

# Einführung in Matlab - Einheit 6

## Numerische Mathematik, Profiler

Jochen Schulz

Georg-August Universität Göttingen 

## 1 Numerische Mathematik

- Poisson Problem
- Differentialgleichungen
- Lorenz-Gleichungen
- Nichtlineare Gleichungen lösen und Optimierung

## 2 Profiler

## 1 Numerische Mathematik

- Poisson Problem
- Differentialgleichungen
- Lorenz-Gleichungen
- Nichtlineare Gleichungen lösen und Optimierung

## 2 Profiler

# Poisson Problem

- Poisson Problem beschreibt stationäre Wärmeverteilungen.
- *Poisson Problem*: Suche  $u \in C^2(\Omega) \cap C(\overline{\Omega})$  mit

$$\begin{cases} -\Delta u &= f & \text{in } \Omega \\ u &= 0 & \text{auf } \partial\Omega \end{cases}$$

für  $\Omega = (0, 1)^2$  und  $f \in C(\Omega)$ .

- *Laplace-Operator*  $\Delta u := \sum_{i=1}^d \frac{\partial^2 u}{\partial x_i^2}$

- Äquidistante Gitterweite  $h = \frac{1}{N}$ ,  $N \in \mathbb{N}$
- Menge aller Gitterpunkte

$$Z_h := \left\{ (x, y) \in \overline{\Omega} \mid x = z_1 h, y = z_2 h \text{ mit } z_1, z_2 \in \mathbb{Z} \right\}.$$

- Innere Gitterpunkte:  $\omega_h := Z_h \cap \Omega$

- Approximation von  $\frac{\partial^2 u}{\partial x^2}(x, y)$

$$\frac{u(x-h, y) - 2u(x, y) + u(x+h, y)}{h^2} = \frac{\partial^2 u}{\partial x^2}(x, y) + \mathcal{O}(h^2)$$

- Approximation von  $\frac{\partial^2 u}{\partial y^2}(x, y)$

$$\frac{u(x, y-h) - 2u(x, y) + u(x, y+h)}{h^2} = \frac{\partial^2 u}{\partial y^2}(x, y) + \mathcal{O}(h^2)$$

- Addition ergibt für  $\Delta u(x, y)$  die Näherung

$$\frac{1}{h^2} (u(x, y-h) + u(x-h, y) - 4u(x, y) + u(x, y+h) + u(x+h, y))$$

- Definition  $u_{i,j} := u(ih, jh)$  ergibt an Gitterpunkten  $(ih, jh)$

$$-u_{i,j-1} - u_{i-1,j} + 4u_{i,j} - u_{i+1,j} - u_{i,j+1} = h^2 f_{ij}$$

mit  $i, j \in \{1, \dots, N-1\}$  und  $f_{ij} := f(ih, jh)$ .

- Randbedingungen ergeben  $u_{0,i} = u_{N,i} = u_{i,0} = u_{i,N} = 0$ ,  $i = 0, \dots, N$ .

- Lexikografische Sortierung der inneren Unbekannten

$$\begin{array}{cccc} (h, (N-1)h) & (2h, (N-1)h) & \dots & ((N-1)h, (N-1)h) \\ \vdots & \vdots & \vdots & \vdots \\ (h, 2h) & (2h, 2h) & \dots & ((N-1)h, 2h) \\ (h, h), & (2h, h) & \dots & ((N-1)h, h) \end{array}$$

ergibt Vektor  $U_{i+(N-1)(j-1)} = u_{i,j}$ .



