Einführung in Matlab und Python - Einheit 6

Numerische Mathematik, Profiler

Jochen Schulz

Georg-August Universität Göttingen

Aufbau

1. **Numerische Mathematik**
   - Poisson Problem
   - Differentialgleichungen
   - Lorenz-Gleichungen
   - Nichtlineare Gleichungen lösen und Optimierung

2. **Profiler**
   - Matlab
   - Python

Aufbau

1. **Numerische Mathematik**
   - Poisson Problem
   - Differentialgleichungen
   - Lorenz-Gleichungen
   - Nichtlineare Gleichungen lösen und Optimierung
2. **Profiler**
   - Matlab
   - Python

Poisson Problem

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

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

  für $\Omega = (0,1)^2$ und $f \in C(\Omega)$.
- *Laplace-Operator* $\triangle u := \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2}$

Diskretisierung

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

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

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

Diskretisierung

- 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 $\triangle u(x, y)$ die Näherung

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

Diskretisierung

- 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 $i, j \in \{1, \ldots, N-1\}$ und $f_{ij} := f(ih, jh)$.
- Randbedingungen ergeben $u_{0,i} = u_{N,i} = u_{i,0} = u_{i,N} = 0$, $i = 0, \ldots, N$.

Diskretisierung

- 2D-Werte von u in einen Vektor speichern: Lexikografische Sortierung der inneren Unbekannten

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

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

Diskretisierung

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

$$AU = F$$

mit

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

$$\begin{aligned} A &:= \frac{1}{h^2} tridiag(-I_{N-1}, T, -I_{N-1}) \in \mathbb{R}^{(N-1)^2 \times (N-1)^2}, \\ T &:= tridiag(-1, 4, -1) \in \mathbb{R}^{(N-1)\times(N-1)}. \end{aligned}$$

Matlab: 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;
```

Matlab: Implementierung

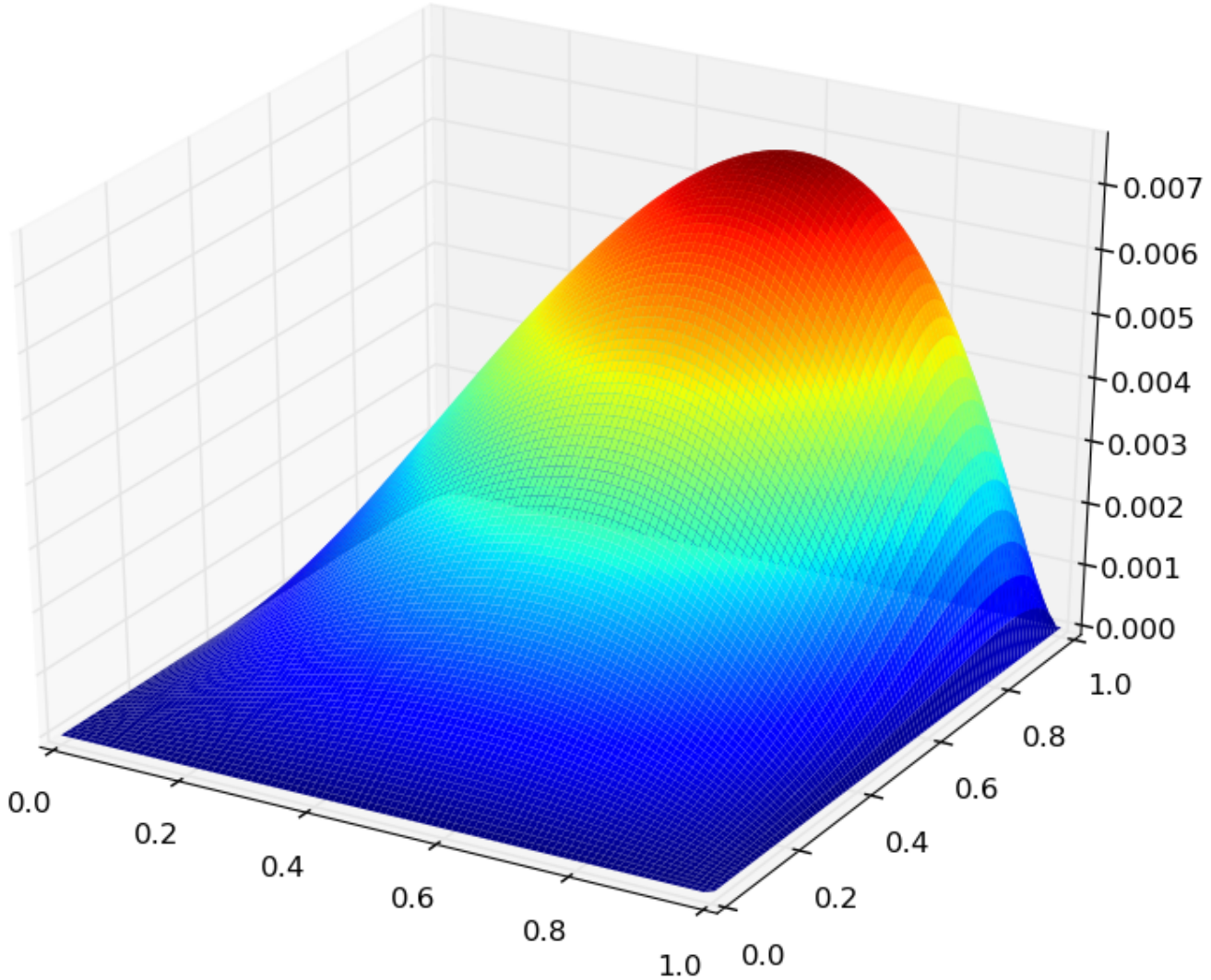
```
% Ergaenze Randbedingungen
loes.x = [ loes.x; zeros(4*(n+1),1)];
loes.mesh = [loes.mesh, [zeros(1,n+1); 0:1/n:1]];
loes.mesh = [loes.mesh, [ones(1,n+1);  0:1/n:1]];
loes.mesh = [loes.mesh, [0:1/n:1; ones(1,n+1)]];
loes.mesh = [loes.mesh, [0:1/n:1; zeros(1,n+1)]];

