Lösung zu Serie 4

1. Siehe Lösung Multiple-Choice.

2. Seien $\underline{A}, \underline{B} \in \mathbb{R}^{n,n}$, mit $\underline{A} = \{a_{ij}\}_{1 \le i,j \le n}$, $\underline{B} = \{b_{ij}\}_{1 \le i,j \le n}$ sodass $a_{ij} = b_{ij} = 0$ für $i > j$.

   Zu zeigen: Für $\underline{C} := \underline{A} \cdot \underline{B} \in \mathbb{R}^{n,n}$, $C = \{c_{ij}\}_{1 \le i,j \le n}$, gilt:
   $c_{ij} = 0$ für $i > j$.

   Beweis:
   Nach Definition von $\underline{C}$ bzw. Definition der Matrix-multiplikation gilt $\underline{A} \cdot \underline{B} =$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} & \cdots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & & \vdots \\ b_{n1} & b_{n2} & b_{n3} & \cdots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} & \cdots & c_{1n} \\ c_{21} & c_{22} & c_{23} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & & \vdots \\ c_{n1} & c_{n2} & c_{n3} & \cdots & c_{nn} \end{pmatrix}$$

   also z.B.

   - $c_{11} = \sum_{k=1}^{n} a_{1k} b_{k1}$;
   - $c_{12} = \sum_{k=1}^{n} a_{1k} b_{k2}$

   und im Allgemeinen:

   $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ für $1 \le i,j \le n$.

Für a_{ik} wissen wir: $a_{ik} = 0$ für $i > k$. (1)
Für b_{kj} wissen wir: $b_{kj} = 0$ für $k > j$ (2)

Wir wollen wissen, was passiert, falls $i \geq j$.
Es gibt 3 Fälle für $k = 1, \ldots, n$ zu unterscheiden:

- $i \geq j \geq k$: In diesem Fall gilt $i > k$, und somit gilt auch (1). Deshalb folgt in diesem Fall:
$$\underbrace{a_{ik}}_{=0} \cdot b_{kj} = 0$$

- $i > k > j$: In diesem Fall gilt $k > j$, und deshalb gilt auch (2). Wir erhalten folglich
$$a_{ik} \cdot \underbrace{b_{kj}}_{=0} = 0$$

- $k \geq i > j$: Hier gilt folglich auch $k > j$, und somit gilt $b_{kj} = 0$ wegen (2) und deshalb
$$a_{ik} \cdot \underbrace{b_{kj}}_{=0} = 0$$

Somit folgt, dass $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} = 0$ für $i \geq j$. □

3.(a) Da wir Einsicht in die MATLAB-Funktion

```
function w = transform(v),
```

haben, können wir Zeile für Zeile als Matrix interpretieren:

```
5 |     w = v(n:-1:1);
```

Als erstes wird der Vektor $\underline{v}$ "auf den Kopf gestellt", dies können wir wie folgt realisieren:

$$\underline{w}_1 = \begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 1 & 0 \\ \vdots & \vdots & \iddots & \vdots & \vdots \\ 0 & 1 & \cdots & 0 & 0 \\ 1 & 0 & \cdots & 0 & 0 \end{pmatrix} \underline{v} \qquad (3)$$

2.
```
w(n) = w(n) + dot(v,ones(n,1));
```

In einem zweiten Schritt wird zum letzten Eintrag von $\underline{w}_1$ $\langle \underline{v}, \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \rangle = \sum_{i=1}^{n} v_i$ hinzugezählt, das neue $\underline{w}_2$ erhalten wir also durch folgende Matrixmultiplikation:

$$\underline{w}_2 = \underline{w}_1 + \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \\ 1 & \cdots & 1 \end{pmatrix} \underline{v} \overset{(3)}{=} \begin{pmatrix} 0 & 0 & \cdots & \cdots & 0 & 1 \\ 0 & 0 & \cdots & \cdots & 1 & 0 \\ \vdots & \vdots & & & \vdots & \vdots \\ 0 & 1 & 0 & \cdots & \cdots & 0 \\ 2 & 1 & \cdots & \cdots & 1 & 1 \end{pmatrix} \underline{v} \qquad (4)$$

