Computational Physics II - Übung 1

Harald Krause, Oliver Hatt

4. Oktober 2013

Aufgabe 1.1

Es sollte eine reelle, symmetrische Matrix A mit Hilfe der Jacobi Methode in Diagonalform $D = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$ überführt werden. Alle Bezeichnungen werden aus dem Script Computational Physics II, Ulli Wolff, WS 2013/14, Modul 22.1 Masterstudiengang Physik entnommen.

Hierzu werden sukzessive orthogonale Drehungen P_{pq} in jeder möglichen p-q-Ebene durchgeführt. Dies geschieht derart, dass bei jeder Drehung immer in den zwei Nicht-Diagonalelementen a_{pq} und a_{qp} Null ensteht.

$$a'_{pq} = a'_{qp} = (c^2 - s^2)a_{pq} + sc(a_{pp} - a_{qq}) \stackrel{!}{=} 0$$

Für die Implementierung wurden wie vorgeschlagen die Formeln (1.27) bis (1.31) mit den auf Seite 9 gegebenen Definitionen von c, s und τ verwendet. Die dazu nötigen Umformungsschritte werden in Teilaufgabe c) erläutert.

Bei der Drehung in der p-q-Ebenen $A_{neu} = P_{pq}^T A P_{pq}$ verändert die p-te und q-te Zeile sowie Spalte. Da die Matrix A symmetrisch ist, und diese Eigenschaft bei Drehung erhalten bleibt, genügt für der Berechnung der resultierenden Elemente ein Durchlauf aller Zeilen. Die Spalteneinräge können so direkt übernommen werden. Es lässt sich nun beobachten wie pro Iteration die Nichtdiagonalelemente gegen Null streben.

Die Funktion `doPpq` erzeugt bei jeder Drehung (jedes p-q-Paar) eine immer mehr diagonalisierte Matrix, nach dem oben beschriebenen Verfahren. Die Iterationen über alle p-q-Paare wird in der Funktion `myEig` durchgeführt. Das Programm bricht die Berechnung ab, sobald alle Nichtdiagonalelemente auf Maschinengenauigkeit Null geworden sind, oder die maximale Anzahl der Itterationen erreicht wurde. Wird keine maximale Anzahl angegeben, dann diese auf 1000 Iterationen begrenzt.

doPpq.m

```
1  function [ Aneu , P ] = doPpq( A, p, q )
2  %
3  % [ Aneu , P ] = doPpq( A, p, q )
4  %   Aneu = P' A P
5  % this function is used to diagonalize the matrix A in the
6  % p-q-plane. (p has to be smaller than q!)
7  % next to the rotated matrix A the rotation matrx P is returned
8
9  global flop;
10 if (exist('flop','var')== 0)
11         flop = 0;
12         fprintf('set global flop = 0');
13 end
14 if ~(p<q), error('p muss kleiner q sein!'); end
15
16 theta = (A(q,q)- A(p,p))./(2*A(p,q));
17 flop = flop + 2;
18
19 if (isfinite(theta))
20         t = sign(theta)./(abs(theta) + sqrt(theta.^2 + 1));
21         flop = flop + 2; % wie viele flops sind sqrt()?!
22 else
23         t = 0;
24 end
25
26 c   = 1./sqrt(t.^2 + 1); % 2 flops
27 s   = t.*c;              % 1 flop
28 tau = s./(1+c);          % 1 flop
29 flop = flop + 4;
30
31 Aneu = A;
32 for i = 1:size(A,1)
33         Aneu(i,p) = A(i,p) - s *( A(i,q) + tau* A(i,p) ); % 2 flops
34         Aneu(i,q) = A(i,q) + s* ( A(i,p) - tau* A(i,q) ); % 2 folps
35
36         Aneu(p,i) = Aneu(i,p);
37         Aneu(q,i) = Aneu(i,q);
38
39         flop = flop + 4;
40 end
41
42 Aneu(p,p) = A(p,p) - t* A(p,q);
43 Aneu(q,q) = A(q,q) + t* A(p,q);
44 flop = flop + 2;
45
46 Aneu(p,q) = 0; Aneu(q,p) = 0;
47
48 P = speye(size(A));
49 P(p,p) = c; P(q,q) = c;
50 P(p,q) = s; P(q,p) =-s;
51 end
```

myEig.m

```
function [ D, V, iter ] = myEig( A, maxIter )
%
%   D = myEig( A );
%  calculates a diagonal matrix with eigenvalues of A on the diagonal
%
%  [ D, V, iter ] = myEig( A, maxIter )
% optional arguments:
%      maxIter:     maximum number of iterations for solving the problem
%                   default = 1000
%
% other return values:
%      V:     product of successive rotation matrixes P_i
%             P1*P2*P3*...*Pi
%      iter: number of preformed iterations
%

if nargin < 2
    maxIter = 1000; % set default argument
end

D = A;
V = eye(size(A)); % V = P1 P2 P3;

% 1 iteration = loop over all pq's
for iter = 1:(maxIter)

    nd = D( eye(size(D))==0 ); % list of nondiagonal elements
    if (abs(nd) == 0)          % if all equal zero
        break;                 % stop iteration
    end

    for q=2:size(A,1)
        for p=1:q-1

            [D, P] = doPpq(D,p,q); % apply pq-rotation
            V = V*P;               % update V

        end % p
    end % q
end % iter
end % function
```

a)