% 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;
```

Python: Implementierung

```
def poisson(f,n):
    x,h = linspace(0,1,n+1,retstep=True)
    A = poissonmatrix(n-1,1.)
    mesh = zeros((2,(n-1)**2))
    F = zeros((n-1)**2)
    for i in range(0,(n-1)):
        for j in range(0,(n-1)):
            F[i+(n-1)*j] = f(i/float(n),j/float(n))
            mesh[:,i+(n-1)*j] = [i/float(n),j/float(n)]
    loes = solve(A.todense(),F)
    loes = hstack( (loes, zeros(4*n) ) )
    mesh = hstack( (mesh, vstack((zeros(n), x[1:])) ) )
    mesh = hstack( (mesh, vstack((ones(n), x[1:])) ) )
    mesh = hstack( (mesh, vstack((x[:-1], ones(n)) ) ) )
    mesh = hstack( (mesh, vstack((x[:-1], zeros(n)) ) ) )
    fig=figure(), ax = Axes3D(fig)
    ax.plot(mesh[0,:],mesh[1,:],loes,'*')
    fig=figure(), ax = Axes3D(fig)
```

Poisson-Lösung



Aufbau

1. **Numerische Mathematik**
   - Poisson Problem
   - Differentialgleichungen
   - Lorenz-Gleichungen
   - Nichtlineare Gleichungen lösen und Optimierung
2. **Profiler**
   - Matlab
   - Python

Gewöhnliche Differentialgleichungen

Sei $I \subset \mathbb{R}$ ein Intervall. Bei einer gewöhnlichen Dgl. sucht man eine Funktion $y : I \longrightarrow \mathbb{R}^n$, 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 $f : I \times \mathbb{R}^n \longrightarrow \mathbb{R}^n$ die rechte Seite ist. Außerdem sei $\frac{d}{dt}y(t) := (\frac{\partial y_1(t)}{\partial t}, \ldots, \frac{\partial y_n(t)}{\partial t})^t$.

Beispiele:

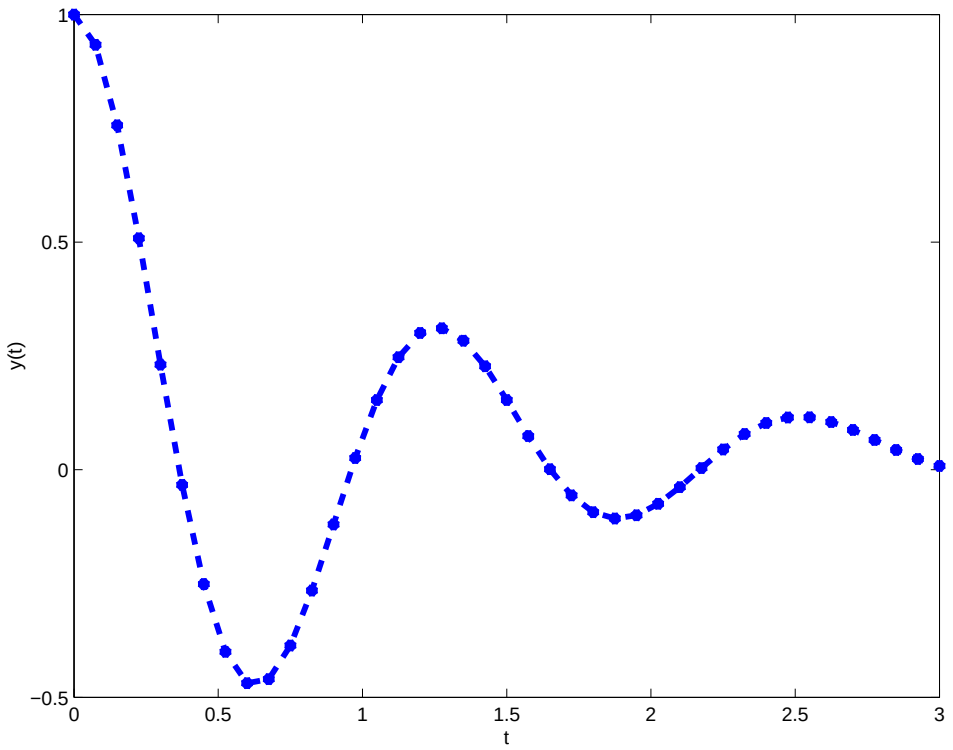
$\frac{d}{dt}y(t) = y(t),\ y(t_0) = y_0,$ Lösung: $y(t) = y_0 e^{t-t_0}$

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

Skalares Beispiel

Löse für $0 \leq t \leq 3$ die Dgl.

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



Matlab: Skalares Beispiel

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);
```

