Einführung in GNU Octave

S. Bouattour, D. Paulus

21. Mai 2003

Inhaltsverzeichnis

1 Grundlagen

1.1 Was ist *Octave* ?

Der Name *Octave* hat mit Musik nichts zu tun! Es ist der Name eines ehemaligen Chemie Professors vom Autor von *Octave* [1]. Dieser Professor war für sein Talent berühmt, schnelle "back of the envelope" Berechnungen durchzuführen. Es ist eine interaktive Skriptsprache, die speziell für vektorbasierte Berechnungen optimiert ist und dabei Standardroutinen der numerischen Mathematik (z.B. EISPACK oder LAPACK) auf einfache Weise zugänglich macht [**?**]. Zusätzlich zu der Lösung von Problemen der Linearen Algebra und der Integralrechnung, dem Lösen von Gleichungssysteme und Polynomen etc., ist es möglich, eigene Funktionen zu definieren und/oder dynamisch geladene Module in C++, C oder Fortran zu verwenden. Dadurch ist Octave fast uneingeschränkt erweiterbar!

1.2 Installation

GNU *Octave* ist eine freie Software, deren Verwendung und Distribution den allgemeinen öffentlichen Lizenzregeln (General Public License GPL) unterliegt.

Linux: *Octave* ist ein Native-Linux-Programm, das bei vielen Distributionen schon als Binary-Package vorhanden ist. Ist dies nicht der Fall, so siehe http://www.octave.org/download.html.

Windows: Obwohl *Octave* ein Linux-Programm ist, so existiert auch eine Portierung nach Windows. Dieses erfordert jedoch dennoch Linux-Grundwissen, weshalb eine FAQ unter http://octave.sourceforge.net/Octave_Windows.htm eingerichtet wurde.

1.3 Aufruf von *Octave*

- Starten von *Octave* durch:
  - `octave` ⟶ Interaktiver Modus.
  - `octave file.m` ⟶ Ausführung der Skriptdatei (s. Abschnitt 1.4) und beenden von *Octave* .
- Verlassen von *Octave* durch `quit` oder `exit`.

1.4 Syntaxkonventionen

Die Syntax von *Octave* ist der von MATLAB sehr ähnlich. *Octave* -Programme können meist von MATLAB ausgeführt werden. Der Umgekehrte Weg ist aufgrund des größeren Funktionsumfang von MATLAB nicht immer gewährleistet.

- Befehle können entweder interaktiv oder über Skriptdateien eingegeben werden.
- Skriptdateien sind Textdateien mit dem Suffix **.m**. Sie werden im interaktiven Modus durch die Eingabe des Dateinamens ohne suffix interpretiert (gelesen und ausgeführt). Sie können aber auch durch den Aufruf von `octave file.m` von der Shell aus interpretiert werden. In beiden Fällen verhalten sie sich so, als ob ihr Inhalt Zeile für Zeile am Prompt eingegeben würde.
- *Octave* ist case-sensitive.
- Das Semikolon (**;**) trennt mehrere Befehle in einer Zeile und unterdrückt die Ausgabe der Auswertung.
- Das Komma (**,**) trennt mehrere Befehle in einer Zeile, unterdrückt deren Ausgabe der Auswertung aber nicht. Dies kann für Ablaufkontrolle und Debugging nützlich sein.

[1] John W. Eaton

- Drei Punkte (. . .) am Zeilenende bedeuten, dass der Ausdruck in der nächten Zeile fortgesetzt wird. Dies ist sinnvoll, um zu lange Zeilen in mehrere aufzuteilen.
- Kommentare beginnen mit dem Zeichen **%** oder **#** .

1.5 Hilfe!

- `help` listet alle Befehle und internen Variablen auf.
- `help name` gibt einen Hilfetext zur Variable oder Funktion "name" aus.
  Beispiel:

```
octave:6> help norm
```

2 Datentypen

Intern bearbeitet *Octave fast* alle Datentypen als Matrizen; auch ein Skalar ist eine 1×1 Matrix.