3.
```
7 |  w = w + v(1)*ones(n,1);
```

Als letztes addieren wir zu w_2 den Vektor $\begin{pmatrix} v_1 \\ \vdots \\ v_1 \end{pmatrix}$:

$$\underbrace{\underline{w}_3}_{=w} = \underline{w}_2 + \begin{pmatrix} v_1 \\ \vdots \\ v_1 \end{pmatrix} \overset{(4)}{=} \begin{pmatrix} 0 & 0 & \cdots & \cdots & 0 & 1 \\ 0 & 0 & \cdots & \cdots & 1 & 0 \\ \vdots & \vdots & & & \vdots & \vdots \\ 0 & 1 & 0 & \cdots & \cdots & 0 \\ 2 & 1 & 1 & \cdots & 1 & 1 \end{pmatrix} \underline{v} + \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ 1 & 0 & \cdots & \cdots & 0 \end{pmatrix} \underline{v}$$

$$= \underbrace{\begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 & 1 \\ \vdots & \vdots & & & \vdots & \vdots \\ \vdots & \vdots & & & 1 & 0 \\ 1 & 1 & 0 & \cdots & \cdots & 0 \\ 3 & 1 & 1 & \cdots & \cdots & 1 \end{pmatrix}}_{=: A} \underline{v}$$

So bekommen wir z.B für $\underline{v} \in \mathbb{R}^3$:

$$\underline{A} = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 3 & 1 & 1 \end{pmatrix}$$

oder für $\underline{v} \in \mathbb{R}^4$:

$$\underline{A} = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 3 & 1 & 1 & 1 \end{pmatrix} \quad \text{etc...}$$

Bemerkung: Natürlich kann auch hier die Strategie aus 3(b) unten angewendet werden!

3(b) Ziel der Aufgabe ist es, einen allgemein gültige Strategie zu finden, mit welcher wir die Matrix $\underline{A}$ berechnen können, wenn wir zwar wissen, wie die Ergebnisse $\underline{w}$, von $\underline{w} = \underline{A}\,\underline{v}$ (hier gegeben durch $\underline{w} = \text{topsecret}(\underline{v})$) für beliebige $\underline{v} \in \mathbb{R}^n$, $n \in \mathbb{N}$, aussehen, wir aber $\underline{A}$ nicht explizit kennen.

Idee: Wir nehmen die kanonische Basis des $\mathbb{R}^n$,

$$\underline{e}_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad \underline{e}_2 = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \ldots, \underline{e}_i = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \leftarrow i.\text{Eintrag}, \ldots, \underline{e}_n = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}.$$

und stellen fest, dass für $\underline{A} = \{a_{ij}\}_{1 \le i,j \le n}$

$$\underline{A}\,\underline{e}_i = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & \cdots & & a_{nn} \end{pmatrix} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \leftarrow i.\text{Eintrag} = \begin{pmatrix} a_{1i} \\ a_{2i} \\ \vdots \\ a_{ni} \end{pmatrix} \quad \text{für alle } i \in \{1, \ldots, n\}. \tag{5}$$

Somit erhalten wir $\underline{A}$ durch Berechnen von $A(:,i) = \text{topsecret}(\underline{e}_i)$ für alle $i = 1, \ldots n$.

Wir überprüfen unsere Behauptung mit der gegebenen "transform.m" MATLAB-Funktion, aus 3(a) indem wir eine MATLAB-Funktion

```
function A = findA_transform(n)
```

schreiben, welches uns A mit obiger

Strategie für ein $n \in \mathbb{N}$ berechnet:

```
1   function A = findA_transform(n)
2   % Eingabe:
3   % n  : natuerliche Zahl, gewuenschte Dimension von A
4   % Ausgabe: A, Matrix welche 'transform.m' repraesentiert
5       % Initialisierung
6       % Wir stellen fest, dass die Spalten der Einheitsmatrix die
7       % kanonische Basis bilden.
8       e = eye(n);
9       A = NaN(n);
10      for i=1:n
11          A(:,i) = transform(e(:,i));
12      end
13  end
```

Tatsächlich erhalten wir die Matrix, welche wir in 3(a) berechnet haben:

```
>> A = findA_transform(3)

A =

     1     0     1
     1     1     0
     3     1     1

>> A = findA_transform(4)

A =

     1     0     0     1
     1     0     1     0
     1     1     0     0
     3     1     1     1
```

Für das Berechnen der Matrix $\underline{A}$ für die MATLAB-Funktion topsecret.p, wandeln wir Zeile 11 im Code geeignet ab:

```
1   function A = findA_topsecret(n)
2   % Eingabe:
3   % n  : natuerliche Zahl, gewuenschte Dimension von A
4   % Ausgabe: A, Matrix welche 'topsecret.p' repraesentiert
5       % Initialisierung
6       % Wir stellen fest, dass die Spalten der Einheitsmatrix die
7       % kanonische Basis bilden.
8       e = eye(n);
9       A = NaN(n);
10      for i=1:n
11          A(:,i) = topsecret(e(:,i));
12      end
13  end
```