Ausrechnen und Plotten

```
tspan = [0,3]; aw = 1;
[t,y] = ode45(@rechte_seite1,tspan,aw);
plot(t,y,'r*--','Linewidth',3,'MarkerSize',15)
xlabel('t'), ylabel('y(t)')
```

Python: Skalares Beispiel

```
from scipy.integrate import odeint
def rechte_seite1(y,t):
    """rechte_seite1   ODE Beispiel
       z  =rechte_seite1(t,y)"""
    return -y -5*exp(-t)*sin(5*t)

tspan =  linspace(0,3,20)
aw = 1
y = odeint(rechte_seite1,aw,tspan,printmessg=True)
plot(tspan,y,'r*--',linewidth=3,markersize=15)
xlabel('t'), ylabel('y(t)')
```

Matlab: ode45 - explizites Runge-Kutta

```
[<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.
- `options`: optionale Parameter des Lösers.
- Spezifiziert man mehr als zwei Zeitpunkte in `tspan`, so gibt MATLAB die Lösung genau an diesen Zeitschritten zurück.

Matlab: Optionen

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)
```

Python: scipy.integrate.ode - verschiedene Löser

```
<r> = scipy.integrate.ode(<f>[,<jac>])
```

- f: Rechte Seite der DGL: $y'(t) = f(t, y)$
- jac: (optionale) Jacobi-Matrix
- r.set_integrator (<name>[,<params>]): Setzt den zu nutzenden Löser <name> mit den Parametern <params>.
- r.set_initial_value (y[, t]): Setzt den Anfangswert.
- r.integrate (t): Findet $y(t)$ und setzt den neuen Anfangswert im Löser.
- r.succesful (): Wahrheitswert über Erfolg der Integration.

Aufbau

1. **Numerische Mathematik**
   - Poisson Problem
   - Differentialgleichungen
   - Lorenz-Gleichungen
   - Nichtlineare Gleichungen lösen und Optimierung
2. **Profiler**
   - Matlab
   - Python

Die Lorenz-Gleichungen

- Chaostheorie / Schmetterlingseffekt.

$$
\begin{aligned}
\frac{d}{dt}y_1(t) &= 10(y_2(t) - y_1(t)) \\
\frac{d}{dt}y_2(t) &= 28y_1(t) - y_2(t) - y_1(t)y_3(t) \\
\frac{d}{dt}y_3(t) &= y_1(t)y_2(t) - 8y_3(t)/3
\end{aligned}
$$

Matlab: Die Lorenz-Gleichungen

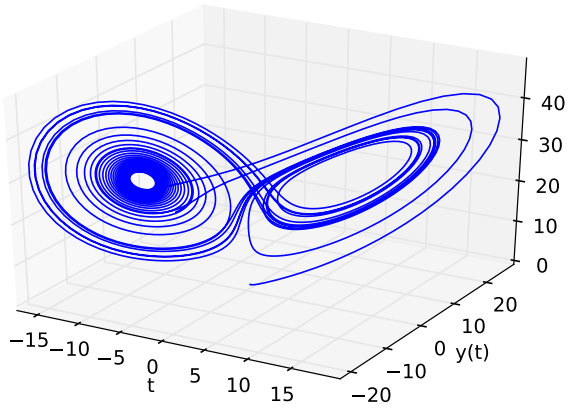
```
function z=lorenz_rechte_seite(t,y)
z=      [10*(y(2)-y(1));...
         28*y(1)-y(2)-y(1)*y(3);...
         y(1)*y(2)-8*y(3)/3];
```