Die Routine wird mit einer zufälligen, symmetrischen Matrix getestet und die Resultate mit der MATLAB-eigenen Funktion `eig` verglichen.

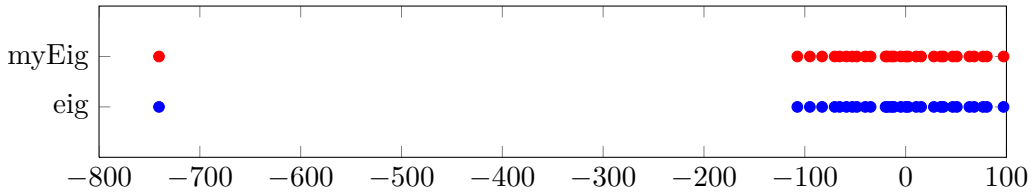
```
clc; clear all;
n = 5;              % set dimension
maxV  = 50;         % matrx entries between -50 and 50
randM = (maxV/2)*rand(n) - maxV/2; % creates nxn-matrix
A = randM + randM';                % returns symmetric matrix

D = myEig(A)
d = eig(A)
```

Das obige Skript zeigt für $n = 5$ die gleichen Eigenwerte der beiden Funktionen.

`myEig`	-144.8011	-30.1247	-12.2197	2.6067	17.7609
`eig`	-144.8011	-30.1247	-12.2197	2.6067	17.7609

auch die grafische Darstellung der Eigenwerte bei $N = 30$ zeigt die Übereinstimmung der Eigenwerten



b)

Die Eigenvektoren der Matrix A können berechnet werden indem die Gesammtdrehung $V = P_1 \cdot P_2 \ldots$ betrachtet wird. Die Matrizen $P_i = P_{12} \cdot P_{13} \ldots$ beschreiben jeweils die Drehung bei einer Iteration. Gesamtdrehung wird als zweites Argument der Funktion `myEig` zurückgegeben.

Da die Matrix V als Produkt der Einzeldrehungen gebildet wird, ist auch diese orthogonal und erfüllt die Relation $V^T A V = D$. Dies wird anhand des folgenden Beispiels gezeigt. Ist V orthogonal, so ergibt das Produkt aus Matrix und ihrer Transponierten die Einheitsmatrix.

$$A = \begin{bmatrix} -24.7959 & -15.2800 & -19.3882 & -19.4694 \\ -15.2800 & -23.9580 & -23.7974 & -9.7149 \\ -19.3882 & -23.7974 & -33.4879 & -24.2888 \\ -19.4694 & -9.7149 & -24.2888 & -27.7005 \end{bmatrix} \qquad V = \begin{bmatrix} 0.4621 & -0.2152 & 0.2685 & -0.8173 \\ 0.4296 & 0.7059 & -0.5494 & -0.1234 \\ 0.6038 & 0.1857 & 0.6008 & 0.4899 \\ 0.4871 & -0.6488 & -0.5149 & 0.2771 \end{bmatrix}$$

$$D = \begin{bmatrix} -84.8527 & 0 & 0 & 0 \\ 0 & -16.6335 & 0 & 0 \\ 0 & 0 & 0.4264 & 0 \\ 0 & 0 & 0 & -8.8826 \end{bmatrix} \qquad V^T V = \begin{bmatrix} 1.0000 & -0.0000 & 0.0000 & 0.0000 \\ -0.0000 & 1.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 & -0.0000 \\ 0.0000 & 0.0000 & -0.0000 & 1.0000 \end{bmatrix}$$

Die Nichtdiagonalelemente des Produkts $V^T V$ liegen in diesem Beispiel in der Größenordnung von 10^{-16}, entspricht damit *sogut wie* einer Einheitsmatrix.

c)

Im Folgenden wird auf die Umformungen von (1.19) und (1.20) auf (1.27) und (1.28) eingegangen.

