Sommercampus 2004
MATLAB-Kurs
2. Teil: Fortgeschrittene Techniken

Gerd Brunner, **Bernard Haasdonk**, Klaus Peschke

Albert-Ludwigs-Universität Freiburg

Institut für Informatik

Lehrstuhl für Mustererkennung und Bildverarbeitung

Prof. Dr.-Ing. H. Burkhardt

Übersicht

- Debugging
- Profiling
- Geschwindigkeits-Optimierung
- Interaktion mit Betriebssystem
- Interaktion mit Programmen
- Grafik-Objekte
- GUI-Programmierung
- Aufgaben

Debugging

- Fehlertypen:
  - Syntax-Fehler: leicht durch MATLAB Interpreter detektierbar dank Fehler-Kommentar, Datei- und Zeilenangabe.
  - Laufzeit-Fehler: schwieriger zu detektieren, lokale Variablen bei Funktionsabbruch verloren.
- Programmausgaben nutzen:
  - Entfernen von Semikolon: `A = B*B;` (keine Ausgabe)
    `A = B*B` (Ausgabe von A)
  - Einzelne Kommentare in MATLAB-Konsole ausgeben

```
disp(M); disp('reached A in mycode.m');
disp(['value s = ',num2str(s)]);
```

  - Alle Ein-Ausgaben der MATLAB-Konsole in eine Datei

```
diary alloutput.txt, diary on, diary off
```

Debugging

- Fehlertypen:
  - Syntax-Fehler: leicht durch MATLAB Interpreter detektierbar dank Fehler-Kommentar, Datei- und Zeilenangabe.
  - Laufzeit-Fehler: schwieriger zu detektieren, lokale Variablen bei Funktionsabbruch verloren.
- Programmausgaben nutzen:
  - Entfernen von Semikolon: `A = B*B;` (keine Ausgabe)

    `A = B*B` (Ausgabe von A)
  - Einzelne Kommentare in MATLAB-Konsole ausgeben

```
disp(M); disp('reached A in mycode.m');
disp(['value s = ',num2str(s)]);
```

  - Alle Ein-Ausgaben der MATLAB-Konsole in eine Datei

```
diary alloutput.txt, diary on, diary off
```

Debugging

- Fehlersuche per Kommandozeile:
  - Programmstop/Debug-Modus starten durch
    Code-Zeile **`keyboard`**
    oder Setzen von temporären Haltepunkten(**`dbstop`**)
  - Debug-Mode ist erkennbar durch das Prompt **`K>>`**
  - Lokale Variablen können in der Kommandozeile aufgelistet (**`whos`**) und eingesehen/modifiziert werden
  - Fortsetzen des Programms durch Kommando **`return`**

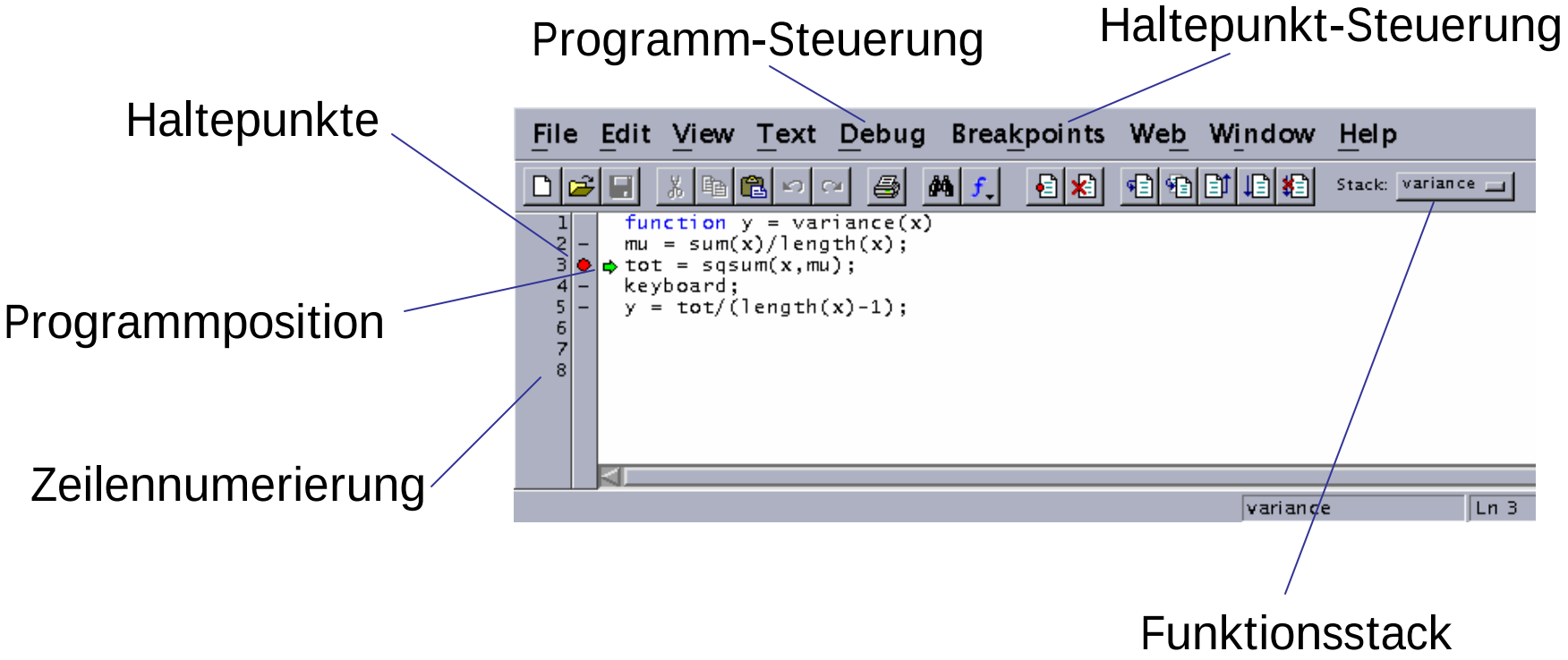
```
>>dbstop myfunction 3          (Haltepunkt gesetzt)
>>myfunction([1 2 3 4]);       (Funktionsaufruf)
3  tot = sqsum(x,mu);          (Auszuführende Zeile)
K>>x                           (Aufruf Ausgabe von x)
x =                            (Ausgabe von x)
     1  2  3  4
```