2.1 Numerische Objekte

- **Reale und Komplexe Skalare:** sind gespeichert in doppelter Genauigkeit. Der Wertebereich [2] liegt zwischen 2.2251×10^{-308} und 1.7977×10^{308}. Die relative Genauigkeit ist 2.2204×10^{-16}.
- **Matrizen:** können eine beliebige Größe haben und ihre Größe dynamisch ändern. Die Werte einer Matrix sind wiederum reale oder komplexe Skalare.
  Beispiel:

```
octave:32> [1 2; 3 4]
ans =

  1  2
  3  4
octave:33> [1+2i  2; 3 4-4i]
ans =

  1 + 2i  2 + 0i
  3 + 0i  4 - 4i
```

2.2 String-Objekte

String-Objekte sind Zeichenfolgen, die zwischen einfachen oder doppelten Anführungszeichen eingeschlossen sind (¨ oder ´. Intern speichert *Octave* Strings als Matrizen von Zeichen. Alle Indizierungsoperationen, die für Matrizen definiert sind, arbeiten auch auf Strings.
Beispiel:

```
octave:11> x = ["das ", "ist ", "ein ", "Beispielsatz."]
x = das ist ein Beispielsatz.

octave:12> y = ["das ist ein Beispielsatz."]
y = das ist ein Beispielsatz.

octave:12> x(8) == y(8)
ans = 1
```

[2] auf Systemen, die das IEEE floating-point Format verwenden

2.3 Datenstrukturen

Datenstrukturen dienen der Zusammenfassung von Objekten verschiedener Typen. Die Syntax ähnelt C-style Strukturen. Intern sind sie als assoziative Arrays implementiert.
Beispiel:

```
octave:17> c.real=2;
octave:18> c.img=3.45;
octave:19> c.name="complex";
octave:20> c   % Ausgabe der Struktur
c =
{
  name = complex
  real = 2
  img = 3.4500
}
```

3 Operationen

- **Wertzuweisung:** `variable = expression`
- **Skalaroperationen:** die üblichen Operatoren sind vorhanden: `- * / ^ \ +`
- **Logische Operatoren:** Die bekannten Operatoren mit der üblichen Semantik sind vorhanden: `< <= > >= == & |`.
  Ausserdem steht `<>` für ungleich und `~` für nicht.
- **Matrixoperationen:** Matrizen sind die wichtigsten Bausteine zur Programmierung in *Octave* . Deswegen ist eine besonders grosse Anzahl an Operationen zur Matrixmanipulation vorhanden. Eine übersichtliche Auflistung folgt deshalb im nächsten Abschnitt.
- **Funktionen**, die bei jeder vernünftigen Mathematik-Software vorhanden sein sollten, fehlen auch bei *Octave* nicht: `sin,cos,exp,log,atan,abs,` usw. Wenn diese Operationen mit Matrizen aufgerufen werden, so werden sie auf jedes Element der Matrix angewendet. Zusätzlich bietet *Octave* viele andere Funktionen an, die sich mit bestimmten Themengebiete der Mathematik befassen (lineare Algebra, nicht-lineare Gleichungen, Arithmetik, Differentialgleichungen, Optimierung, Statistik, Manipulation von Polynomen, Kontrolltheorie, Bildverarbeitung und Sprachverarbeitung usw.).

3.1 Matrixoperationen

- Zeilenvektor:

  ```
  v = [ 1 2 3 ] oder v = [1,2,3,4]
  ```

- Spaltenvektor:

  ```
  v = [ 1; 2; 3 ]
  ```

  Optional kann auch jedes Semikolon durch eine neue Zeile ersetzt werden:

  ```
  v = [ 1
  2
  3 ]
  ```

- Automatische Erzeugung von Zeilen-Vektoren mit konstantem Inkrement:

  Begin:Inkerement:Ende

```
octave:47> x = 1:0.5:3
x =

  1.0000  1.5000  2.0000  2.5000  3.0000
```

- Erzeugung einer Matrix:

```
octave:48> A = [1 2; 3 4]
A =

  1  2
  3  4
```

- Zusammensetzung von Matrizen:

```
octave:48> A = [1 2; 3 4]
A =

  1  2
  3  4
octave:49> b = [5;6]
b =

  5
  6
octave:50> C=[A b]
C =

  1  2  5
  3  4  6
```

- `+ - *` bezeichnen Matrixaddition,- substraktion und -multiplikation.
- `A'` transponiert und konjugiert A.
- `A.'` transponiert A.
- Elementweise Operatoren werden ausgedrückt durch einen dem Operator vorangestellten Punkt: `.* ./ .- .^` usw.

```
A = [ 1 2; 3 4]; A.^2 % Elementweise potenzieren.
A^2 % Quadrierung der Matrix
```

- Indizierung und Slicing:
  - `v(k)` ist das k-te Element des Zeilen- oder Spaltenvektor, wobei Indices bei 1 beginnen (nicht bei 0, wie bei Java/C/C++).
  - `A(k,l)` ist das Matrixelement a_{kl}
  - `v(m:n)` ist das "Slice" des Vektors `v` vom m-ten bis zum n-ten Eintrag.
  - `A(k,:)` ist die k-te Zeile der Matrix A.
  - `A(:,l)` ist die l-te Spalte der Matrix.
- Wichtige Matrixoperationen:
  - `A(k,:) = zv` weist der k-ten Zeile von A den Zeilenvektor zv zu.
  - `A(k,:) = []` löscht die k-te Zeile aus A.

