Programmierpraktikum im WS 2011/2012

Denny Otten, V5-142, dotten@math.uni-bielefeld.de

3. Vektorisierung und Laufzeitoptimierung

1 Wiederholung: Kontrollstrukturen

Bevor das Erlernte vom letzten Zettel zu schnell wieder verschwindet, ein paar einfache Fingerübungen, die die Kontrollstrukturen wiederholen.

Aufgabe 1. *Schreibe eine Funktion, die als Eingabe eine Funktion, ihre Ableitung (beide als Funktionshandles) und einen Punkt x haben soll. Dann soll sie für alle Schrittweiten $h = 0.8^n,\ n = 0, \dots, 170$ den Fehler, berechnen, wie weit der Differenzenquotient von dem tatsächlichen Wert der Ableitung entfernt ist, berechne also $|\frac{f(x+h)-f(x)}{h} - f'(x)|$ für $h = 0.8^n,\ n = 0, \dots, 170$. Probiere mindestens mit $f(x) = x^2$ an der Stelle $x = 2$.*
Zeichne den Fehler mit `loglog` *(siehe* `help loglog`*). Was fällt Dir auf?*

Aufgabe 2. *Benutze in dieser Aufgabe* `for`*–Schleifen und* ***nicht*** *die eingebaute Matrixmultiplikation!*
Definiere eine Funktion `innerprod`*, die das euklidische innere Produkt zweier Vektoren $u, v \in \mathbb{R}^m$ berechnet, also $u^T v = \sum_{i=1}^{m} u_i v_i$.*
Definiere eine Funktion `matmult`*, die das Matrixprodukt zweier Matrizen berechnet, d.h.* `matmult(A,B)=A*B`*.*
(Benutze eine `if`*-Abfrage um fehlerhafte Eingaben abzufangen!)*

Aufgabe 3 (Zusatzaufgabe zum Knobeln)**.** *Ein magisches Quadrat der Größe n^2 ist eine Matrix mit n Zeilen und n Spalten, die jede natürliche Zahl von 1 bis n^2 genau einmal enthält und die die Folgenden Eigenschaften hat:*
Die Summe der Einträge jeder beliebigen Zeile ist dieselbe, wie die Summe der Einträge jeder beliebigen Spalte und auch dieselbe wie die Summe der Einträge auf jeder der beiden Diagonalen. Man kann magische Quadrate ungerader Größe nach folgenden Regeln erzeugen:

Regel 1 *Die 1 kommt in das Feld unmittelbar unter der Mitte des Quadrates.*

Regel 2 *Wenn die Zahl x in der Zeile i und der Spalte k positioniert wurde, dann bestimme neue Zeilen- und Spaltenindizes wie folgt: Zuerst setze i auf $i + 1$ und k auf $k + 1$. Sind dies ungültige Zeilen- oder Spaltenangaben, so verwende die Regel 4 um gültige Zeilen- und Spaltenangaben zu erhalten. Ist das Feld mit diesen Indizes bereits belegt, so verwende Regel 3 um Indizes für ein leeres Feld zu bestimmen. Platziere $x + 1$ hier und verwende wieder Regel 2, bis alle Zahlen platziert wurden.*

Regel 3 *Setze den Zeilenindex auf $i + 1$ und den Spaltenindex auf $k - 1$. Sind diese neuen Koordinaten ungültig, so benutze Regel 4, ist das Feld mit diesen Koordinaten bereits besetzt, so wende Regel 3 erneut an.*

Regel 4 *Die Zeilen- und Spaltennummern sind $1, \dots, n$. Ergibt sich eine zu kleine Zeilen- oder Spaltennummer, so setze die entsprechende auf n. Ergibt sich eine zu große Spalten- oder Zeilennummer, so setze die Nummer auf 1.*

Implementiere diese Regeln und erzeuge so verschieden große magische Quadrate. Kontrolliere, dass es sich tatsächlich um magische Quadrate handelt.

