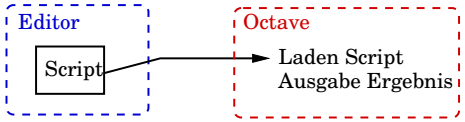
Octave - Intro (1)

Allgemeine Benutzung

- Aufruf über Kommando *octave* am Shell-Prompt
- Interaktive Befehlseingabe im Octave-Fenster
  oder Ausführen zuvor erstellter Scripte
- Scripte werden mit Editor erstellt
  - Editor *Emacs* hat Octave-Unterstützung (z.B. Syntax-Highlighting)
    Aufruf des Emacs über Kommando *emacs datei &* am Shell-Prompt
  - Dateinamenserweiterung ".m", z.B. *aufgabe1.m*
  - Aufruf in Octave über Scriptnamen *ohne* ".m"-Erweiterung
    mit TAB automatische Ergänzung am Octave-Prompt möglich



Octave - Intro (2)

Allgemeine Syntax

- Octave ist casesensitiv
- Kommentare von '#' bis Zeilenende
  Konvention für korrekte Einrückung im Emacs:
  - '##' reine Kommentarzeile
  - '#' Kommentar am Zeilenende (wenn vorne Code steht)
- Kommandoabschluss Zeilenumbruch oder ';'
  ohne ';' wird Ergebnis des Kommandos ausgegeben
  Mehrzeilige Befehle mit '...' oder '\' am Zeilenende
- Funktionsaufrufe: *func (arg1, [arg2, ...])*
- eingebaute Variablen (*pi, e, i*) nicht unterscheidbar von
  Benutzervariablen (auch lokal überschreibbar!)

Octave - Intro (3)

Variablen und Werte

- Interpretersprache => keine Deklaration nötig (anders als C)
  erste Zuweisung erzeugt Variable:

```
a = 10; # Anlage Variable und Zuweisung
```

- zwei skalare Datentypen:
  - *Zahl*: keine Unterscheidung Integer/Real; immer als *double* gespeichert
  - *String*: in single oder double Quotes; Escape durch Backslash (`'It\'s true'`)
- Matrixkonstruktor '[ ]' erzeugt Matrix/Vektor aus Skalaren

```
A = [ 1, 0; 0, 1 ] # 2x2 Einsmatrix
```

  - Zeilenwerte durch Komma oder Spaces getrennt
  - Spaltenwerte durch Semikolon getrennt

Octave - Intro (4)

Operatoren

- übliche arithmetische Operatoren +, -, *, / und
  ^ (oder **) für Potenzieren sind *überladen*, d.h. verhalten
  sich je nach beteiligten Datentypen verschieden:

```
A = [1,2] * 5;     # skalare Multiplikation
B = [1,2] * [1;2]; # Matrixprodukt
```

- Zuweisungsoperator: =
  Vergleichsoperatoren: ==, >, >=, <, <=, != (ungleich)
- spezielle Matrixoperatoren:
  - Transposition: `'` bzw `.'` (mit bzw. ohne komplexe Konjugation)
  - Elementweise Verknüpfung durch Punkt vor Operatoren, z.B. `[1,2] .* [2,3]`
  - Indexoperator: Klammern, Indizes starten bei 1

```
A(1,5)   # Zeile 1, Spalte 5
```

  - Konstruktor gleichmäßiger Vektoren: Doppelpunkt (oder *linspace*)

```
1:3      # erzeugt [1 2 3]
1:0.5:3  # erzeugt [1 1.5 2 2.5 3]
```

Octave - Intro (5)

dynamische Speicherverwaltung

- Zuweisung an zu hohen Index => automatische Vergrößerung

```
## Anlegen einer 2x2 Matrix
A = [ 1, 0; 0, 1 ];
## Implizite Umwandlung in 2x5 Matrix
## restliche Werte auf 0 initialisiert
A(1,5) = 1;
```