Lineares Gleichungssystem für  $U = (U_i)_{i=1}^{(N-1)^2}$

$$AU = F$$

mit

- $F := (f_i)_{i=1}^{(N-1)^2}$  mit  $f_{i+(N-1)(j-1)} = f(ih, jh)$ ,  $i, j \in \{1, \dots, N-1\}$ ,
- 

$$A := \frac{1}{h^2} \text{tridiag}(-I_{N-1}, T, -I_{N-1}) \in \mathbb{R}^{(N-1)^2 \times (N-1)^2},$$

$$T := \text{tridiag}(-1, 4, -1) \in \mathbb{R}^{(N-1) \times (N-1)}.$$

# Implementierung

```
function loes = poisson (f,n)
f = fcnchk(f);
A = gallery('poisson',n-1);
% Erzeuge rechte Seite und Mesh
loes.mesh = zeros(2,(n-1)^2);
F = zeros((n-1)^2,1);
for i = 1:(n-1)
    for j = 1:(n-1)
        F(i+(n-1)*(j-1)) = (1/n)^2*f(i/n,j/n);
        loes.mesh(:,i+(n-1)*(j-1)) = [i/n; j/n];
    end
end
% Loese das lineare System
loes.x = A \ F;
```

```
% Ergaenze Randbedingungen
```

```
loes.x = [ loes.x; zeros(4*(n+1),1)];  
loes.mesh = [loes.mesh, [zeros(1,n+1); 0:1/n:1]];  
loes.mesh = [loes.mesh, [ones(1,n+1); 0:1/n:1]];  
loes.mesh = [loes.mesh, [0:1/n:1; ones(1,n+1)]];  
loes.mesh = [loes.mesh, [0:1/n:1; zeros(1,n+1)]];
```

```
% Plotten
```

```
plot3(loes.mesh(1,:),loes.mesh(2,:),loes.x,'*');  
figure;  
[X,Y] = meshgrid(0:1/n:1,0:1/n:1);  
Fi = TriScatteredInterp(loes.mesh(1,:)', loes.mesh(2,:)',  
    loes.x,'linear');  
Z = Fi(X,Y);  
surf(X,Y,Z); shading flat;
```

## 1 Numerische Mathematik

- Poisson Problem
- Differentialgleichungen
- Lorenz-Gleichungen
- Nichtlineare Gleichungen lösen und Optimierung

## 2 Profiler

# Gewöhnliche Differentialgleichungen

Sei  $I \subset \mathbb{R}$  ein Intervall. Bei einer gewöhnlichen Dgl. sucht man eine Funktion  $y: I \rightarrow \mathbb{R}^n$ , so dass

$$\frac{d}{dt}y(t) = f(t, y(t)), t \in I \quad y(t_0) = y_0,$$

wobei  $y_0 \in \mathbb{R}^n$  ein vorgegebener Anfangswert an  $t_0 \in I$  und  $f: I \times \mathbb{R}^n \rightarrow \mathbb{R}^n$  die rechte Seite ist. Außerdem sei

$$\frac{d}{dt}y(t) := \left( \frac{\partial y_1(t)}{\partial t}, \dots, \frac{\partial y_n(t)}{\partial t} \right)^t.$$

**Beispiele:**

$$\frac{d}{dt}y(t) = y(t), \quad y(t_0) = y_0, \quad \text{Lösung: } y(t) = y_0 e^{t-t_0}$$

$$\frac{d}{dt}y(t) = e^y \sin(t), \quad \text{Lösung: } y(t) = -\log(\cos(x) + C), \quad C + \cos(x) > 0$$

# Skalares Beispiel

Löse für  $0 \leq t \leq 3$  mit `ode45` die Dgl.

$$\frac{d}{dt}y(t) = -y(t) - 5e^{-t}\sin 5t, \quad y(0) = 1.$$

- Die rechte Seite als eigene Funktion:

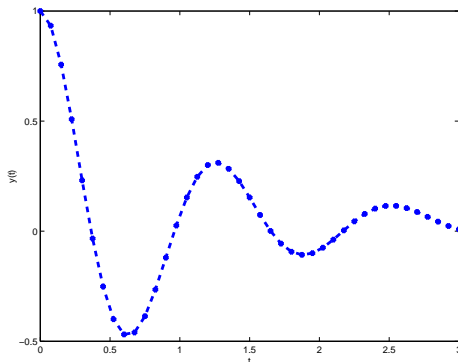
```
function z = rechte_seite1(t,y)
% rechte_seite1    ODE Beispiel
%                z  =rechte_seite1(t,y)

z = -y -5*exp(-t)*sin(5*t);
```