Debugging

- Übersicht über Debug-Kommandos:
  - Haltepunkt setzen: **`dbstop`**
  - Haltepunkt löschen: **`dbclear`**
  - Allgemeiner Halt: **`dbstop if warning`**
  - Funktionsaufruf-Stack **`dbstack`**
  - Liste der Haltepunkte **`dbstatus`**
  - Programm fortsetzen **`dbcont`**
  - Schritt(e) ausführen **`dbstep`**
  - Funktion mit Zeilennummern **`dbtype`**
  - Funktionshierarchie aufwärts **`dbup`**
  - Funktionshierarchie abwärts **`dbdown`**
  - Kompletter Programmabbruch **`dbquit`**

Debugging

- Grafische Debugger-Oberfläche:
  - automatisches öffnen des Programm-Codes
  - Variablenübersicht im Hauptfenster
  - Debug-Mode im Hauptfenster



Profiling

- herkömmlich: „heuristisches“ Optimieren
  - willkürliches Code-Optimieren
  - „Design-basierte“ Entscheidung für zu optimierende Funktionen
  - oft suboptimal, da Code oft nur „vermeintlich relevant“: exakte Aufruf-Häufigkeit oder Laufzeit oft verschätzt.
- besser: Profiling
  - Bestimmen der exakten Häufigkeit von Funktionsaufrufen
  - Durchführen von exakten Laufzeitmessungen pro Codezeile
  - Ermitteln von „tatsächlich relevanten“ Code-Stellen

  $\Rightarrow$ Hinweise auf Anpassung des Designs, Umprogrammierung

Profiling

- MATLAB-Profiler:
  - pro Zeile: Laufzeit, Häufigkeit
  - pro Funktion: Aufruf-Häufigkeit, parent/child-Funktionen
  - Protokollieren der Reihenfolge der Funktionsaufrufe

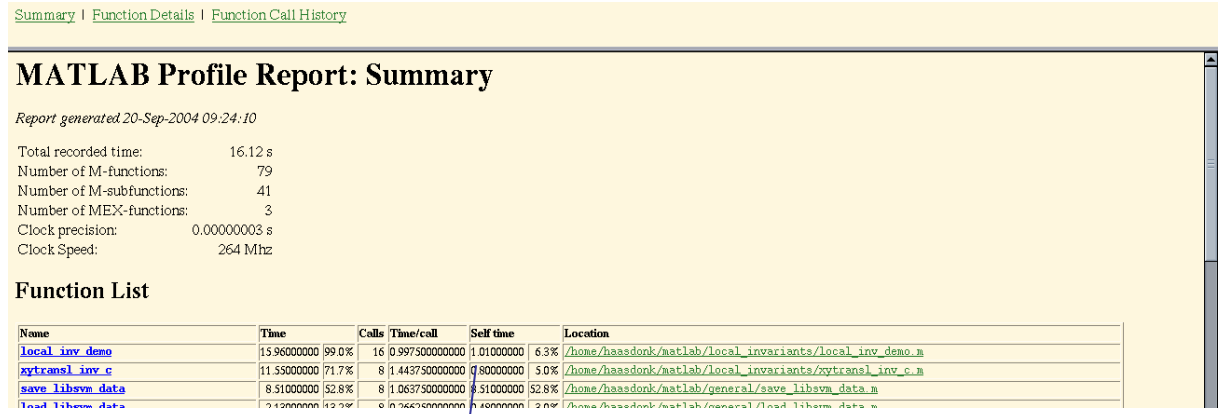
```
>>profile on -history     (Beginn des Profiling)
>>myfunction1, ...        (Funktionsaufrufe)
```