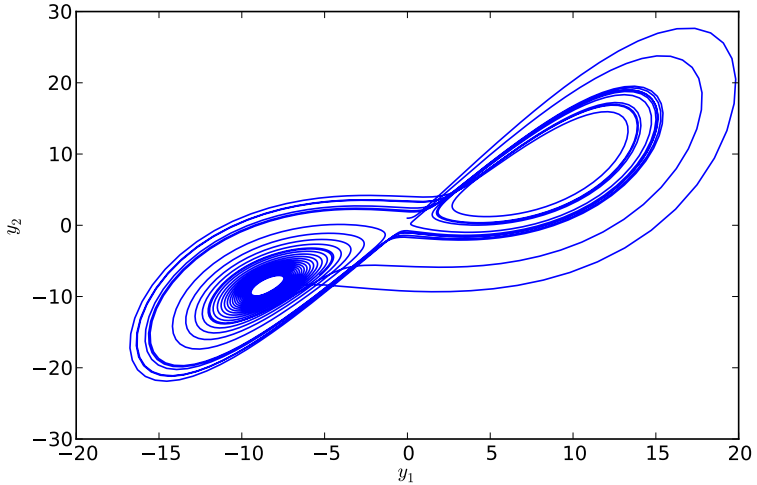
```
%  Eine Approximation der Lorenzgleichungen
%  Gerd Rapin 08.01.2004
tspan = [0,30]; aw = [0;1;0];
options = odeset ('AbsTol',1e-7,'RelTol',1e-4);
[t,y] = ode45(@lorenz_rechte_seite,tspan,aw, options);
subplot(2,2,1),plot3(y(:,1),y(:,2),y(:,3)),
subplot(2,2,2),plot(y(:,1),y(:,2)), xlabel('y_1'),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

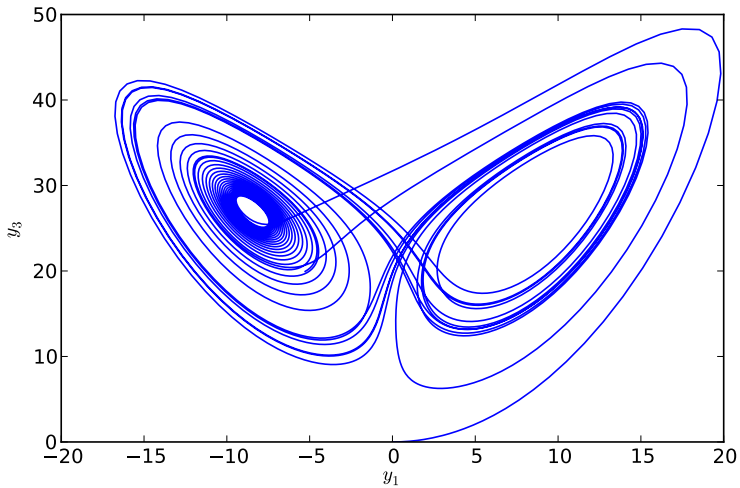
Python: Die Lorenz-Gleichungen

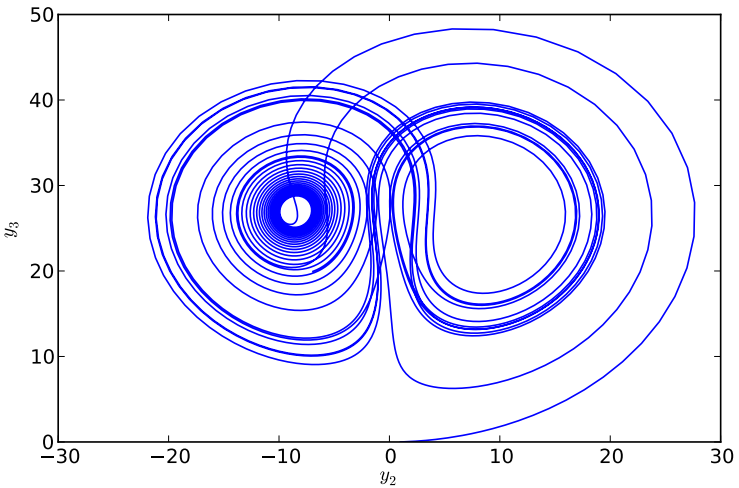
```
def lorenz_rhs(t,y):
    return array([[10*(y[1]-y[0])], [28*y[0]-y[1]-y[0]*y
        [2]], [y[0]*y[1]-8*y[2]/3]])
y = array([0,1,0])
r = ode(lorenz_rhs)
r.set_initial_value(y, 0)
r.set_integrator('dopri5',atol=1e-7,rtol=1e-4)
tmax = 30,dt = 0.01,t=[]
while r.successful() and r.t < tmax:
    r.integrate(r.t+dt)
    t.append(r.t)
    y = vstack( (y, r.y) )
fig = figure(figsize=(16,10))
ax = fig.add_subplot(2, 2, 1, projection='3d')
ax.plot(y[:,0],y[:,1],y[:,2]),xlabel('t'), ylabel('y(t)')
subplot(2,2,2),plot(y[:,0],y[:,1]), xlabel('y_1')
subplot(2,2,3),plot(y[:,0],y[:,2]), xlabel('y_1')
subplot(2,2,4),plot(y[:,1],y[:,2]), xlabel('y_2')
```

Die Lorenz-Gleichungen









Aufbau

1. **Numerische Mathematik**
   - Poisson Problem
   - Differentialgleichungen
   - Lorenz-Gleichungen
   - Nichtlineare Gleichungen lösen und Optimierung

2. **Profiler**
   - Matlab
   - Python

Nichtlinearer Gleichungslöser

Löst nichtlineare Gleichungen $F(x) = 0$

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

```
from scipy import optimize
xres,info,i,mesg = optimize.fsolve(fun,x0,options)
```

Input:

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

Output

- xres: Ergebnis.
- fval| info['fvec']: Funktionsauswertung $f(xres)$.
- options: Liste benannter Argumente.

optimset und options

```
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.

```
options
```

- xtol: Abbruchkriterium für aktuellen Punkt x.
- full_output: Schalter für detaillierte Ausgabe.
- maxfev: maximale Anzahl Funktionsaufrüfe.