# Skalares Beispiel

- Ausrechnen und Plotten

```
tspan = [0,3]; aw = 1;  
[t,y] = ode45(@rechte_seite1,tspan,aw);  
plot(t,y,'*--','Linewidth',3)  
xlabel('t'), ylabel('y(t)')
```



```
[<t>,<y>] = ode45(<@fun>, <tspan>, <aw>, <options>)
```

- **@fun** steht für die rechte Seite der Dgl. (m-File).
- $aw \in \mathbb{R}^n$  ist der Anfangswert.
- **tspan** gibt das Zeitintervall an, auf dem die Dgl. berechnet werden soll. Normalerweise ist es von der Form **tspan**=[t\_0, t\_1]. Dann wird die Dgl. auf dem Intervall  $[t_0, t_1]$  berechnet (Anfangswert:  $y(t_0) = aw$ ).
- Rückgabewerte: Vektoren  $t$  und Matrizen  $y$ . Dabei ist  $y(:, i)$  die Lösung an der Stelle  $t(i)$ . Die Punkte  $t_i$  werden automatisch bestimmt.
- Durch die optionale Angabe von **options** kann der Löser gezielt eingestellt werden.
- Spezifiziert man mehr als zwei Zeitpunkte in **tspan**, so gibt MATLAB die Lösung genau an diesen Zeitschritten zurück.



Die genauen Parameter der ODE-Löser können durch

```
options = odeset('Eigenschaft 1','Spez. 1',...  
                'Eigenschaft 2','Spez. 2',...)
```

gesteuert werden. Die wichtigsten Parameter sind **AbsTol** (Default  $10^{-6}$ ) und **RelTol** (Default:  $10^{-3}$ ).

Beispiel:

```
options = odeset('AbsTol',1e-7,'RelTol',1e-4)
```

Löser	Steifigkeit	Algorithmus	Ordnungen
ode45	nicht steif	Expliziter Runge-Kutta Löser	4, 5
ode23	nicht steif	Expliziter Runge-Kutta Löser	2, 3
ode113	nicht steif	Explizites Mehrschrittverfahren	1 - 13
ode15s	steif	Implizites Mehrschrittverfahren	1 - 5
ode23s	steif	Modifiziertes Rosenbrockverfahren	2, 3
ode23t	mittel steif	implizite Trapez Regel	2, 3
ode23tb	steif	Implizites Runge-Kutta Verf.	2, 3

## 1 Numerische Mathematik

- Poisson Problem
- Differentialgleichungen
- Lorenz-Gleichungen
- Nichtlineare Gleichungen lösen und Optimierung

## 2 Profiler

# Die Lorenz-Gleichungen

- Chaostheorie / Schmetterlingseffekt.