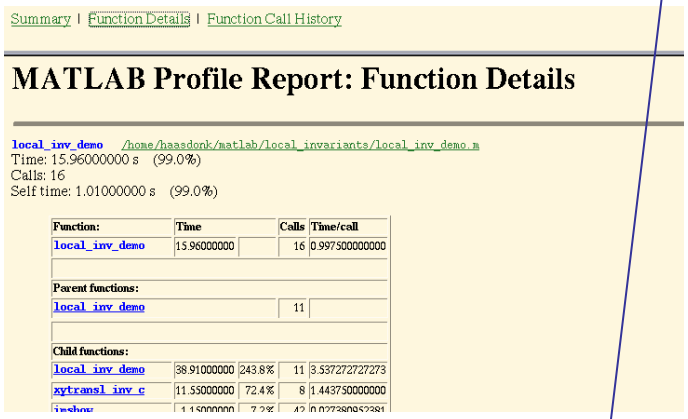
- Profile-Ergebnisse als Variable:

```
>>p = profile('info')
p =      FunctionTable: [123x1 struct]
       FunctionHistory: [2*10000 double]
        Clockprecision: 3.2584e-0.8
                  Name: 'MATLAB'
            ClockSpeed:  264
```

Profiling

- Profile-Ergebnisse als HTML: **`profile report`**

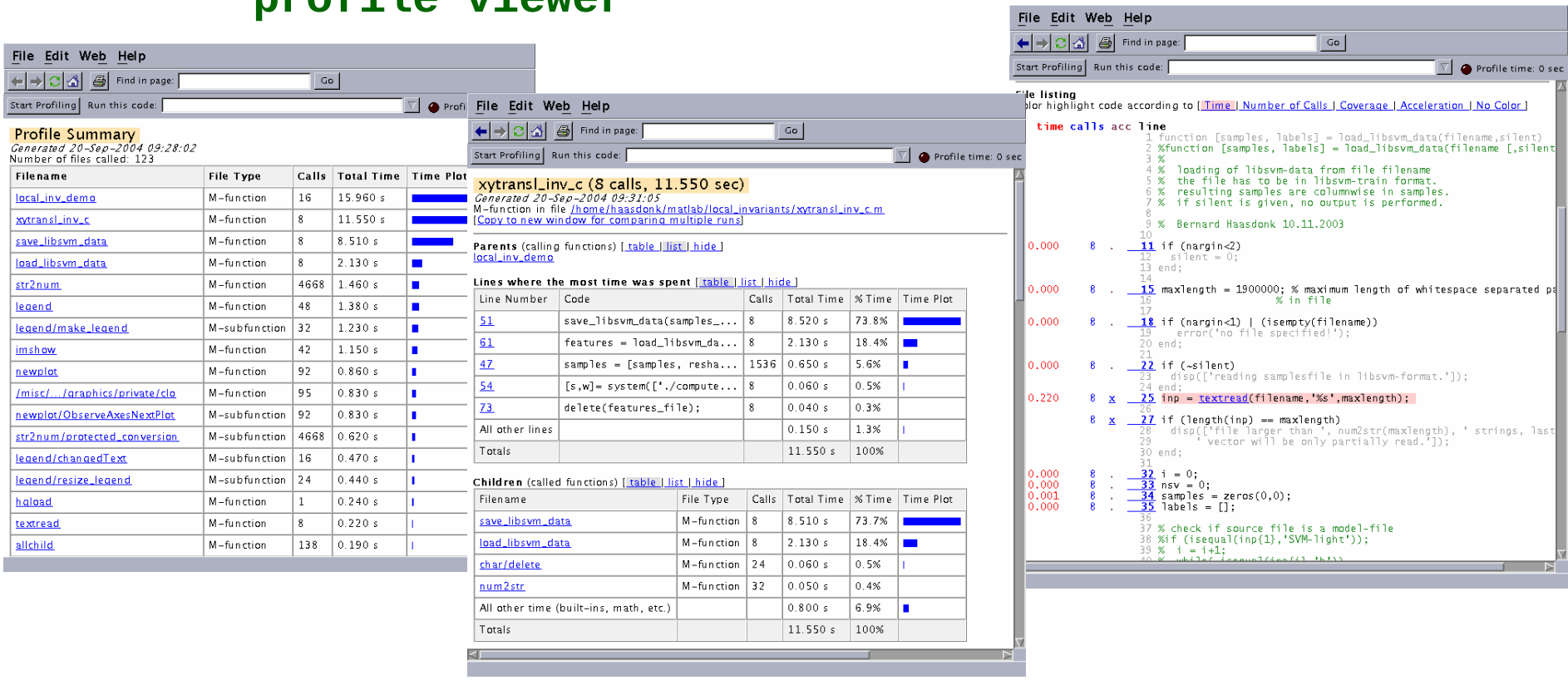




- nützlich: ‘self-time‘ und ‘History‘

Profiling

- Profile-Ergebnisse interaktiv:
  `profile viewer`



- nützlich: komplette Code-listings, Selektion von Zeilen
- auch sinnvolles Tool für Debugging, Code-Einarbeitung

Geschwindigkeits-Optimierung

- Preallocation
  - MATLAB erlaubt dynamisches Erweitern von Feldern. Speicherverwaltung erfordert jedoch Laufzeit.
  - Falls Endgrösse eines Feldes bekannt ist, macht frühe Speicherallokation Sinn.
  - Vektoren/Matrizen:

```
X = []; X(10,20)=0;  oder  X = zeros(10,20);  (0-Matrix)
Y = 5 * ones(20,30);                          (5-Matrix)
```

  - Cell-arrays:

```
C = {}; C(10)={ [] };  oder  C = cell(1,10);
```

  - Strukturen:

```
S = []; S.feld1 = []; S.feld2  = []; S(10).feld2 = []
oder  S = struct('feld1', C ,'feld2' , C)
```

Geschwindigkeits-Optimierung

- Vektorisierung
  - Problem von Interpreter-Sprachen: Schleifen
  - Stärke von MATLAB: Operationen auf Matrizen

  ⇒ Umformulieren von Algorithmen, indem Schleifen durch Matrix-Operationen implementiert werden.
