Wichtige Scilab-Befehle

In der Konsole (Hauptfenster) werden Befehle direkt eingegeben oder Programme ausgeführt, unter dem Editor (aufzurufen mittels `edit` bzw. des entsprechenden Buttons in der Menüleiste) werden Programme geschrieben. Werden diese abgespeichert und anschließend in der Konsole mittels z.B. `getf('mein_programm.sci')` eingebunden, kann man das Programm von dort aus (oder in anderen Programmen oder Skriptdateien) ausführen.

Es gibt zwei Arten von Dateien: Funktionsdateien mit der Endung `.sci` und Skriptdateien (Makros) mit der Endung `.sce`.
Funktionsdateien beginnen z.B. mit `function [y]=mein_programm(argumente)` und enden mit `endfunction`, wohingegen Skriptdateien, die ebenfalls im Editor geschrieben werden, keine geschlossenen Funktionen sein müssen, sondern einfach eine Ansammlung an Befehlen enthalten (um etwa Matrizen zu definieren und anschließend Funktionen aufzurufen). Siehe die Beispieldateien am Ende. Man kann die Skriptdatei ausführen, indem man entweder bei aktivem Editor-Fenster Strg.+ℓ drückt oder in der Konsole `exec('mein_skript.sce')` eingibt (das druckt allerdings alle darin enthaltenen Befehle aus).
Im Gegensatz zu Funktionen sind im Skriptfile definierte Variablen auch nach dem Ausführen des Skripts noch präsent (*globale Variablen*; abgesehen von Input- und Outputargumenten sind Variablen innerhalb einer Funktion nur *lokal*).

Generieren von Vektoren und Matrizen:

- Matrix von Hand eingeben: `A = [1 2.0 3 4; 5.3 6 7 -1]`. Einzelne Komponenten werden durch `,` oder Leerzeichen getrennt, Zeilenumbrüche mittels `;` . Zeilenvektoren z.B. durch `x=[0 9.7 1.2];`, Spaltenvektoren durch `x=[0; 9.7; 1.2];` .

- Einzelne/mehrere Komponenten von Matrizen bzw. Vektoren manipulieren:
  `A(2,3)=-5;` , `A(:,2)=[0;6]` (`:` steht also für "ganze Zeile" bzw. "ganze Spalte"); beachte, dass im letzten Beispiel ein Spaltenvektor zum Einsatz kommen muss, da die 2. *Spalte* von A definiert wird. `A(2,2:4)=[1, 2, 5]` definiert die Komponenten $a_{22}, \ldots, a_{24}$ von A.

- Erzeugen eines Vektors $x \in \mathbb{R}^n$ mit $x_i = i$ mittels `x = 1:n` . Will man eine andere Einteilung, ist auch z.B. `x = 2.5:0.1:5.5` möglich (dann $x = (2.5, 2.6, \ldots, 5.5)$).

- `A = rand(m,n)` erzeugt eine Zufallsmatrix mit Dimensionen m mal n. Analog erzeugt `b = rand(m,1)` einen Zufalls-Spaltenvektor der Länge m.

- `x = linspace(a,b,n)` erzeugt einen Vektor mit n Komponenten, die äquidistant zwischen a und b liegen, d.h. $x_i = a + (i-1)\,h$, $h = (b-a)/(n-1)$.

  Beispiel: Plotten der Funktion $f(x) = \cos(x)^2\, sin(x)$ auf dem Intervall $[0, 2\pi]$:
  ```
  x = linspace(0, 2*%pi, 100);
  y = cos(x).^2 .* sin(x);
  plot2d(x,y);
  ```

- `eye(n,n)` erzeugt die Einheitsmatrix I_n.

- `zeros(m,n)` und `ones(m,n)` erzeugen Matrizen in $\mathbb{R}^{m\times n}$, die mit lauter Nullen bzw. Einsen aufgefüllt sind. Wichtig zur Konstruktion spezieller Matrizen:

Beispiel: Generieren der Stokesmatrix

$$S := \begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \in \mathbb{R}^{(n+m)\times(n+m)}$$

mittels `S = [A B';B zeros(m,m)]` .