Matlab: Beispiel

```
x0 = -5;  % Startwert
options = optimset('Display','iter','TolFun',10^-5);
f = @(x) 2*x  - exp(-x);
[x,fval] = fsolve(f,x0,options)
```

```
x =

    0.351733710993003

fval =

    -6.926083040426079e-10
```

Python: Beispiel

```
from scipy import optimize
x0 = -5  # Startwert
f = lambda x: 2*x - exp(-x)
res,info,i,mesg = optimize.fsolve(f,x0,xtol=1e-5,
    full_output=True)
print ("res: {} \nnfev: {} \nfvec: {}".format(res,info['
    nfev'],info['fvec']) )
```

```
res: [ 0.35173371]
nfev: 13
fvec: [ -1.50035540e-12]
```

Matlab: Ableitungsfrei / Nebenbedingungen

Finde Minimum ...
ohne Nebenbedingungen, multidimensional (Nelder-Mead-Simplex):

```
[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)
```

Python: Ableitungsfrei / Nebenbedingungen

Finde Minimum ...
ohne Nebenbedingungen, multidimensional (Nelder-Mead-Simplex):

```
fmin(func, x0, args=(), xtol=0.0001, ftol=0.0001, maxiter
    =None, maxfun=None, full_output=0, disp=1, retall=0,
    callback=None)
```

- func: Function handle
- x0: Start-wert/-vektor
- xtol, ftol: Abbruch-Toleranz in x und *func*.

mit Nebenbedingungen, multidimensional:

```
fminbound(func, x1, x2, args=(), xtol=1e-05, maxfun=500,
    full_output=0, disp=1)
```

Newtonverfahren

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

1D-Fall:

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

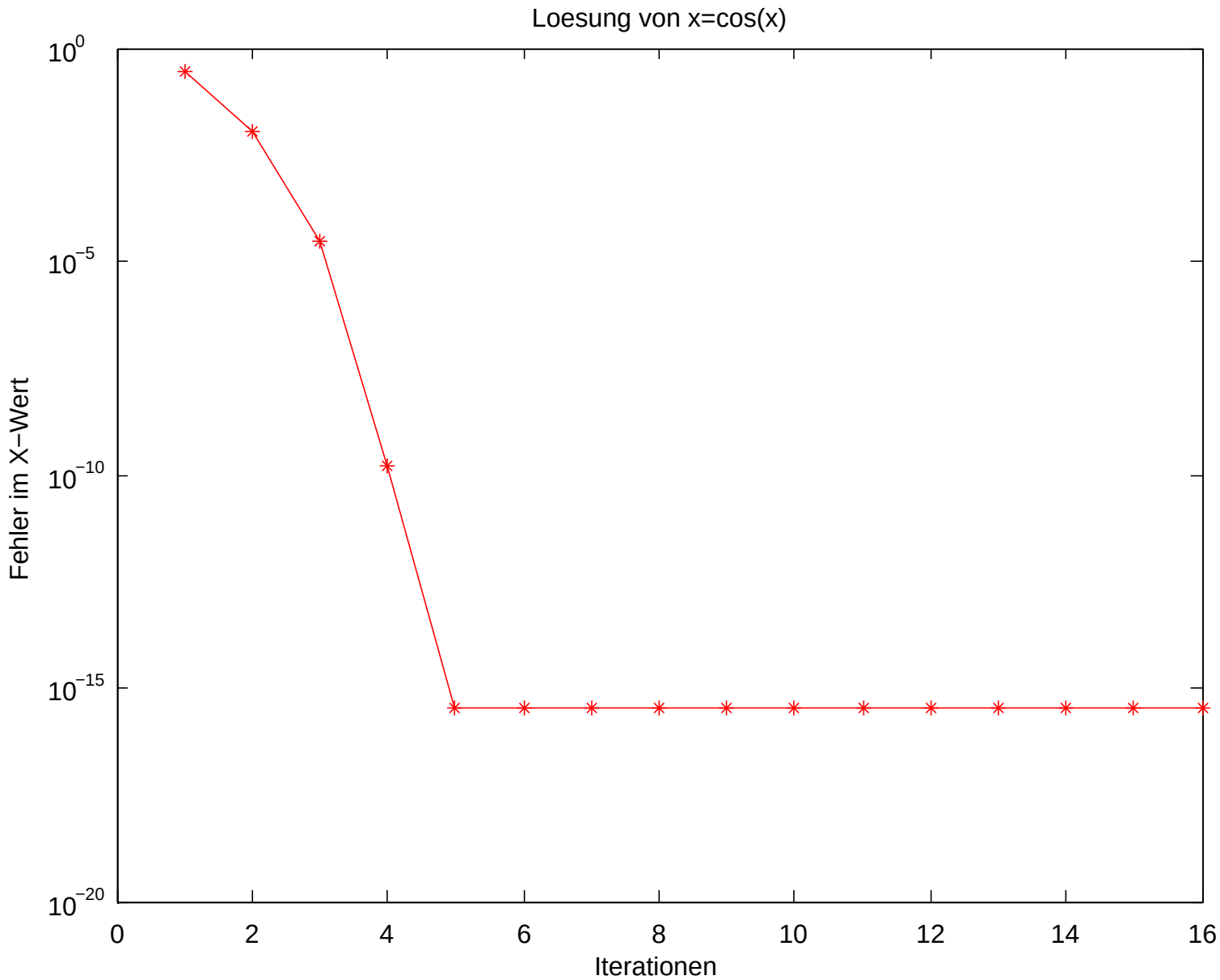
Matlab: Newtonverfahren - Implementation

```
% 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')
```