  - elementweise Matrizen-Operationen: **`+ - min exp`**

    und „Punktieren“ von Matrix-Operatoren: **`.* .^`**

```
for x = 1:nx, for y = 1:ny,
   S(x,y) = (A(y,x)+A(x,y))^2
end;end;
```

⇓

```
S = (A+A').^2
```

Korrelationsmatrix

$$\mathbf{R} := \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i \cdot \mathbf{x}_i^T$$

⇓

```
R = (X * X')/size(X,2);
```

  - weitere Funktionen: **`find, sum, cumsum, repmat`**

Geschwindigkeits-Optimierung

- Optimierung bei Verwendung von komplexen Zahlen

```
z1 = a1 + i* b1;  z2 = a2 + i * b2;
z3 = z1 * z2;
a3 = real(z3)
b3 = imag(z3)
```

  - `i` kann nicht optimiert werden

⇒ Definition von `i=sqrt(-1);`

  - komplexe Operationen sind etwas langsamer als reelle

⇒
```
a3 = a1 * a2 – b1 * b2
b3 = a1 * b2 + a2 * b1
```

Interaktion mit Betriebssystem

- Shell-Kommandos ausführen:
  - in MATLAB-Kommandozeile: „!“

```
>>!emacs new_function.m &
```

  - im Code (keine interaktiven Befehle)

```
[stat, res] = unix('latex myfile.tex');
```

- Daten-Import/Export:
  - MAT-Files (C/C++, fortran Bibliotheken verfügbar)

```
save('allvar');     save('singlevar','dummy','L');
load('allvar');     temp = load('singlevar');
```

  - Binärdaten (**fopen, fread, fwrite, fclose**)
  - Formatierter text (**fscanf, fprintf, save -ASCII**)

```
[name,age,hght] = textread('data.txt','%s%d%f');
```

  - Bilddateitypen (**imread, imwrite**), ...