  - `A(:,l) = sv` weist der l-ten Spalte von A den Spaltenvektor sv zu.
  - `A(:,l) = []` löscht die l-te Spalte aus A.

- Wichtige Matrixfunktionen:
  - `length(v)` gibt die Länge des Vektors v zurück.
  - `[Zeilen, Spalten]=size(A)` gibt die Grösse von A zurück.
  - `eye(m,n)` erzeugt eine $m \times n$ Matrix mit 1-ern auf der Diagonalen.
  - `zeros(m,n)` erzeugt eine Nullmatrix.
  - `ones(m,n)` erzeugt eine Matrix, deren Elemente gleich 1 sind.
  - `rand(m,n)` erzeugt eine Zufallsmatrix mit gleichverteilten Einträgen. Andere Verteilungen, z.B. Normalverteilung siehe `help randn`.

4 Kontrollstrukturen

4.1 Schleifen

Die Syntax von for- und while Schleife soll anhand eines Beispiel vorgestellt werden:

```
for i=0:-2:-10
printf("%d\n",i);
end

while k<K
k=k+1;
end
```

4.2 if- und switch Anweisungen

Analog zu Schleifen werden hier zwei Beispiele vorgeführt:

```
if x==0
   error('x ist 0!')
else
   y=1/x;
end

switch pnorm
  case 1;
     sum(abs(v))
  case inf;
     max(abs(v))
  otherwise
     sqrt(v'*v)
end
```

4.3 Funktionen

Funktionen werden innerhalb von Textdateien mit dem Suffix **.m** definiert. Die Unterschiede zu Skripten bestehen im Wesentlichen in folgenden Punkten:

- Das erste Code Fragment (keine Kommentare) innerhalb einer *Skriptdatei* ist keine Funktionsdeklaration. (ist die erste Zeile z.B `1;` so wird eine .m Datei zu einer Skriptdatei).
- Funktionen können mit Argumenten aufgerufen werden.

- Variable innerhalb einer Skriptdatei und ausserhalb von Funktionen sind *global*.
- Alle innerhalb der Funktionen verwendeten Variablen sind *lokal*.
- `global name` deklariert `name` als globale Variable.

  ```
  function foo(arg1,arg2)
  global N %vdeklariere N als globale Variable
  tue was..
  end
  ```

  Wird N in der Funktion verändert, so ist die Änderung im gesamt Workspace sichtbar.
- Der Funktionsname stimmt i.A. mit dem Namen der Funktionsdatei überein (In MATLAB muss das so sein); Ist das nicht der Fall, so muss die Datei, in der die Funktion definiert ist, explizit über den Befehl `load` geladen werden.
- In einer Skriptdatei können mehrere Funktionen definiert werden. (In MATLAB ist dieses Feature nicht vorhanden!!)
- Dokumentation: *Octave* ist in der Lage, den ersten Kommentarblock nach der Deklaration des Funktionskopfes zu erkennen und beim Aufruf von `help` für diese Funktion auszugeben. Dies gilt für Funktionen, die in Skript- und in Funktionsdateien definiert sind.

Beispiel:

```
function [R_ortho,d]=Orthogonalize(R)
%%Orthogonalize a square matrix based on svd.
%%returns orthogonal Matrix and its determinant.
if(rows(R) != columns(R))
    error ("Orthogonalize: Orthogobalization requires a square matrix.");
end

[u,s,v]=svd(R); %Funktion liefert mehr als ein Parameter zur"uck.

      % recalculate R with all singular
      % values set to 1
      R_ortho = u*v';
      d=det(R_ortho);
endfunction
```

5 Fehlermeldungen

- **Parse error** meldet einen Syntaxfehler an der mit (x) markierten Stelle:

  ```
  [ 1 2 .. 3 4]
  parse error:
  >>> [ 1 2 .. 3 4]
            ^
  ```

- **Runtime Error** meldet einen Laufzeitfehler, dessen Inhalt eine Art Backtrace für die Befehlsreihe angibt, um die Quelle des Fehlers leicht verfolgen zu können.

  ```
  octave:1> a=[1 2 3]; %deklaration eines Zeilenvectors
  octave:2> a*b      % Multiplikation mit b: nicht definiert
  error: 'b' undefined near line 2 column 3
      %colum 3: dritte Stelle in der Zeile
  error: evaluating expression near line 2, column 3
  ```

```
    % line 2: aktuelle Zeile. s. Prompt
error: evaluating binary operator ‘*’ near line 2, column 2
error: evaluating assignment expression near line 2, column 2
```