- `diag` hat zweierlei Bedeutung, je nachdem, ob man es auf einen Vektor oder eine Matrix anwendet:
  `d = diag(A)` liefert einen Vektor `d`, der die Diagonalelemente von `A` enthält.
  `B = diag(d)` generiert eine $(n \times n)$–Matrix B mit Diagonalelementen d_i.
  Kombiniert man beide, so liefert `diag(diag(A))` eine Diagonalmatrix mit der Diagonalen von A.

Matrix-Vektor-Berechnungen:

- `[m,n] = size(A)` gibt die Dimensionen der Matrix A aus; man kann sie auch einzeln mittels `m = size(A,1)` und `n = size(A,2)` abfragen. Für Vektoren liefert `length(x)` die „Länge", was für Spaltenvektoren dasselbe ist wie `size(x,1)`.
- `norm(x,k)` berechnet die k–Norm $\|x\|_k$ des Vektors x, wobei $1 \le$ `k` $\le$ `%inf`. Analog für Matrizen.
- `s = sum(A,1)` summiert für $A \in \mathbb{R}^{m\times n}$ alle Spalteneinträge zusammen und liefert einen Vektor in $\mathbb{R}^{1\times n}$ mit Komponenten $s_j = \sum_{i=1}^{m} A_{ij}$. Analog summiert `sum(A,2)` in „Zeilenrichtung".

  Beispiel: Die Massenmatrix $M \in \mathbb{R}^{n\times n}$ soll durch *lumping* in eine Diagonalmatrix M_ℓ mit Einträgen $M_{\ell ii} = \sum_{j=1}^{n} M_{ij}$ verwandelt werden.
  Unter Scilab lautet das kompakt `Ml = diag(sum(M,2))`.
- Transponieren mittels `'`. (Vorsicht: Im Komplexen entspricht `'` dem $\overline{(\,)}^T$.)
- `x' * y` berechnet das (euklidische) Skalarprodukt $(x, y)_2 = x^T y$.
- **Wichtig:** Lösen von Gleichungssystemen mittels `\` : $Ax = b$ wird gelöst durch Eingabe von `x = A \ b;` (für $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ und $\det A \neq 0$).
- `rank(A)` berechnet den Rang der Matrix A, `inv(A)` die Inverse A^{-1} (sparsam verwenden!), `det(A)` die Determinante und `spec(A)` die Eigenwerte.
- `[Q,R] = qr(A)` liefert die QR-Zerlegung der Matrix $A = Q\,R$, `[L,U,P] = lu(A)` ihre LR-Zerlegung $P\,A = L\,U$ (P: Permutationsmatrix).

Oft verwendete Kommandos:

- `min`, `max`, `abs` $(= |\cdot|)$.
- `sqrt` $(= \sqrt{\ })$, `log` $(= \ln)$, `exp`, `sin`, `cos`, `tan`.
- `for i=1:n, ... , end` durchläuft die Schleife n mal. Die Kommas sind nur nötig, falls alles in eine Zeile geschrieben wird.
  Beachte, dass alle Umgebungen (wie `if` – `else`, `while`, `try` – `catch`, `select` – `case`) *immer* mit `end` beendet werden müssen (unter C ist das `{ ... }`).

- Mit `break` kann man eine `for`- oder `while`-Schleife vorzeitig abbrechen, indem man in der Schleife ein Abbruchskriterium festlegt, z.B. `if a > 10, break; end` .
  Mit `return;` statt `break;` verlässt man gleich die ganze Funktion.

- **Wichtig:** Der Befehl `find`, mit dem man einen kurzen und übersichtlichen Programmierstil erreichen kann:
  `ind = find(x > 0)` liefert einen Indexvektor `ind`, so dass komponentweise `x(ind)` > 0. Es geht auch `ind  = find((x > -2) & (x <= 10))`. Dann $-2 <$ `x(ind)` ≤ 10.
  Vorsicht, bei Gleichheitsanfragen `==` verwenden, *nicht* `=`.
  `find(x)` ist eine Kurzschreibweise für `find(x ~= 0)`, sucht also alle nichtverschwindenden Komponenten von `x`.