$$\frac{d}{dt}y_1(t) = 10(y_2(t) - y_1(t))$$

$$\frac{d}{dt}y_2(t) = 28y_1(t) - y_2(t) - y_1(t)y_3(t)$$

$$\frac{d}{dt}y_3(t) = y_1(t)y_2(t) - 8y_3(t)/3$$

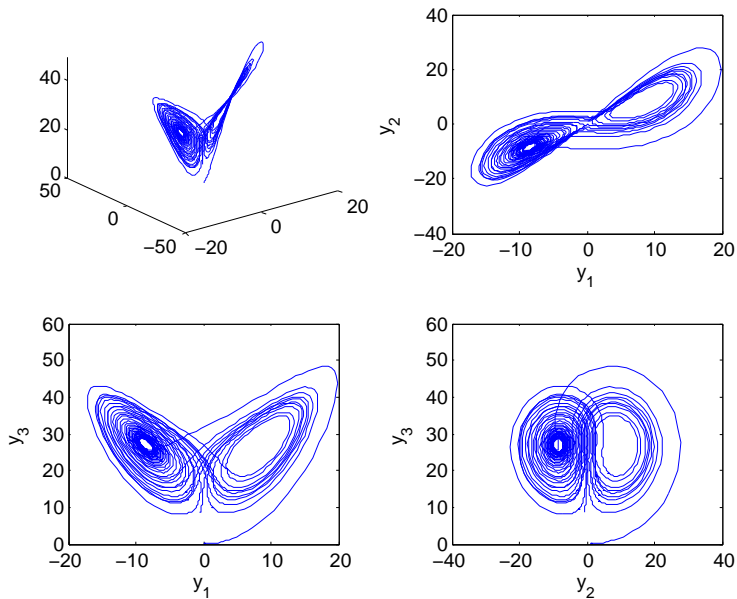
rechte Seite:

```
function z=lorenz_rechte_seite(t,y)
z=      [10*(y(2)-y(1)); ...
        28*y(1)-y(2)-y(1)*y(3); ...
        y(1)*y(2)-8*y(3)/3];
```

# Die Lorenz-Gleichungen

```
%-----  
%   lorenz_gl.m  
%   Eine Approximation der Lorenzgleichungen  
%   Gerd Rapin 08.01.2004  
%-----  
tspan = [0,30]; aw = [0;1;0];  
options = odeset ('AbsTol',1e-7,'RelTol',1e-4);  
[t,y] = ode45(@lorenz_rechte_seite,tspan,aw, options);  
subplot(2,2,1),plot3(y(:,1),y(:,2),y(:,3)),  
subplot(2,2,2),plot(y(:,1),y(:,2)), xlabel('y_1'), ylabel  
('y_2');  
subplot(2,2,3),plot(y(:,1),y(:,3)), xlabel('y_1'), ylabel  
('y_3');  
subplot(2,2,4),plot(y(:,2),y(:,3)), xlabel('y_2'), ylabel  
('y_3');
```

# Die Lorenz-Gleichungen



## 1 Numerische Mathematik

- Poisson Problem
- Differentialgleichungen
- Lorenz-Gleichungen
- Nichtlineare Gleichungen lösen und Optimierung

## 2 Profiler

Löst nichtlineare Gleichungen

```
[x,fval] = fsolve(fun,x0,options)
```

- `fun`: Function handle.
- `x0`: Startvektor.
- `options`: Struktur der options (siehe `optimset`).



```
options = optimset('param1',value1,'param2',value2,...)
```

- 'TolFun': Abbruchkriterium für den Funktionswert.
- 'TolX': Abbruchkriterium für aktuellen Punkt  $x$ .
- 'MaxIter': Maximale Anzahl Iterationen.
- 'Display': Regelt Ausgabe im Commandwindow.
- 'Algorithm': Wahl des genutzten Algorithmus.

# Beispiel

```
x0 = -5; % Startwert
options = optimset('Display','iter');
f = @(x) 2*x - exp(-x);
[x,fval] = fsolve(f,x0,options)
```

x =

0.351733710993003

fval =

-6.926083040426079e-10

# Ableitungsfrei / Nebenbedingungen

ohne Nebenbedingungen, multidimensional:

```
[x,fval,exitflag,output] = fminsearch(fun,x0,options)
```

- `fun`: Function handle.
- `x0`: Start-wert/vektor/matrix.
- `options`: Struktur der options (siehe `optimset`).

