: Abbildung zu Anfangswertproblem oben ode45

```
function fit()
% mein name, Aufgabe 6

t=[0.1 0.2 0.3 0.05 0.35 0.25];      % data
b=[0.395 0.134 -0.119 0.6 -0.2 0];   % data
l=1; c=0;                            % inital guesses
tol=10^-10;                          % tolerance

[l,c]=solver(t',b',l,c,tol)          % solver

x=[0:0.01:0.4];                      % plot result
y=c*exp(l*x).*cos(2*pi*x);
plot(x,y);
hold on;
plot(t,b,'r*');                      %plot data
xlabel('t');
ylabel('f(t)');
legend('fit','data');
hold off;
print -depsc fit.eps

function [L,C] = solver(t,b,l,c,tol)
% t  data
% f  data
% l  parameter lambda, initial guess
% c  parameter C, initial guess
% tol exit criterium
%
% [L,C]  solution for lambda and C
f=@(l,c)c*exp(l*t).*cos(2*pi*t)-b;
J=@(l,c)[c*exp(l*t).*cos(2*pi*t).*t exp(l*t).*cos(2*pi*t)];
s = J(l,c)\f(l,c);    %linear least squares solved by \
l=l-s(1);
c=c-s(2);
while (norm(s) > tol)
s = J(l,c)\f(l,c);
s
l= l-s(1);
```