- Logische Verknüpfungen sind `~` („nicht"), `&` (und), `|` (oder). Ob eine Suche mittels `find` (siehe oben) überhaupt einen entsprechenden Index geliefert hat, lässt sich mittels `if ~isempty(ind), ...` überprüfen.

Vektorisierter Programmierstil:

`for`–Schleifen sind in Scilab relativ langsam, die entstehenden Programme außerdem schwierig zu lesen. Besser ist es, sich einen vektorisierten Programmierstil anzugewöhnen. Die Programme sind dann deutlich kompakter und werden unter Scilab sehr schnell ausgeführt.

- Komponentweise Multiplikation, Division, Potenzieren mittels `.*` und `./` und `.^` . Bei Multiplikation und Division müssen beide Vektoren/Matrizen dieselbe (!) Dimension haben: Für $x, y \in \mathbb{R}^n$ ist `w = x .* y` ein Vektor in $\mathbb{R}^n$ mit $w_i = x_i y_i$.

  Beispiel: Statt umständlich mehrere `for`–Schleifen zu verwenden, kann die Matrix $XS^{-1} = \mathrm{diag}(x)\,\mathrm{diag}(s)^{-1}$ einfach generiert werden durch `diag(x ./ s)`.

- *Weiteres Beispiel:* Ein Eliminationsschritt im Gauß-Algorithmus würde lauten
  ```
  b(i+1:n,1) = b(i+1:n,1) - b(i,1) * A(i+1:n,i) / A(i,i);
  A(i+1:n,i:n) = A(i+1:n,i:n) - A(i+1:n,i) / A(i,i) * A(i,i:n);
  ```
  (spart zwei `for`-Schleifen!)

- Das Symbol `$` bedeutet "höchster vorhandener Index"; ist z.B. $x = (2, 0, 1, 3, 7)$, so liefert `x(3:$)` die Ausgabe `1 3 7` (`$` könnte im obigen Beispiel also anstelle von `n` verwendet werden).

Textausgabe und Graphiken:

- `printf('Text');` (eine C-Adaption) gibt den `Text` aus; wenn man vorher z.B. `ausgabetext = 'Programm erfolgreich beendet';` definiert hat, kann man auch `printf(ausgabetext);` (also ohne Striche) verwenden.
  Zeilenumbrüche durch `\n`, Tab-Abstände durch `\t`, ganze Zahlen können ausgegeben werden durch `%d`, floating point-Zahlen durch `%f` und z.B. Zahlen in Exponentialschreibweise mit zwei Nachkommastellen durch `%.2e`.

  Beispiel: Die Ausgabe des Resultats am Programmende könnte lauten
  ```
  printf('\nRel. Residuum nach %d Iterationen betraegt %.2e\n\n', k, relres);
  ```

- `plot2d(x,y)` plottet eine 2D-Graphik, wobei `x` als x- und `y` als y-Koordinatwerte verwendet werden (müssen deshalb von gleicher Länge sein).
  Erweiterbar durch Optionen wie
  `plot2d(x,y,style=1,rect=[xmin ymin xmax ymax],logflag='nl')`, wo im Bereich $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ eine durchgehende Linie auf einer semilogarithmischen Skala geplottet wird (für weitere Details siehe `help plot2d`).
  Mit `xlabel('...')`, `ylabel('...')`, `title('...')` lassen sich die x-Achse, y-Achse und der Titel der Graphik beschriften.
  Durch den Befehl `set(gca(),"auto_clear","off")` (Matlab-Äquivalent: `hold on`) werden alle nachfolgenden Plot-Befehle in dieselbe Graphik ausgegeben; das ermöglicht das Übereinanderplotten von Ergebnissen.

Weitere nützliche Kommandos:

- Mit `help <Funktion>` kann man sich den Hilfetext (Erklärungen zur Verwendung der Argumente etc.) zu einer bestimmten Funktion anzeigen lassen.
  Kennt man die gesuchte Funktion nicht, sondern nur einen Schlüsselbegriff, kann man mit `apropos <Begriff>` danach suchen.

- Fehlermeldungen bzw. Warnungen ausgeben: `error('...');` bzw. `warning('...');` (Eine Fehlermeldung führt zum Programmabbruch, eine Warnung nicht.)