Interaktion mit Programmen

- MATLAB Array
  - dies ist einziges Datenformat für alle möglichen Objekttypen
  - Schnittstellen für dieses Datenformat ermöglicht Kombination mit anderen Programmiersprachen
  - Schnittstelle für C/C++: Datentyp **`mxArray`**
    Zugriff auf Felder via mx-Routinen

    `int mxGetM(mxArray *)` (erste Dimension)

    `double * mxGetPr(mxArray *)` (realteil der Daten)
- C/C++ Code: MATLAB ruft C (mex-Interfaces)
  - C-File (mit bestimmter Struktur) **`mymexfunc.c`**
  - Kompilieren mittels **`mex mymexfunc.c`**
    dies ergibt **`mymexfunc.mexglx`**
  - MATLAB Hilfetext in **`mymexfunc.m`**
  - Ausführen in MATLAB via **`mymexfunc`**

Interaktion mit Programmen

- Aufbau einer mex-Datei

```
#include “mex.h“
void myfunc(double *y, *x)
{
  y[0] = x[1]; y[1] = x[0];
}
void mexFunction(
  int nlhs, mxArray *plhs[],
  int nrhs, const mxArray *prhs[])
{
  double *y,*x;
  plhs[0]=
    mxCreateDoubleMatrix(2,1,mxREAL);
  y = mxGetPr(plhs[0]);
  x = mxGetPr(prhs[0]);
  y = myfunc(y,x);
}
```

Berechnung-Abschnitt

Interface-Abschnitt

Interaktion mit Programmen

- C/C++ Code: C ruft MATLAB („engine“ programs)
  - MATLAB-Engine läuft als Hintergrund Prozess, möglicherweise auf anderem Rechner.
  - Eigene C-Programme kommunizieren mit MATLAB-Engine, stellen Berechnungsanfragen.
  - Statt grosser MATLAB-Bibliothek nur kleine „Kommunikationsbibliothek“ erforderlich.

```
#include “engine.h“                    (Engine-Bibliothek einbinden)
void main() {
  Engine *ep; mxArray *a,*d;
  ep = engOpen(“\0“);                 (Engine öffnen)
  engPutArray(ap,a);                  (Variable setzen)
  engEvalString(ep,“d=A*A“);          (Berechnung durch Engine)
  d = engGetArray(ep,“d“);            (Variable holen)
  engClose(ep);}                      (Engine schließen)
```

Interaktion mit Programmen

- Weitere Möglichkeiten:
  - Platform-unabhängig:

    Kompilieren von MATLAB-Funktionen in standalone Anwendungen oder C/C++ Code via **`mcc`**
  - Fortran mex-Dateien
  - Windows: ActiveX

    MATLAB als ActiveX objekt, steuerbar von jedem „Automation Controller“, z.B. MS-Office,

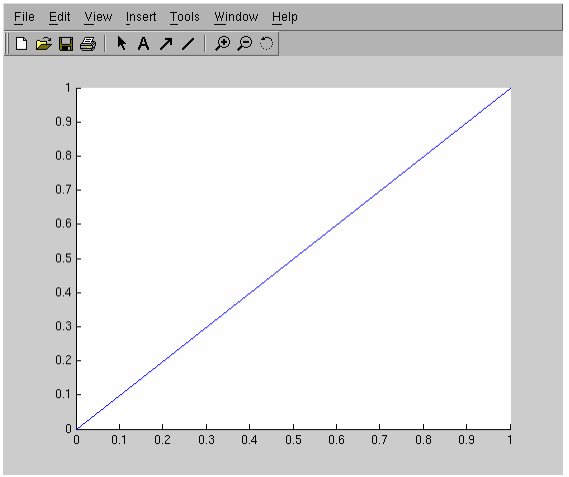
    Visual Basic/C++ Programme
  - Windows: Dynamic Data Exchange (DDE)

    ermöglichen von DDE-conversations zwischen

    Windows Anwendungen. Unterschiedliche Unterstützung für MATLAB als client oder server.

Grafik-Objekte

- Handle Graphics
  - alle Elemente in einem Grafikfenster sind durch Handles (eindeutige Zahl) repräsentiert.
  - ein Handle **h** hat eine Liste von Property/Value-Paaren, einsehbar via **get(h)**, z.B. Type, Position
  - Setzen der Werte (falls) möglich via **set**