mit Nebenbedingungen, multidimensional:

```
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

Das Newtonverfahren ist ein Verfahren zum Lösen von Gleichungen mit stetig, differenzierbaren Funktionen  $f: \mathbb{R}^n \mapsto \mathbb{R}^n$

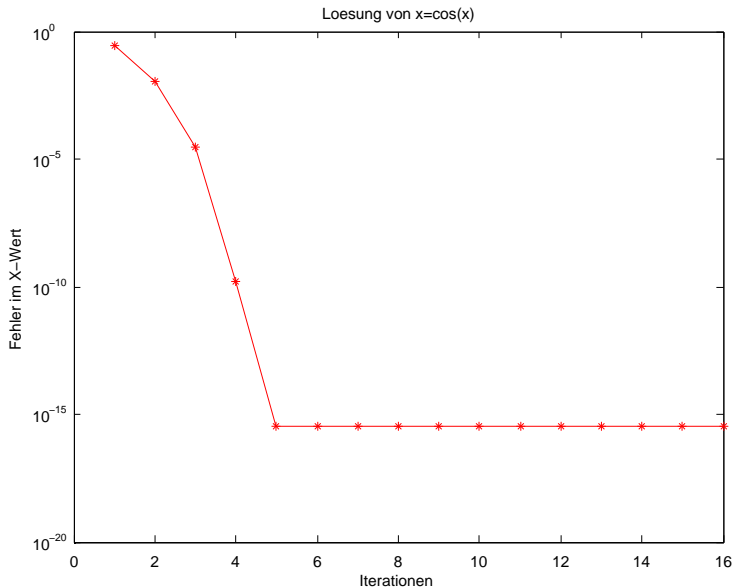
**1D-Fall:**

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

# Newtonverfahren - Implementation

```
% Programm zur Loesung von  $x=\cos(x)$ 
xn = 1; % Startwert Newton
xr = 0.739085133215161; % wahre Loesung
xnlist = xn-xr;% Fehlerliste
for i=1:15
    xn = xn-(xn-cos(xn))/(1+sin(xn)); % Newton
    xnlist = [xnlist xn-xr];
end
% Listenausgabe
format long
xnlist'
% Plotausgabe
semilogy(1:length(xnlist),abs(xnlist),'r*-')
title('Loesung von  $x=\cos(x)$ ')
xlabel('Iterationen')
ylabel('Fehler im X-Wert')
```

# Newtonverfahren - Konvergenz

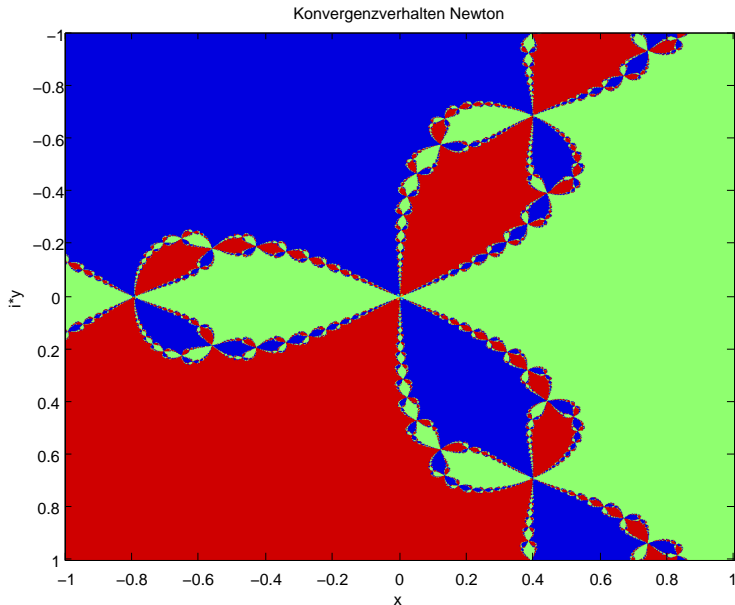


# Newtonverfahren - Konvergenz

```
n = 1000;
x = linspace(-1,1,n);
[X,Y] = meshgrid(x,x);
Z = complex(X,Y);
TOL = 0.2;
V = zeros(n,n);
for idx = 1:20
    Z = Z - (Z.^3-1)./(3*Z.^2);
    ind = find (Z.^3 - 1 < TOL);
    V(ind) = angle(Z(ind));
end
image(x,x,V,'CDataMapping','scaled');
caxis([-2.5 2.5]);
title('Konvergenzverhalten Newton')
xlabel('x'),ylabel('i*y')