- Auswahl von Teilmatrizen durch Vektoren als Indizes

```
## Auswahl der zweiten Zeile
A(2, 1:5)  # beachte 1:5 == [1 2 3 4 5]
## Reduktion A auf 2x2 Matrix
A = A(1:2, 1:2)
```

reiner Doppelpunkt ist Abkürzung für ganze Zeile/Spalte

```
A(1, :)  # komplette 1. Zeile
A(:, 1)  # komplette 1. Spalte
```


Octave - Intro (6)

Eingebaute Funktionen/Befehle (1)

- allgemeine Funktionen
  - `help command` - Online Hilfe zu *command* (beenden mit 'q')
  - `exit,quit` - beendet Octave
  - `who` - zeigt Variablen an; für Optionen (z.B. *-long*): *help who*
  - `clear var1 var2 ...` - löscht Variablen (ggf. alle)
- Matrix-Funktionen
  - `eye(n)` - erzeugt n × n Einheits-Matrix
  - `zeros(n,m)` - erzeugt n × m Matrix mit Nullen
  - `ones(n,m)` - erzeugt n × m Matrix mit Einsen
  - `rand(n,m)` - erzeugt n × m Matrix mit Zufallszahlen in (0,1)
  - `size(A)` - Anzahl Zeilen und Spalten der Matrix *A*, z.B. *[rows, cols] = size(A)* oder *cols = size(A)(2)*
  - `length(A)` - Maximum Zeilen-/Spaltenzahl

Octave - Intro (7)

Eingebaute Funktionen/Befehle (2)

- String-Funktionen
  - `sprintf` - gibt formatierten String zurück, z.B. *s = sprintf("Anzahl: %d\n", n)*
  - `strcat` - String Concatenation
  - `substr` - extrahiert Teilstring aus
  - `str2num` - Umwandlung String in Zahl
- Ein- und Ausgabe

  Variable ohne ';' gibt Wert aus. Berechnung ohne ';' gibt Ergebnis aus und weist es interner Variablen *ans* zu
  - `disp` - Ausgabe mit anschließendem Newline
  - `printf` - Formatierte Ausgabe (wie gleichnamige C-Funktion)
  - `input` - Eingabeaufforderung

  Ferner viele Funktionen der C stdio-Bibliothek verfügbar

Octave - Intro (8)

Grafik (1)

- Grafikdarstellung über externes Programm *gnuplot*
  - gnuplot muss ebenfalls installiert sein
    - unter Linux bei jeder Distribution dabei
    - beim Windows-Installer dabei
    - bei MacOS X Paket separates Application-Bundle
    - Umgebungsvariable GNUTERM auf "aqua" setzen (sonst wird X11 benötigt)
- Funktionen zum Zeichnen von Kurven

`plot(x,y)`
- zeichnet die Punkte *x(i), y(i)* als Kurve; (*x* und *y* sind gleich große Vektoren)
- *x* kann weggelassen werden; dann werden *x*-Werte 1 bis *length(y)* genommen

`hold on, hold off`
- Plots übereinander (*on*) oder jeweils neuer Plot (*off*)

`clearplot`
- Löscht aktuellen Plot im Grafikfenster

Octave - Intro (9)

Grafik (2)

- Beispiele
  - Zeichne *sin(x)* im Intervall [0,9]:

```
x = 0:0.1:9;
plot(x,sin(x));
```

  - Zeichne im selben Plot *cos(x)* dazu:

```
hold on;
plot(x,cos(x));
```

- Bemerkung:
  - das funktioniert, weil *sin(x)* für einen Vektor *x* als Input den Wert für jede Komponente berechnet:

$$\sin([x_1, x_2, \ldots, x_n]) := [\sin(x_1), \sin(x_2), \ldots, \sin(x_n)]$$

  - solche eine Funktion heißt in Octave *vektorisiert*

Octave - Intro (10)

Grafik (3)

- Anpassen des Grafikformats
  - drittes, optionales Argument in *plot*:
    - `plot(x,y,"-;text;")` Linienstil und Legendentext
    - Beispielwerte für Linienstil: '-' = Kurve, '^' = Stabdiagramm, '*' = Punkte
    - Text kann auch weggelassen werden: `plot(x,y,"*")`
    - vollständige Dokumentation siehe *help plot*