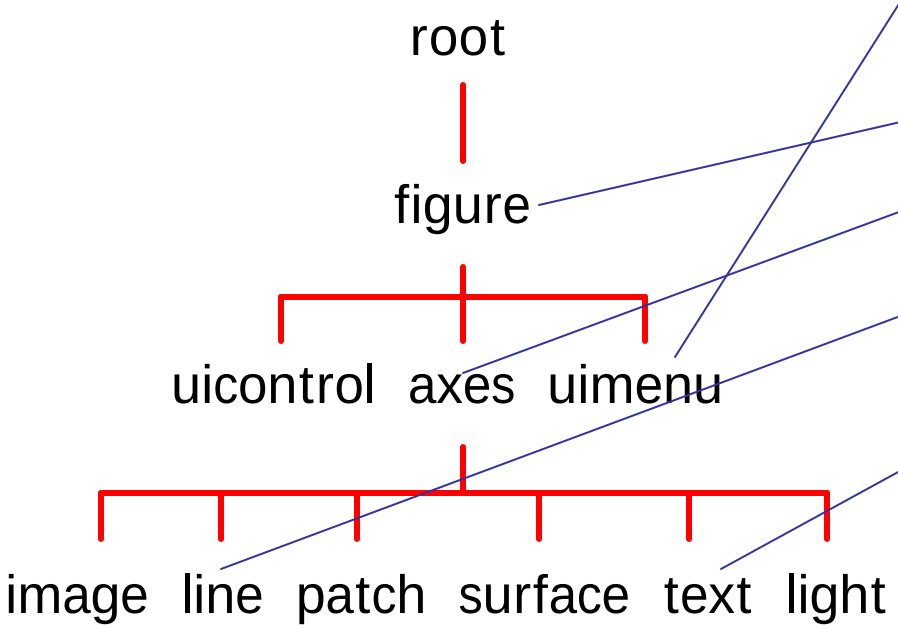


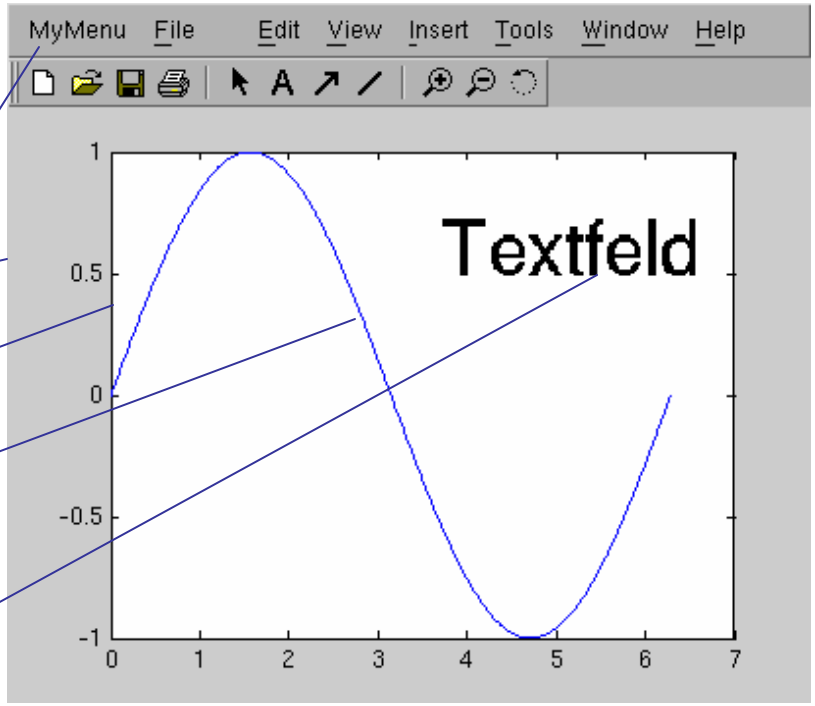

```
>>l = line([0 1],[0 1])
l = 106.0002
>>set(l,'Color',[1 0 0]);
```

```
>>get(l)
    Color = [0 0 1]
    LineStyle = -
    LineWidth = [0.5]
    Xdata = [0 1]
    Ydata = [0 1]
    Parent = [101]
```

Grafik-Objekte

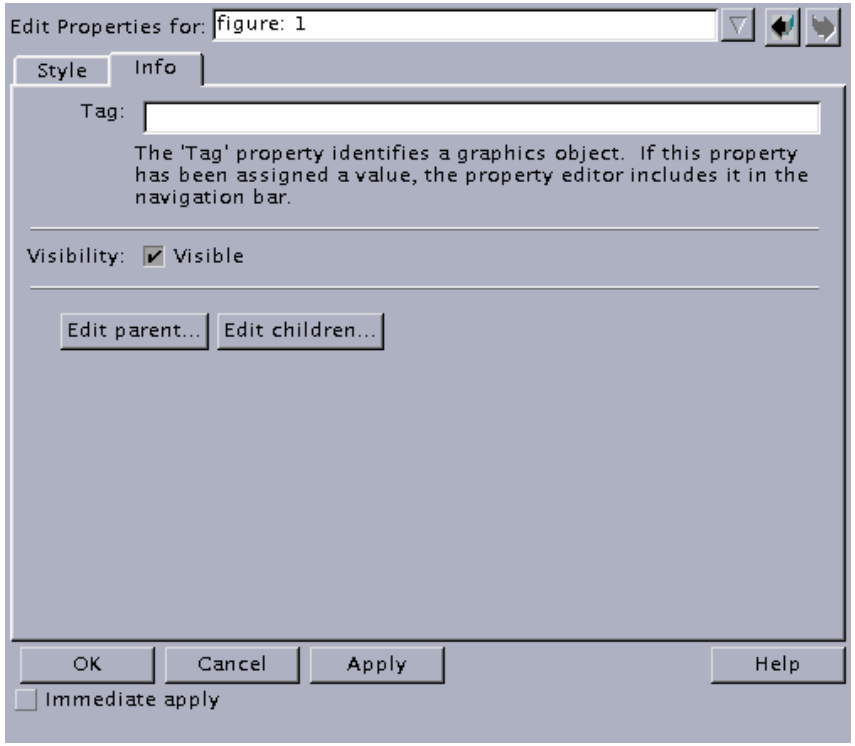
- Grafik Hierarchie
  - zugänglich durch Properties
    'Children','Parent'
  - Baumstruktur