2 Optimierung von Programmen

Vektorisierung

Wie schon im Tutorial gesehen, kann man in Matlab oft `for`-Schleifen dadurch einsparen, dass man Operatoren benutzt, die elementweise auf Matrizen opererieren, wie

`.* (Multiplikation)  ./  .\ (Division von links bzw. rechts)  .^ (Exponentiation)`

(Achtung: mit `.'` wird die Transposition von Matrizen bezeichnet!) Man kann also statt einer `for`-Schleife

```
for n=1:N
   y(n) = x(n)^2;   % greift auf jedes Element einzeln zu
end
```
einfach schreiben

```
y = x.^2;           % so ist es doch uebersichtlicher!
```

Außerdem können auch die logischen Operatoren `&` (und) und `|` (oder) auf Matrizen und Vektoren arbeiten (beachte, dass `&&` und `||` dies *nicht* können!): Als Beispiel bestimme für eine zufällige $N \times N$-Matrix die Anzahl der Einträge im Intervall $[0.3, 0.7]$

```
M=rand(N);
anzahl=0;
for i = 1:N
  for j = 1:N
    anzahl=anzahl+(M(i,j)>=0.3&&M(i,j)<=0.7);
  end
end
```

```
M=rand(N);
anzahl=sum(sum(M>=0.3&M<=0.7));
```

An diesem Beispiel kann man schon bei relativ kleinen Matrizen enorme Geschwindigkeitsunterschiede feststellen!
Vektorisierung wird häufig benutzt, um den Code übersichtlicher zu machen. Manchmal kann man auch die Ausführgeschwindigkeit erhöhen (siehe auch: `www.mathworks.com/support/tech-notes/1100/1109.html`).

Aufgabe 4. *Vektorisiere die Funktion* $\frac{\sin(x^2+y^2)}{x^2+y^2}$ *und zeichne diese mit* `ezmesh`.

Aufgabe 5. *Miss die Geschwindigkeit der folgenden Funktion*

```
function d = loop_dist(x,y,z)
  n=length(x);
  for k=1:n
    d(k)= sqrt(x(k)^2+y(k)^2+z(k)^2);
  end
  d = min(d);
end
```

Listing 1: `loop_dist.m`

für verschiedene Vektorgrößen in mehreren Durchläufen mit `tic; <Anweisungen>; toc;`. *Schreibe die obige Funktion in eine vektorisierte Version* `vec_dist` *um und vergleiche die Geschwindigkeiten. Schreibe dazu ein Matlab Skript, in dem die Vektoren* `x,y,z` *definiert werden (etwa mit* `rand`*), die Funktionen* `loop_dist` *und* `vec_dist` *aufgerufen und jeweils die Laufzeit gemessen wird.*

Vorreservierung von Speicher

Das Programm in Aufgabe 5 ist deshalb so langsam, weil Matlab in jedem Schleifendurchlauf neuen Speicher für den Vektor `d` reservieren muss. Wenn man diesen vor der Schleife z.B. mit
`d = zeros(n,1);` reserviert, bzw. bereitstellt, so beschleunigt sich der Code erheblich.

Aufgabe 6. *Beschleunige die Funktion* `loop_dist` *nur dadurch, dass Du vorher den Speicher reservierst (also musst Du vorher eine Matrix* `d` *der passenden Größe erzeugen!). Vergleiche die Geschwindigkeiten von* `loop_dist` *ohne Speicherreservierung,* `loop_dist` *mit Speicherreservierung und der vektorisierten Version* `vec_dist` *aus Aufgabe 5.*

Aufgabe 7. *Versuche folgenden Code zu beschleunigen und miss den Geschwindigkeitsunterschied.*

```
x=-0.99; y=0.05;
a(1)=1;  b(1)=0;
for k=2:8000
  a(k) = x*a(k-1) - y*b(k-1);
  b(k) = y*a(k-1) + x*b(k-1);
end
```

Eine Zusatzaufgabe:

Aufgabe 8. *Vektorisiere den folgenden Code unter Benutzung der Funktion* `meshgrid` *(siehe:* `help meshgrid`*)*

```
for i = 1:500
  for j = 1:500
    r(i,j) = sqrt(i^2+j^2);
  end
end
```

Schreibe ein Skript um die durchschnittliche Laufzeit von 10 Durchläufen für die originale und die vektorisierte Version des Codes zu vergleichen.