roots([1 0 0 -1])
```

# Newtonverfahren - Konvergenz





## 1 Numerische Mathematik

- Poisson Problem
- Differentialgleichungen
- Lorenz-Gleichungen
- Nichtlineare Gleichungen lösen und Optimierung

## 2 Profiler

Performance ist bei realen Problemen schnell ein wichtiger Aspekt um Software zu erhalten die das gegebene Problem gut und schnell löst.

## **Einflussfaktoren:**

- Algorithmus
- **Implementation**
- Betriebssystem und Programmiersprache
- Hardware

Tool um Bottlenecks und Fehler (Bugs) herauszufinden.

## Features

- (zeilenweise) Ausführungszeit
- Eltern-Kind-Beziehungen der Funktionen und deren Ausführungszeit
- Anzahl Funktionsaufrufe
- Geschichte (History) der Funktionsaufrufe
- Farbige Darstellung und Frontend

```
profile on  
<Befehle>  
profile viewer
```

Optionen und Kommandos:

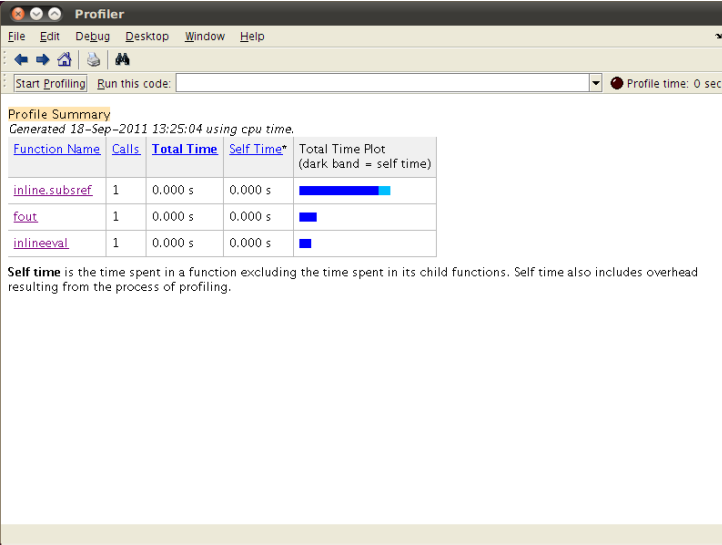
- `-history`: Speichert die Funktionsaufruf-history.
- `-timer real`: Setzt die Zeitmessung auf Realzeit.
- `-timer cpu`: Setzt die Zeitmessung auf CPU-Zeit.
- `resume`: Setzt den Profiler fort.

# Profiler - Beispiel

```
function res = fout(x)
res = exp(x).*x.^2;
```




```
% profile inline-function vs. native function
x = linspace(1,100,1000000);
fin = inline('exp(x).*x.^2');
profile on -history -timer real
fin(x);
fout(x);
profile viewer
p = profile('info');
```

# Profilerwindow



The screenshot shows a window titled "Profiler" with a menu bar (File, Edit, Debug, Desktop, Window, Help) and a toolbar with icons for navigation and execution. Below the toolbar is a status bar with "Start Profiling" and "Run this code:" buttons, and a "Profile time: 0 sec" indicator.

**Profile Summary**  
Generated 18-Sep-2011 13:25:04 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
<a href="#">inline.subsref</a>	1	0.000 s	0.000 s	
<a href="#">fout</a>	1	0.000 s	0.000 s	
<a href="#">inlineeval</a>	1	0.000 s	0.000 s	

**Self time** is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

# History der Funktionsaufrufe

```
% profile inline-function vs. native function
x = linspace(1,100,100000);
fin = inline('exp(x).*x.^2');
profile on -history
fin(x);
fout(x);
profile viewer
p = profile('info');
for n = 1:size(p.FunctionHistory,2)
    if p.FunctionHistory(1,n)==0
        str = 'entering function: ';
    else
        str = 'exiting function: ';
    end
    disp([str p.FunctionTable(p.FunctionHistory(2,n))
        .FunctionName])
end
```

# Profiling Poisson

```
profile on
poisson(@(x,y) x*y.^4,100)
profile viewer
```