Grafik-Objekte

- Property Editor
  - grafische Oberfläche zum Einsehen/Manipulieren der Objekt-Hierarchie einer Figure/Axes
  - Beispiel: aktuelle Figure editieren via **`propedit(gcf)`**



GUI-Programmierung

- uicontrols
  - interaktive Objekte in einer Figure

    `pushbutton`

    `togglebutton`

    `radiobutton`

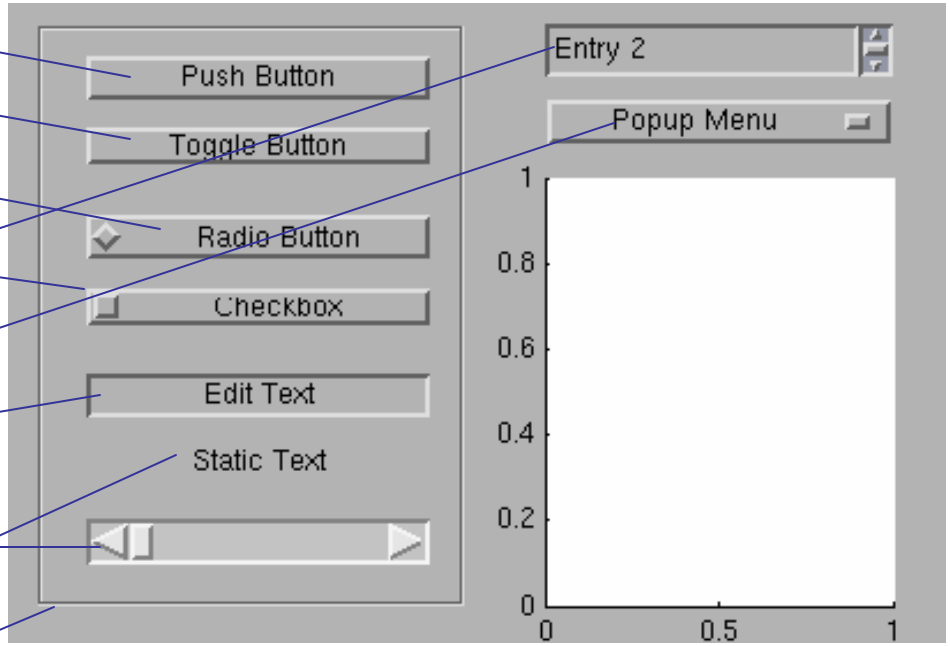
    `checkbox`

    `listbox`

    `popupmenu`

    `edit`

    `slider`



  - statische Objekte

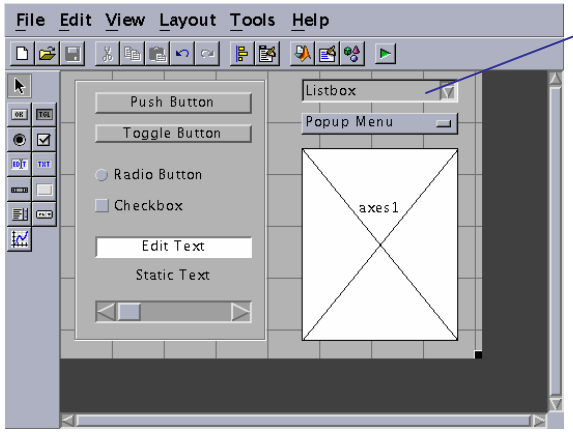
    `text`

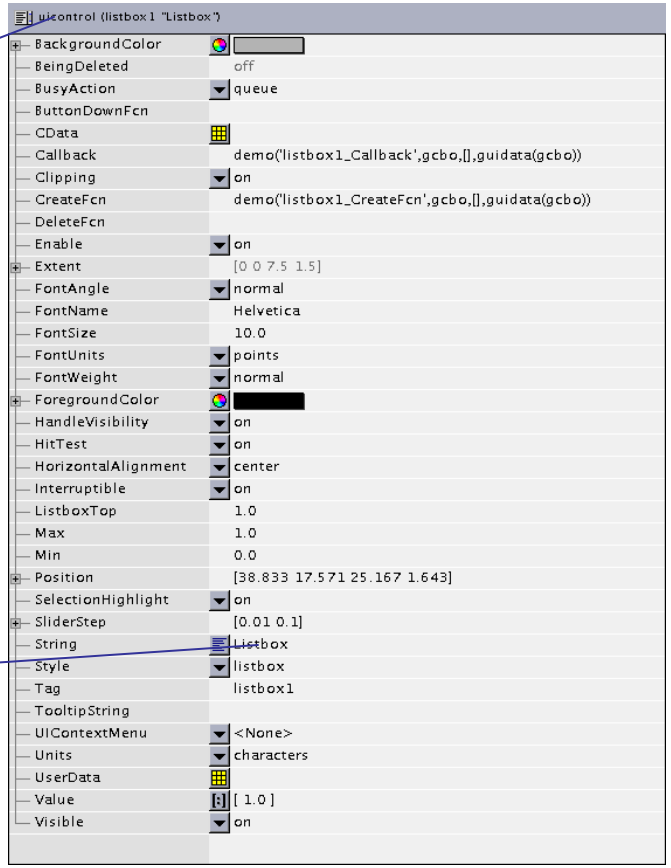
    `frame`

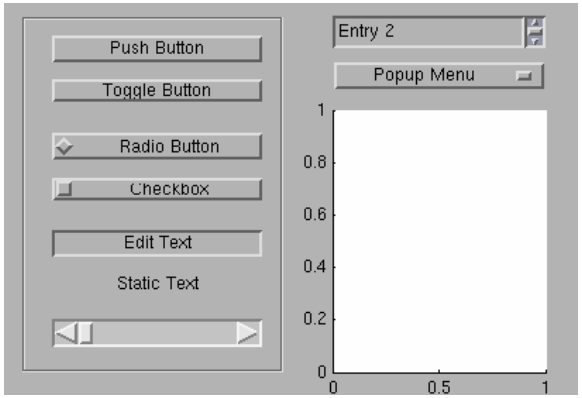
  - Erzeugung z.B via **`h=uicontrol(gcf,'Style','text')`**

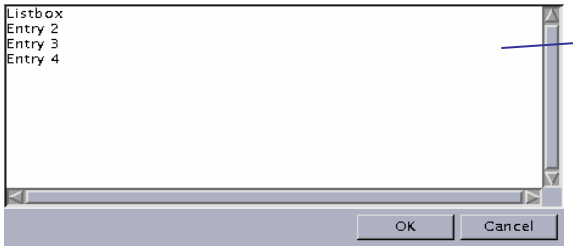
GUI-Programmierung