Python: Newtonverfahren - Implementation

```
def newton_example():
    # Programm zur Loesung von x=cos(x)
    xn = 1 # Startwert Newton
    xr = 0.739085133215161 # wahre Loesung
    xnlist = xn-xr # Fehlerliste
    for i in range(0,15):
        xn = xn-(xn-cos(xn))/(1+sin(xn)) # Newton
        xnlist = hstack((xnlist, xn-xr))
    print xnlist
    #Plotausgabe
    semilogy(range(1,len(xnlist)+1),abs(xnlist),'r*-')
    title('Loesung von x=cos(x)')
    xlabel('Iterationen')
    ylabel('Fehler im X-Wert')
```

Newtonverfahren - Konvergenz



Matlab: Newtonverfahren - Konvergenz

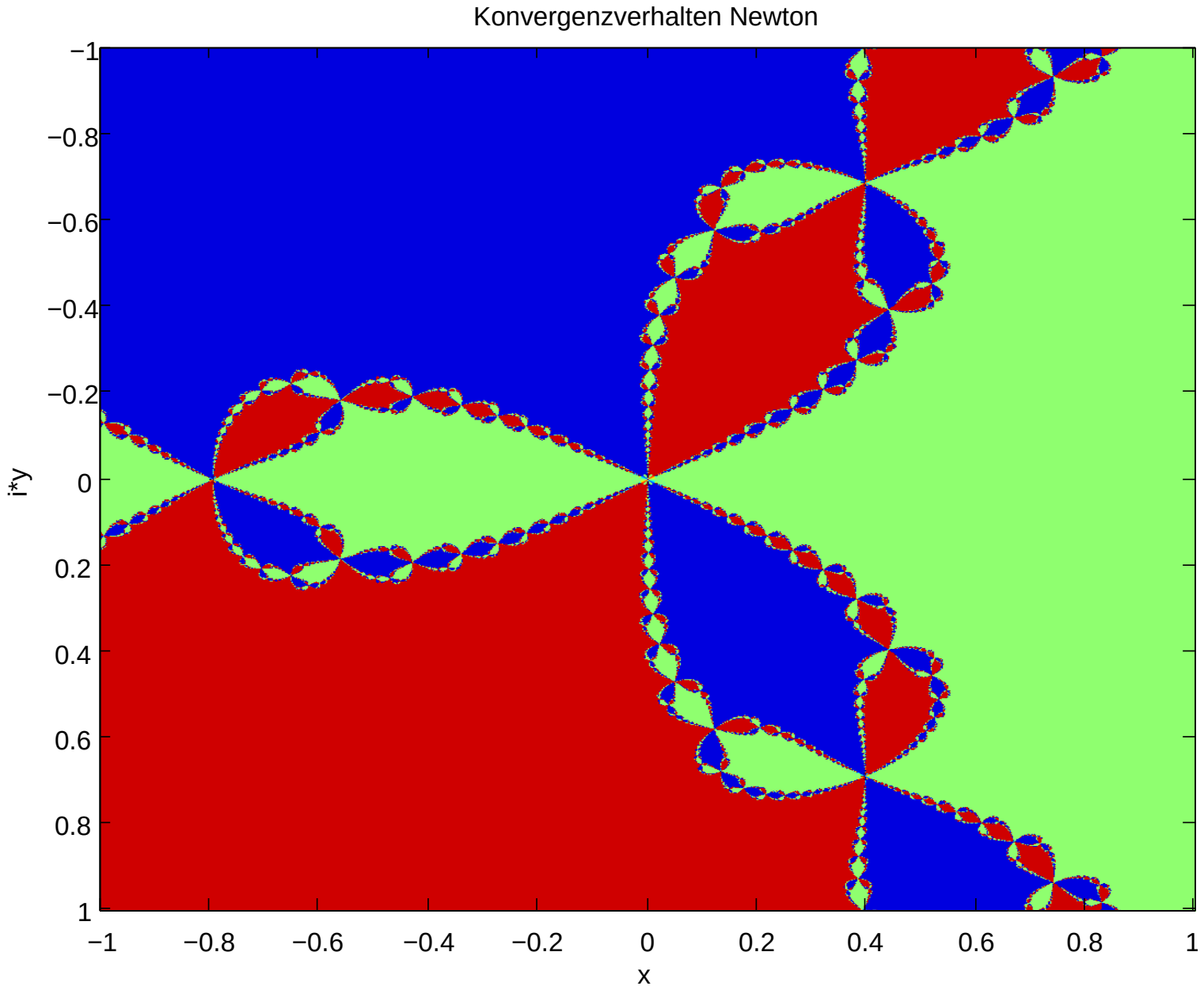
```
n = 1000;
x = linspace(-1,1,n);
[X,Y] = meshgrid(x,x);
Z = complex(X,Y);
TOL = 0.2;
V = zeros(n,n);
for idx = 1:20
    Z = Z - (Z.^3-1)./(3*Z.^2);
    ind = find (Z.^3 - 1 < TOL);
    V(ind) = angle(Z(ind));
end
image(x,x,V,'CDataMapping','scaled');
caxis([-2.5 2.5]);
title('Konvergenzverhalten Newton')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

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

Python: Newtonverfahren - Konvergenz