Der Profiler

Der Matlab Profiler hilft, Stellen im Programmcode zu finden, für die Matlab besonders lange braucht. Er wird mit `profile on` in der Kommandozeile gestartet. Danach wird der zu analysierende Code ausgeführt. Mit `profile viewer` wird eine detaillierte Übersicht über die Laufzeiten angezeigt. Der Profiler hat noch viele weitere Optionen, die mit `help profile` bzw. in der Online Hilfe finden lassen unter
`Matlab->Desktop Tools and Development Environment->Tuning and Managing M-Files->Profiling for Improving Performance`

Aufgabe 9. *Speichere den folgenden Code in der Skriptdatei* `ops.m`

```
rand('twister',1), randn('state',1)
N=100; niter=100;
a=100*rand(N); b=randn(N);
for n = 1:niter
    a+b;
    a-b;
    a.*b;
    a./b;
    sqrt(a);
    exp(a);
    sin(a);
    tan(a);
end
```

Listing 2: unter `ops.m` abspeichern

und analysiere die Laufzeit einzelner Operationen wie folgt mit dem Profiler:

```
>> profile on
>> ops;
>> profile viewer
>> profile off
```

Mache Dich mit den Optionen des Profilers vertraut. Benutze dazu auch die Beispiele aus Aufgabe 7, die Du bereits mit `tic; ... ; toc` *analysiert hast und vergleiche die Ergebnisse. Fällt Dir dabei etwas auf?*

Der Debugger

Der Matlab Debugger ist äußerst hilfreich um die Funktionsweise von Matlab Programmen zu verstehen und um Programmierfehler zu suchen. Er ermöglicht es, an bestimmten Stellen im Code Haltepunkte zu setzen und schrittweise durch den Code zu laufen. Variablen können dann jeweils an der Debugger-Kommandozeile `K>>` ausgewertet und sogar verändert werden.
Im Matlab Editor werden Haltepunkte durch klicken rechts neben der Zeilennummer gesetzt bzw. gelöscht und alle weiteren Kommandos sind per Mausklick auf die Debugger-Toolbar ausführbar.

Wichtige Debuggerbefehle für das Kommandofenster (für diejenigen, die das lieber mögen):

`>> dbstop`	Setze Stopppunkt (z. B. `>> dbstop if error`)
`K>> dbclear`	Deaktiviert Stoppunkt
`K>> dbcont`	Setzt Berechnung bis zum nächsten Stopppunkt fort
`K>> dbstatus`	Listet alle Stopppunkte auf
`K>> dbstep`	Macht einen Schritt im Code
`K>> dbtype`	Listet den Quellcode mit Zeilennummern
`K>> dbquit`	Beendet den debugger

Weitere Informationen über den Debugger finden sich in der Online Hilfe unter `help debug` oder unter `doc debug`.

Rekursive Funktionen

... Oder: Um Rekursion zu verstehen, muss man Rekursion verstehen...

Rekursive Funktionen sind Funktionen, die sich selbst aufrufen. An ihnen kann man wunderbar den Matlab Debugger kennenlernen. Eine typische rekursive Funktion sieht so aus:

```
function erg = rfun(in)
  ...
  erg = rfun(in-1)     % rufe rfun in der Definition von rfun
                       % mit einem kleineren input parameter auf
end
```

Aufgabe 10. *Versuche den folgenden **rekursiven** Code zur Berechnung der Determinante (z.B. mit Hilfe des Debuggers) zu verstehen und versehe den Code mit entsprechenden Kommentaren.*

```
function d = rdet(A)
  n = length(A);
  if n==1
    d=A(1,1);
  else
    d = 0;
    sgn = 1;
    for i=1:n
      d = d + sgn * A(1,i) * rdet(A(2:n, [1:i-1, i+1:n]));
      sgn = -sgn;
    end
  end
end
```

Listing 3: `rdet.m`

Ermittle in einer Schleife über die Dimension n der Matrix ($n \leq 9$), wie lange das Programm jeweils zur Berechnung der Determinante benötigt. Stelle den Zusammenhang zwischen Matrixgröße und Laufzeit grafisch dar. Passe dabei die Skalierung der Achsen im Menü des Grafikfensters unter `Edit->Axes Properties` *so an, dass der Zusammenhang gut sichtbar wird. Vergleiche das mit der in Matlab eingebauten Funktion* `det`.