  Beispiel: `if find(x<0), error('x hat negative Komponenten'); end`

- `clc` „reinigt" das Konsolenfenster, `clear x` löscht die Variable namens `x`, `exit` schließt Scilab.

- Mit der Pfeiltaste ↑ kann man durch frühere Befehle blättern, mit `abc` plus ↑ nur durch solche, die mit `abc` beginnen.

- Auskommentieren erfolgt mittels `//`.

- Zahlenformat einstellen: `format('v',8)` bewirkt z.B., dass 5 Nachkommastellen angezeigt werden. e-Format: `format('e',zahl)`.

- Vordefinierte Konstanten sind: `%i` ($= i$), `%e` ($= e$), `%pi` ($= \pi$), `%eps` (Maschinengenauigkeit, $\approx 2.2 \cdot 10^{-16}$), die logischen Werte `%t` ("wahr") und `%f` ("falsch"), `%inf` ($= \infty$), `%nan` ("not a number").

- Messen der Rechendauer in Sekunden: `tic, <Befehle/Programmaufruf>, toc` .
  (Achtung, das Ergebnis hängt auch von der CPU-Auslastung ab!)

- Einfache Funktionen können auch direkt auf der Konsole definiert werden (sog. *inline functions*): `deff('z=f(x,y)','z=3*x^2-2*y');` definiert $f(x, y) = 3x^2 - 2y$ und kann anschließend aufgerufen werden, z.B. `f(2,3)`.

Referenzen zu weiteren Scilab-Einführungen:

- `http://www.neng.usu.edu/cee/faculty/gurro/Software_Calculators/Scilab_Docs/SCILAB_Scripts.htm`
  Ansammlung von Skripten (`.sce`-Dateien) zu diversen Anwendungen.

- `http://comptlsci.anu.edu.au/Scilab/primer.pdf`
  Simpel gehaltene, elementare Einführung.
- `http://www.neng.usu.edu/cee/faculty/gurro/Software_Calculators/Scilab_Docs/Scilab_documents/ProgrammingWithSCILAB.pdf`
  Einführung mit vielen Beispielprogrammen und näheren Erläuterungen zu Umgebungen wie `if ... end`.

Beispiele für Scilab-Dateien:

Skriptdatei mit Endung `.sce`:

```
// Definiere eine spd. Matrix A, eine rechte Seite b und loese
// anschliessend das Gleichungssystem mittels cg-Verfahren.
n = 100;
h = 1 / (n-1);

A = 2 * eye(n,n);
for i = 2 : n
   A(i,i-1) = -1;
   A(i-1,i) = -1;
end
A = 1/h^2 * A;
b = rand(n,1);

x = cg(A,b,1e-9);            // Siehe unten.

relres = norm(b - A*x,2) / norm(b,2);
printf('\nDas relative Residuum betraegt %.2e.\n', relres);
```

Funktionsdatei namens `cg.sci`:

```
function x = cg(A,b,tol,kmax,x)
// Loest das lineare System A x = b mittels cg-Verfahren.
// Beachte: A muss spd. sein.
// tol:  Toleranz fuer das relative Residuum (optional).
// kmax: maximale Anzahl von Iterationsschritten (optional).
// x:    Startvektor (optional).

[n,m] = size(A);
if (n ~= m) | (n ~= size(b,1))
   error('Dimensionen stimmen nicht.');
end

nargin = argn(2);      // Anzahl uebergebener Argumente.
if nargin < 3, tol = 1e-8;     end
if nargin < 4, kmax = n;       end
if nargin < 5, x = zeros(n,1); end
```

```
// Initialisierung.
gi  = A*x - b;
ng0 = gi' * gi;
ngi = ng0;
w   = gi;

// Iteration.
for i = 1 : kmax
    y    = A*w;
    rho  = ngi / (w' * y);
    x    = x  - rho * w;
    gi1  = gi - rho * y;
    ngi1 = gi1' * gi1;

    // Konvergenzcheck.
    if sqrt(ngi1/ng0) < tol
        return;
    end

    gamm = ngi1 / ngi;
    w    = gi1 + gamm * w;

    // Update i -> i+1.
    gi  = gi1;
    ngi = ngi1;
end

endfunction
```