```
def newton_konvergenz():
    n = 1000
    x = linspace(-1,1,n)
    [X,Y] = meshgrid(x,x)
    Z = X + 1j*Y
    TOL = 0.02
    V = zeros((n,n))
    for idx in range(0,20):
        Z = Z - (Z**3-1)/(3*Z**2)
        ind = find (Z**3 - 1 < TOL)
        V.ravel()[ind] = angle(Z.ravel()[ind])
    figure()
    imshow(V)
    colorbar()
    title('Konvergenzverhalten Newton')
    xlabel('x'),ylabel('i*y')
    return V
```

Newtonverfahren - Konvergenz



Aufbau

1. Numerische Mathematik
   - Poisson Problem
   - Differentialgleichungen
   - Lorenz-Gleichungen
   - Nichtlineare Gleichungen lösen und Optimierung

2. **Profiler**
   - Matlab
   - Python

Perfomance

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

Einflussfaktoren:

- Algorithmus
- Implementation
- Betriebssytem und Programmiersprache
- Hardware

Profiler

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

Aufbau

1. Numerische Mathematik
   - Poisson Problem
   - Differentialgleichungen
   - Lorenz-Gleichungen
   - Nichtlineare Gleichungen lösen und Optimierung

2. **Profiler**
   - Matlab
   - Python

Benutzung

```
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.

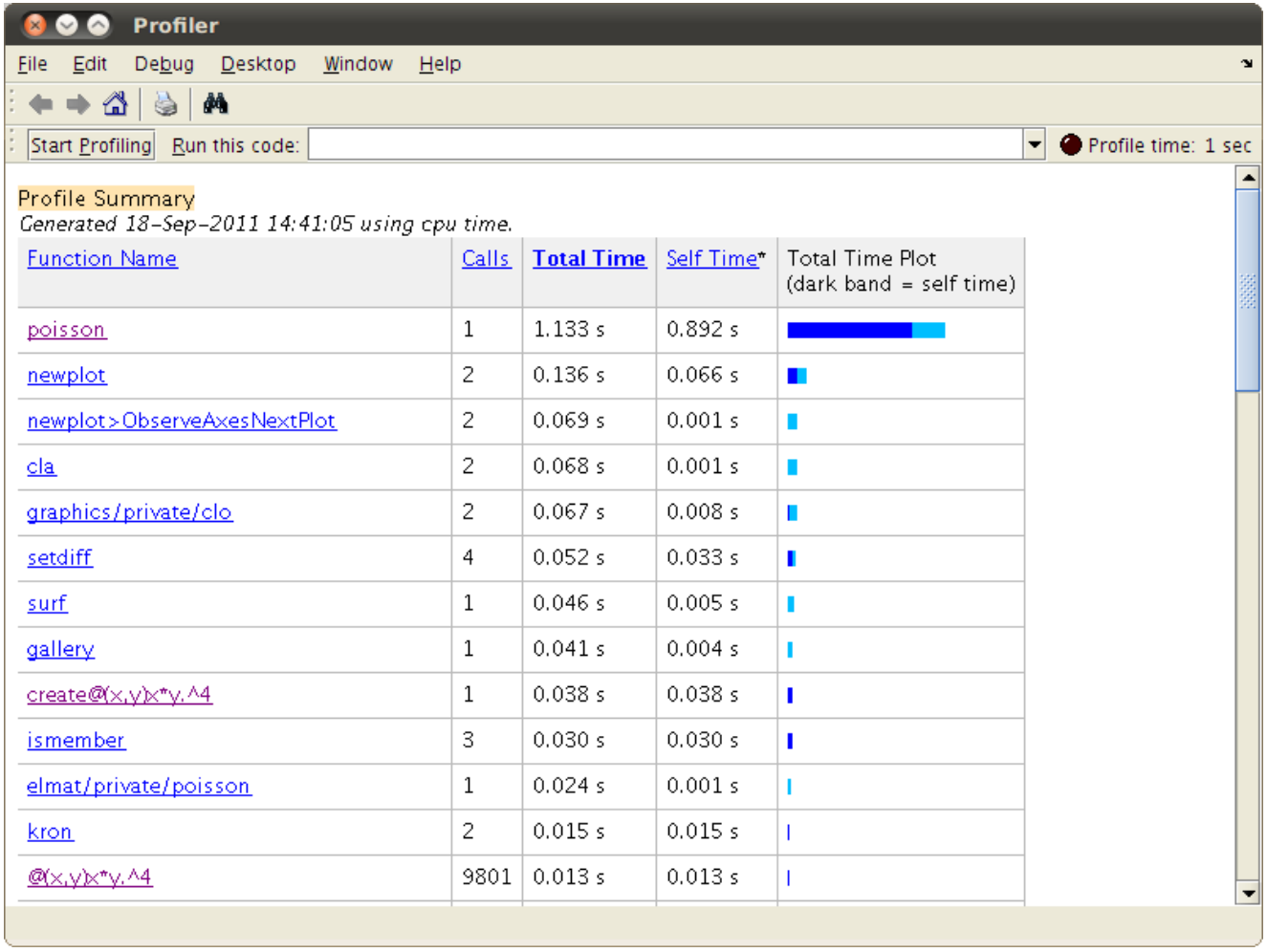
Profiler - Beispiel

```
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');
```