Aufgabe 11. *Implementiere die Fakultät $fac(n) = 1 \cdot ... \cdot n$ **rekursiv** in einer Funktion* `rfac` *und iterativ mit einer Schleife in einer Funktion* `ifac`. *Vergleiche jeweils für verschiedene n die Laufzeit beider Funktionen und stelle wie in der vorherigen Aufgabe den Zusammenhang zwischen n und der Rechenzeit grafisch dar.*

Die Kochkurve (oder Kochschneeflocke)

Mit Hilfe rekursiver Funktionen lassen sich selbstähnliche Objekte, wie Fraktale elegant erzeugen.

Aufgabe 12. *Die folgende Funktion zeichnet eine Kochkurve für verschiedene Rekursionstiefen. Speichere die beiden folgenden Funktionen in einer Datei koch.m und teste ihn mit dem Aufruf*
`>> koch(4)`.

```
function koch(Tiefe)
%KOCH generates the Koch-curve
% KOCH(TIEFE): erzeugt rekursiv eine Kochkurve vorgegebener Tiefe (max. 6)
  if Tiefe > 6
    Tiefe = 6;
    disp('Rechne mit max. Tiefe 6');
  end
  links = [0;0];
  rechts = [1;0];
  N = ceil(Tiefe/2);
  for k=1:Tiefe
    subplot(N,2,k)          % subplots erzeugen
    kochfun(links, rechts, k) % Kochkurve erzeugen
    axis('equal'); title(['Kochkurve: Stufe=' num2str(k)]);
  end
end

function kochfun(links, rechts, stufe)
%KOCHFUN
% KOCHFUN(LINKS, RECHTS, STUFE)
% links, rechts sind die Koordinaten des linken bzw. rechten Endpunktes
% stufe ist die Rekursionstufe
  if stufe == 0 % Ende der Rekursion wenn Stufe = 0
    plot([links(1),rechts(1)],[links(2),rechts(2)]); % Verbinde links
    hold on                                          % mit rechts
  else
    RS = (sqrt(3)/6)*[0, 1; -1, 0];   % Rotations/Skalierungs Matrix.
    mitte_links = (2*links + rechts)/3;
    mitte_rechts = (links + 2*rechts)/3;
    oben = (links + rechts)/2 + RS*(links - rechts);
    % Rekursiver Aufruf von kochfun
    kochfun(links, mitte_links, stufe-1)      % Linker Zweig
    kochfun(mitte_links, oben, stufe-1)       % Mittlerer linker Zweig
    kochfun(oben, mitte_rechts, stufe-1)      % Mittlerer rechter Zweig
    kochfun(mitte_rechts, rechts, stufe-1)    % Rechter Zweig
  end
end
```

Listing 4: `koch.m`

Zusatz: Iterative Version der Kochkurve

Der folgende Code ist eine iterative Version der Kochkurve, die komplexe Zahlen benutzt und wesentlich schneller ist als die rekursive Version.

```
function koch_fast(Tiefe)
% schnelle Version der Kochkurve
  r3=sqrt(3)/2;
  z=[0, 0.5+i*r3, 1]; % links, mitte, rechts
  N=ceil(Tiefe/2);
  for k=1:Tiefe
    r=diff([z 0])/3;
    z=[z; z+r; z+(1.5+i*r3)*r; z+2*r];
    z = z(:).'; % Transposition .'
    % Kochkurve zeichnen
    subplot(N,2,k)
    plot([z, 0])
    axis('square'); title(['Stufe=' num2str(k)]);
  end
end
```

Listing 5: `kochfast.m`