Und erhalten für A z.B im Falle n=3 und n=4:

```
>> A = findA_topsecret(3)

A =

     8     1     6
     3     5     7
     4     9     2

>> A = findA_topsecret(4)

A =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

4 (a)

```
function A = buildminij(n)
% Eingabe der Funktion: natuerliche Zahl n
% Rueckgabe der Funktion: reelle Matrix A der Dimensionen nxn
    % Initialisierung
    A = NaN(n);
    for i=1:n
        for j=1:n
            if i <= j
                A(i,j) = i;
            else A(i,j) = j;
            end
        end
    end
end
```

(b)

```
function A = buildminij_genius(n)
% Eingabe der Funktion: natuerliche Zahl n
% Rueckgabe der Funktion: reelle Matrix A der Dimensionen nxn
    % Betrachten wir die Struktur von A genauer, stellen wir fest,
    % dass im oberen Dreieck die Eintraege in den Zeilen konstant ist,
    % und zwar immer die Zeilennummer angibt.
    % Dasselbe gilt im Unteren Dreieck fuer die Spalten.
    % Wir konstruieren deshalb eine Matrix T, welche ueberall den
    % Zeilenindex als Eintrag hat:
    T = (1:n).' * ones(1,n);
    % A erhalten wir somit, indem wir die obere Dreiecksmatrix von
    % T nehmen, dann die untere Dreiecksmatrix der transponierten
    % Matrix T, welche die Spaltenindizes repraesentiert.
    % Da wir nun die Diagonale doppelt gezaehlt haben, muessen wir
    % sie einmal wieder abziehen.
    A = triu(T) + tril(T') - diag(1:n);
end
```

5. (a) Für $\underline{a}, \underline{b} \in \mathbb{R}^n$ gilt

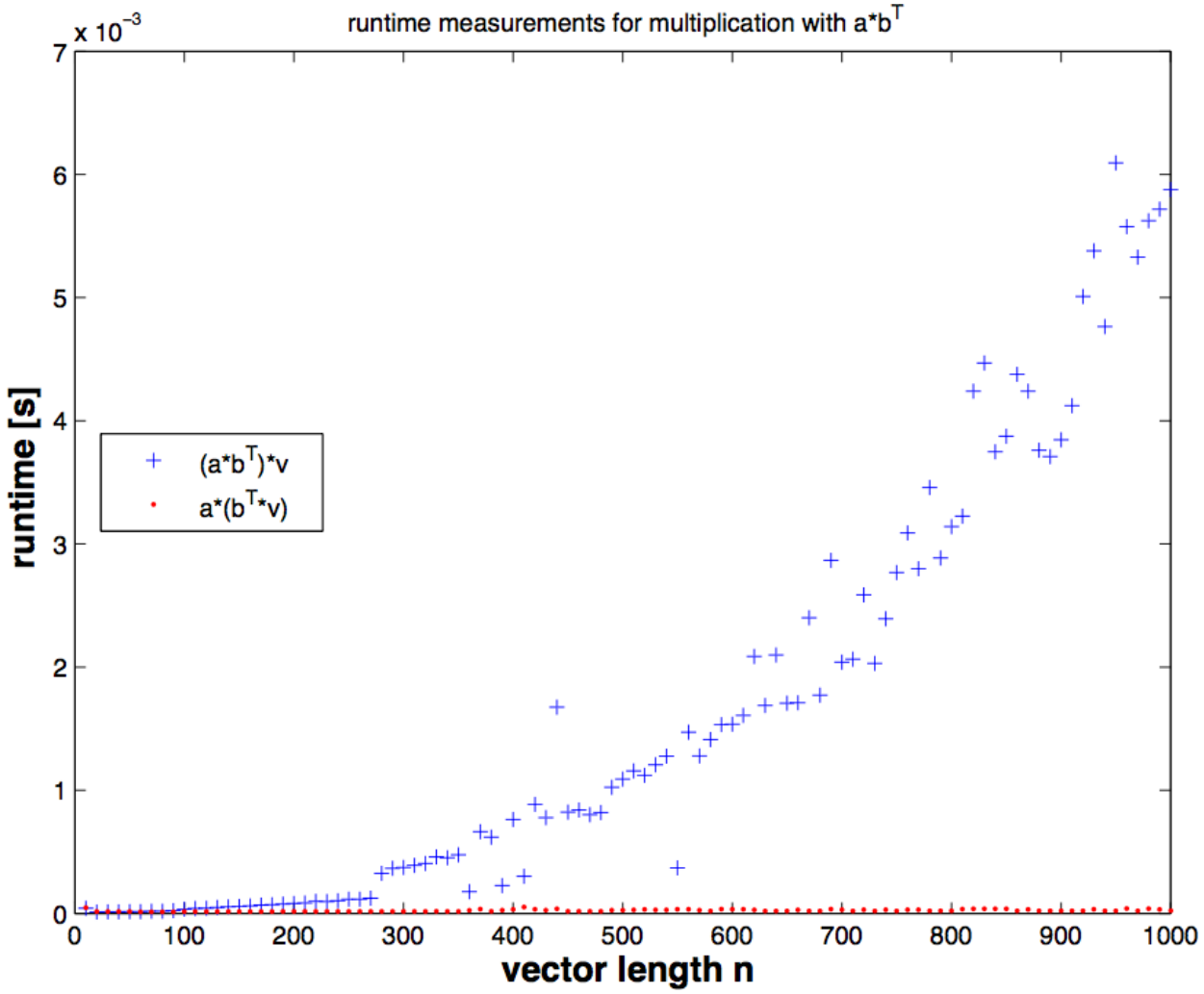
$$\underline{A} = \underline{a}\,\underline{b}^T$$

$$= \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \begin{pmatrix} b_1 & \cdots & b_n \end{pmatrix} = \begin{pmatrix} a_1 b_1 & \cdots & a_1 b_n \\ \vdots & \ddots & \vdots \\ a_n b_1 & \cdots & a_n b_n \end{pmatrix} = \{a_i b_j\}_{1 \le i,j \le n}$$

(b) Wir ergänzen das MATLAB-Template rankonemulttiming.m:

```
function rankonemulttiming
% Measurement of runtimes for different ways of multiplying a vector with a
% matrix of the form $\Va\cdot\Vb^T$, where $\Va$ and $\Vb$ are column
% vectors, a so-called tensor product matrix.
N = 1000;                       % Maximal size of vectors
a = rand(N,1); b = rand(N,1); % Initialize random column vectors of length $N$
v = rand(N,1);

res = []; % matrix for recording times
% conduct timings for vectors of different size
for n=10:10:N
    t1 = realmax;
    for j=1:3
        tic;
        w1 = rankonemultslow(a(1:n),b(1:n),v(1:n));
        t1 = min(toc,t1);
    end
    t2 = realmax;
    for j=1:3
        tic;
        w2 = rankonemultfast(a(1:n),b(1:n),v(1:n));
        t2 = min(toc,t2);
    end
    norm(w1-w2), % Check for agreement of results
    res = [res; n, t1, t2];
end

% Create plots of runtimes.
figure;
plot(res(:,1),res(:,2),'b+',res(:,1),res(:,3),'r.');
xlabel('{\bf vector length n}','fontsize',14);
ylabel('{\bf runtime [s]}','fontsize',14);
title('runtime measurements for multiplication with a*b^T');
legend('(a*b^T)*v','a*(b^T*v)','location','best');

print -depsc2 '../../LANMFigures/rankonemulttiming.eps';
end

% The following two functions perform algebraically equivalent operations,
% however at vastly different computationl cost. Note the different order of
% multiplications implied by the positions of the brackets.

function  w = rankonemultslow(a,b,v), w = (a*b')*v; end
function  w = rankonemultfast(a,b,v), w = a*(b'*v); end
```

Die Ausgabe sieht folgendermassen aus:



(c) • Bei der MATLAB-Funktion abmult1.m berechnen wir als erstes die Matrix $\underline{A} = \underline{a} \cdot \underline{b}^T$, was uns einen Aufwand von $\mathcal{O}(n^2)$ kostet.
Danach berechnen wir $\underline{A}\,\underline{v}$, was eine Matrix-Vektor-Multiplikation darstellt und somit Aufwand $\mathcal{O}(n^2)$ kostet.
Insgesamt bekommen wir also:

$$\mathcal{O}(n^2) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$$ Aufwand für abmult1.m

• Bei abmult2.m berechnen wir zuerst $u := \underline{b}^T \cdot \underline{v}$, was einer Vektor-Vektor-Multiplikation entspricht und

somit Aufwand $O(n)$ hat.
Danach führen wir eine skalare Vektormultiplikation durch, nämlich $\underline{a} \cdot u$, wieder wenden wir dabei einen Aufwand von $O(n)$ auf.
Total bekommen wir:
$O(n) + O(n) = O(n)$ für abmult2.m

Somit ist die Variante abmult2.m deutlich kostengünstiger, wie auch der Laufzeitplot in 5(b) verrät.