Beispiel:

```
x = [0:0.1:9];
plot(x,sin(x),"-;sin;");
hold on;
plot(x,cos(x),"^;cos;");
```


Octave - Intro (11)

Grafik (4)

- Einstellung der Achsendimensionen

`axis(`x_{min}`,`x_{max}`,`y_{min}`,`y_{max}`)`

- Spezielle Plots

```
bar(x,y)      - erzeugt Balkendiagramm
stairs(x,y)   - erzeugt Treppenfunktion
                (keine lineare Interpolation der Werte)
```

- Export und Druck der Grafik

```
print(filename,options)
```

  - filename extension bestimmt Dateityp, z.B. *print("plot.eps")* oder *print("plot.fig")*
  - weitere Optionen möglich, z.B. "-mono" für Schwarz-Weiß
  - oder "-FTimes-Roman:20" für Font und Größe

Octave - Intro (12)

Benutzerdefinierte Funktionen

- Beispiel

```
## Funktion, die zwei Zahlen addiert
function y = addfunc(x1,x2)
  y = x1 + x2;
endfunction
```

- Bemerkungen
  - Argumentwerte *x1, x2* werden als Kopie übergeben (*call by value*) Im Kopf deklarierte Variablen *y, x1, x2* leben nur in Funktion
  - Returnwert durch Zuweisung an temporäre Variable *y*
  - in Funktionsrumpf definierte Variablen sind lokal Zugriff auf globale Variablen mit Qualifier *global*
  - auch mehrere Rückgabewerte möglich: *function [y1,y2] = func(...)* oder auch gar kein Rückgabewert: *function func(...)*

Octave - Intro (13)

Funktionen und Scriptfiles

- Funktionsfiles
  - normalerweise Funktionen vor Benutzung definieren
    Alternative: Definition in *Funktionsfile*
    - Dateiname = Funktionsname + .m
    - Datei speichern im Octave-Suchpfad (*DEFAULT_LOADPATH, LOADPATH*)
  - Funktionsfile wird dann automatisch geladen, wenn die gleichnamige Funktion zum erstenmal verwendet wird
- Scriptfiles
  - enthalten beliebige Anweisungen (auch Funktionsdefinitionen)
  - Dateiname ebenfalls mit *.m* Erweiterung,
    Aufruf über Dateinamen ohne *.m* Erweiterung
  - damit Octave Scriptfiles nicht mit Funktionsfiles verwechselt, empfiehlt das Octave Handbuch folgenden Trick:

```
## beginne Script mit Dummykommando ungleich "function"
1;  # diese Anweisung tut nichts
```


Octave - Intro (14)

Kontrollfluss

- Verzweigung mit *if*

```
if (a>1)
  printf("a > 1");
else
  printf("a <= 1");
end   # oder 'endif'
```

  - Kombination von Bedingung mit '&' und '|'; Negation mit '!' oder '~'
  - Weitere Alternativzweige mit *elseif* möglich

- Schleifen mit *for* und *while*

```
for i=1:3                  while i<=3
  ## do some stuff           i = i+1;
end   # oder 'endfor'      end   # oder 'endwhile'
```

  - auch Schrittweite ungleich 1 in *for*-Schleife möglich, z.B. *for 1:0.5:3*
  - Schleifenabbruch (innerste Schleife) mit *break* möglich

Octave - Vektorisierung (1)

Vektorisierung

- for-Schleifen laufen im Interpreter ab => langsam
- Workaround: verlagere Schleifen wenn möglich in die in C implementierte Funktion (*Vektorisierung*)

```
## Loop Version
function y = add_loop(x)
  y = x;
  for k = 1:length(x)
    y(k) = y(k) + 1;
  endfor
endfunction
```

```
## vektorisierte Version
function y = add_vect(x)
  y = x + ones( size(x) );
endfunction
```

```
## messe Laufzeit
x = 1:50000;
t = cputime();
add_loop(x);
t = cputime() - t;
printf("add_loop: %f\n", t);
```

```
## messe Laufzeit
x = 1:50000;
t = cputime();
add_vect(x);
t = cputime() - t;
printf("add_vect: %f\n", t);
```

```
add_loop: 2.640000
```

```
add_vect: 0.010000
```