Aus der Multiplikation von V mit P erhält man (1.19) und (1.20):

$$v'_{i,p} = c\, v_{i,p} - s\, v_{i,q} \tag{1}$$

$$v'_{i,q} = s\, v_{i,p} + c\, v_{i,q} \tag{2}$$

Die Gleichungen (1) und (4) ist laut Skript gleich der folgenden Ausdrücke, die auch in `doPpq` verwendet wurden.

$$v'_{i,p} = v_{i,p} - s\,(v_{i,q} + \tau\, v_{i,p}) \tag{3}$$
$$= (1 - s\,\tau)\, v_{i,p} - s\, v_{i,q} \tag{4}$$
$$v'_{i,q} = v_{i,q} + s\,(v_{i,p} - \tau\, v_{i,q}) \tag{5}$$
$$= (1 - s\,\tau)\, v_{i,q} + s\, v_{i,p} \tag{6}$$

Der Vergleich von (1) mit (4) und (2) mit (6) zeigt, dass lediglich zu zeigen bleibt, der Koeffizient $(1 - s\,\tau) = c$ ist.

$$\begin{aligned} 1 - s\,\tau &= \frac{1+c}{1+c} - \frac{s^2}{1+c} && \text{mit } \tau := \frac{s}{1+c} \\ &= \frac{1 - s^2 + c}{1+c} \\ &= \frac{c\,(c+1)}{c+1} && \text{mit } c^2 = 1 - s^2 \\ &= c && \textbf{qed.} \end{aligned}$$

Damit ist gezeigt, dass die Gleichungen äquivalent sind.

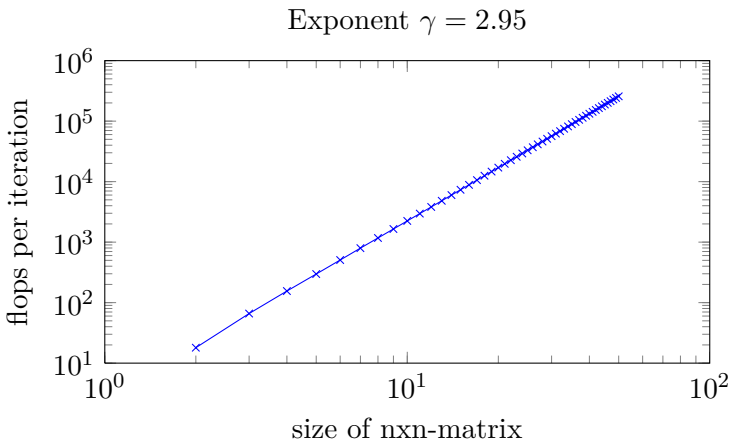
d)

Für eine Matrix der Größe N ist zu überprüfen wie viele Flops für eine komplette Iteration nötig sind. Im ersten Schritt wird die Abhängigkeit des Rechenaufwands analysiert um diese Abhängigkeit im zweiten Schritt durch einen Zähler im Programm zu überprüfen.

Eine Iteration besteht aus $N(N-1)$ Drehungen. Für eine Drehung werden zunnächst die Parameter s, c, t und τ bestimmt. Danach werden für 2 Spalten $(N-2)$ Werte nach Formel 2 bzw. 6 mit je 2 Multiplikationen berechnet.
Demnach verhält sich die die Anzahl der Flops proportional zu N^3.

$$\#\text{FLOPS} \simeq \left(\frac{N^2+N}{2}\right)_{\text{Drehungen}} [\, 2_{\text{Spalten}} \cdot 2\ (N-2) + const] \simeq 2\,N^3$$

Zur Bestimmung der benötigten Flops wurde ein Zähler, als globale Variable `flop`, in das Programm implementiert.
Für die Auswerung der Flops in Abhängigkeit der Matrixdimension, wird eine Schleife ausgeführt, bei der die Dimension von $n = 4$ bis auf $n = 50$ erhöht wird und jedes Mal die Multiplikationen gezählt werden. Im logarithmischen Plot kann ein konstanter Exponent verifiziert werden und der Exponent durch einen linearen Fit bestimmt werden.



Die Grafik wurde mit folgendem Skript erstellt:

N-Abhängigkeit

```
1   clear all; clc
2   global flop;
3   maxn = 50;
4   randomA = 50*rand(maxn);
5   randomA = randomA + randomA';
6
7   flo = zeros(maxn,1);
8   for n = 1: maxn
9       flop = 0;               % reset global flop counter
10      A = randomA(1:n,1:n); % get symmetric nxn-matrix
11      [D, V ,iterDone] = myEig(A);
12      flo(n) = flop./iterDone;
13  end
14
15  % plot flops per iteration over n
16  figure(4);
17  loglog(1:maxn,flo,'x-')
18  xlabel('size N of NxN-matrix ')
19  ylabel('flops per iteration')
20  %%
21  %model:  flops / Iteration = c * N^(gamma)
22  int = (10:maxn)';   % N-Intervall to calculate exponent gamma
23  p = polyfit(log(int),log(flo(int)),1);
24  gamma = p(1);
25  title(sprintf('Exponent $\\gamma = %.2f$',gamma))
```