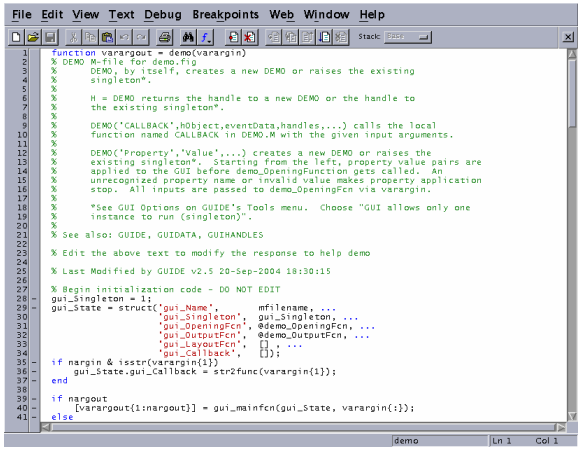
- Layout der GUI mittels interaktivem Editor **`guide`** ⇒ **`myGUI.fig`** + **`.m`**











GUI-Programmierung

- Implementation der Callback-Funktionen in **myGUI.m**
  - Callback-Funktion eines Objekts wird bei User-Interaktion aufgerufen
  - Referenz auf aktuelle Figure via **gcbf**

```
function pushbutton1_Callback(…)
close(gcbf);      (Schliesse Figure der GUI nach Knopfdruck)
```

  - Referenz auf aktuelles Objekt via **gcbo**

```
function slider1_Callback(…)
phi = 2*pi*get(gcbo,'Value');  (Winkel aus Slider-Wert)
```

  - Suche das Handle eines anderen Objektes via **findobj**

```
st = findobj(gcbf,'Tag','text1')  (statisches Textfeld)
set(st,'String',num2str(phi));    (setze Winkel als Text)
```