Profiling Poisson

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



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
```

Aufbau

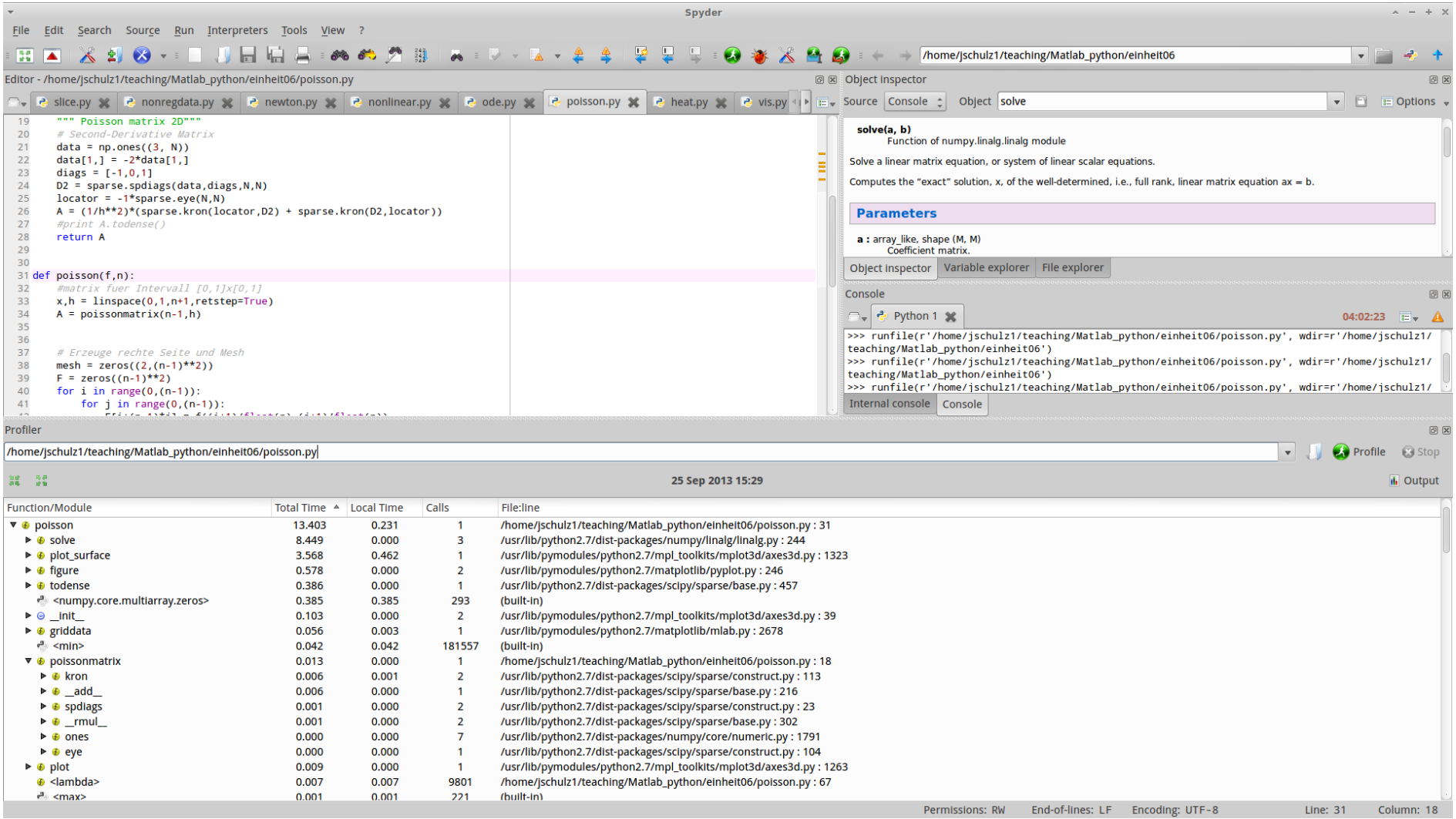
1. Numerische Mathematik
   - Poisson Problem
   - Differentialgleichungen
   - Lorenz-Gleichungen
   - Nichtlineare Gleichungen lösen und Optimierung

2. **Profiler**
   - Matlab
   - Python

Benutzung

- F10: Startet das gegebene Programm im grafischen Profiler.
- Der Profiler listet alle Funktionen in ihrer Hierarchie auf und zeigt deren Zeitverbrauch.
- Das ganze Programm wird geprüft.
- es wird nicht zeilenweise geprüft; nur Funktionsaufrüfe (siehe line – profiler)

Profiling Poisson



line profiler

Der line_profiler `http://pythonhosted.org/line_profiler/` kann exakt sagen, welche Zeilen wieviel Zeit verbraucht haben.
Dazu benutzt man einen sogenannten decorator an die Funktion, die man prüfen möchte

```
@profile
def function ():
```

und ruft dann (ausserhalb von spyder) den eigentlichen Profiler auf:

```
kernprof.py -l -v <file.py>
```