Octave - Vektorisierung (2)

Vektoroperationen

Bezeichnung	Vektoroperation	Schleifenversion
		`for i=1:length(x)`
Vektoraddition	`z = x + y`	`  z(i) = x(i) + y(i)`
skalare Multiplikation	`z = x * 5`	`  z(i) = x(i) * 5`
punktweise Multiplik.	`z = x .* y`	`  z(i) = x(i) * y(i)`
punktweise Division	`z = x ./ y`	`  z(i) = x(i) / y(i)`
punktweise Potenz a)	`z = x .^ 2`	`  z(i) = x(i) ^ 2`
punktweise Potenz b)	`z = 2 .^ x`	`  z(i) = 2 ^ x(i)`
		`endfor`

Beispiel

Erzeugung des Vektors $x = (1, p, p^2, ..., p^n)$ mittels

```
x = p .^ (0:n);
```

Octave - Vektorisierung (3)

Konstruktion von Vektoren

- normale Lösung mittels *for*-Schleife:

```
## Erzeugung von [5, 5, ..., 5]
x = [];     # leerer Vektor
for i = 1:n
  x(i) = 5;
endfor
```

Probleme:

- Schleifendurchlauf im Interpreter
- dynamische Speichererweiterung bei jedem Schleifendurchlauf (vermeidbar durch Preallocierung mit *x = zeros(1,n)*)

In Octave bessere Lösung: `x = 5 * ones(1,n);`

Octave - Vektorisierung (4)

Vektor-/Matrix-Konstruktoren

Konstruktor	**Ergebnis**
`a:b`	$(a, a+1, a+2, \ldots, b)$
`a:eps:b`	$(a, a+\varepsilon, a+2\varepsilon, \ldots, b)$
`linspace(a,b,n)`	$(a, a+\delta, a+2\delta, \ldots, b)$ mit $\delta = (b-a)/(n-1)$
`ones(n,m)`	$n \times m$ Matrix aus Einsen
`zeros(n,m)`	$n \times m$ Matrix aus Nullen
`eye(n,m)`	$n \times n$ Einheitsmatrix
`diag(x,k)`	Diagonalmatrix mit Vektor x auf der k-ten Nebendiagonale

Beispiel

Varianten zur Erzeugung des Vektors *x = (a, 2a, 3a, ..., na)*:

```
x = (a * ones(1:n)) .* (1:n);
x = a * (1:n);
x = a:a:n*a;
```


Octave - Vektorisierung (5)

Vektorisierte Funktionen (1)

- viele eingebaute Funktionen sind "vektorisiert"
  - eigentlich skalare Funktionen (wie z.B. *sin*) akzeptieren auch Vektor bzw. Matrix als Input
  - Ergebnis ist Matrix der einzelnen Funktionswerte

$$\sin\begin{pmatrix} a & b \\ c & d \end{pmatrix} := \begin{pmatrix} \sin(a) & \sin(b) \\ \sin(c) & \sin(d) \end{pmatrix}$$

  - Wenn die Funktion nicht als Octave-Script, sondern als C-Funktion implementiert ist, deutlicher Performancegewinn (Schleifen laufen dann nicht im Interpreter ab)

Octave - Vektorisierung (6)

Vektorisierte Funktionen (2)

- Beispiel:

Erzeugung Plotdaten für Funktionsgraphen

```
## langsame Version
## mittels for-Schleife
x = 0:0.1:9;
y = zeros(size(x));
for i = 1:length(x)
  y(i) = sin(x(i));
endfor
plot(x,y);
```

```
## schnelle Version durch
## Nutzung vektorisierter Funktion
x = 0:0.1:9;
y = sin(x);
plot(x,y);
```

Bemerkungen:

- *size(x)* gibt Dimension einer Matrix zurück
- Schleifenversion alloziert benötigten Speicher zu Beginn mit *y = zeros(size(x))*
  - => kein Zeitverlust durch Nachallocierung in jedem Schleifendurchlauf