LINEARE ALGEBRA ROUTINEN

Julia Becker
04.04.2012
Bonn

Inhaltsverzeichnis

- numerische Berechnungen mit Matrizen basieren auf der Analyse der Matrizen-Multiplikation
  - es werden verschiedene Herangehensweisen und Darstellungen vorgestellt
- algebraische Operationen können durch die lineare Algebra sowie numerisch dargestellt werden

- Algorithmen werden in einer stilisierten Form von MATLAB dargestellt, sowie Begriffe der Software LAPACK benutzt
- vorgestellte Algorithmen sollten als zusammenfassende Darstellung der Idee aufgefasst werden und kritisch angewendet werden

Matrizen

- **Matrix-Notation**

$$A \epsilon \mathbb{R}^{mxn} \Leftrightarrow A = (a_{ij}) = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \; a_{ij} \epsilon \mathbb{R}$$

- **Matrix-Operationen**
  - $C = A^T \Longrightarrow c_{ij} = a_{ji} \quad (\mathbb{R}^{mxn} \longrightarrow \mathbb{R}^{nxm})$
  - $C = A + B \Longrightarrow c_{ij} = a_{ij} + b_{ij} \quad (\mathbb{R}^{mxn} + \mathbb{R}^{mxn} \longrightarrow \mathbb{R}^{mxn})$
  - $C = \alpha A \Longrightarrow c_{ij} = \alpha a_{ij} \quad (\mathbb{R} x \mathbb{R}^{mxn} \longrightarrow \mathbb{R}^{mxn})$
  - $C = AB \Rightarrow c_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj} \quad (\mathbb{R}^{mxp} x \mathbb{R}^{pxn} \longrightarrow \mathbb{R}^{mxn})$

Matrizen

- **Zerlegung der Matrix in Zeilen und Spalten**

$$A \epsilon \mathbb{R}^{mxn} \Leftrightarrow A = \begin{bmatrix} r_1{}^T \\ \vdots \\ r_m{}^T \end{bmatrix}, r_k \epsilon \mathbb{R}^n$$

$$A \epsilon \mathbb{R}^{mxn} \Leftrightarrow A = [c_1, \dots, c_n], c_k \epsilon \mathbb{R}^m$$

- **Doppelpunkt-Notation**
  - bezeichnet k-te Zeile: $\mathrm{A} \epsilon \mathbb{R}^{mxn},\ A(k, :) = [a_{k1}, \dots, a_{kn}]$
  - bezeichnet k-te Spalte: $A \epsilon \mathbb{R}^{mxn},\ A(:, k) = \begin{bmatrix} a_{1k} \\ \vdots \\ a_{mk} \end{bmatrix}$

Vektoren

- **Vektor-Notation**

$$x \epsilon \mathbb{R}^{nx1} \Leftrightarrow x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \; x_i \epsilon \mathbb{R}$$

$$x \epsilon \mathbb{R}^{1xn} \Leftrightarrow x = (x_1, \ldots, x_n) \; x_i \epsilon \mathbb{R}$$

- **Vektor-Operationen**

  $a \epsilon \mathbb{R}, x \epsilon \mathbb{R}^n, y \epsilon \mathbb{R}^n$

  - $z = \alpha x \Rightarrow z_i = \alpha x_i$
  - $z = x + y \Rightarrow z_i = x_i + y_i$
  - $c = x^T y \Rightarrow c = \sum_{i=1}^{n} x_i y_i$

Vektoren

- Vektor-Operationen
  - $z = x * y \Rightarrow z_i = x_i y_i$
- Saxpy
  - $y = \alpha x + y \Longrightarrow y_i = \alpha x_i + y_i$

    → „scalar a x plus y“

Vektoren und Matrizen

- **Gaxpy, generalized saxpy**

$A \epsilon \mathbb{R}^{mxn}, x \epsilon \mathbb{R}^{n}, y \epsilon \mathbb{R}^{m}$

$$y = Ax + y \;\; \Rightarrow y_i = \sum_{j=1}^{n} a_{ij} x_j + y_i \quad , i = 1{:}\, m$$

Skalarprodukt und Saxpy

- **Algorithmus 1.1.1.** (Skalarprodukt, $\mathrm{x}, \mathrm{y} \epsilon \mathbb{R}^n, c = x^T y$)
  - for i = 1:n

    c = c + x(i)y(i)

    end

- **Algorithmus 1.1.2.** (Saxpy; $\mathrm{x}, \mathrm{y} \epsilon \mathbb{R}^n; a \epsilon \mathbb{R}, \mathrm{y} = \mathrm{ax} + \mathrm{y}$)
  - for i = 1:n

    y(i) = ax(i) + y(i)

    end

Matrix-Vektor Multiplikation und Gaxpy

- **Algorithmus 1.1.3.**

  (**<u>Gaxpy</u>: Zeilenversion,** $A \epsilon \mathbb{R}^{mxn}, x \epsilon \mathbb{R}^{n}, y \epsilon \mathbb{R}^{m}, y = Ax + y$)

```
for i = 1:m
      for j = 1:n
                y(i) = A(i,j)x(j) + y(i)
      end
end
```

  - **innere Schleife stellt eine Saxpy Operation dar**

Matrix-Vektor Multiplikation und Gaxpy

- **Algorithmus 1.1.4.**
  (<u>Gaxpy</u>: Spaltenversion, $A \epsilon \mathbb{R}^{mxn}, x \epsilon \mathbb{R}^{n}, y \epsilon \mathbb{R}^{m}, y = Ax + y$**)**

```
for j = 1:n
    for i = 1:m
        y(i) = A(i,j)x(j) + y(i)
    end
end
```

  - **innere Schleife stellt eine Saxpy Operation dar**

Matrix-Vektor Multiplikation und Gaxpy

- **A als eine Sammlung von Reihen, Algorithmus 1.1.3 :**
  - for i = 1:m
    $y(i) = r_i^T x + y(i)$
    end
- **A als eine Ansammlung von Spalten, Algorithmus 1.1.4. :**
  - for j = 1:n
    $y = x_j c_j + y$
    end
  - **y erscheint hier als fortlaufende Vektorsumme, die immer wieder durch Saxpy Updates aktualisiert wird**

Matrix-Vektor Multiplikation und Gaxpy

- **Algorithmus 1.1.3. bzw. 1.1.4. in <u>Doppelpunkt-Notation</u>**
  - for i = 1:m
    y(i) = A(i,:)x + y(i)
    end

  - for j = 1:n
    y = x(j)A(:,j) + y
    end
    - **diese Darstellungsart entfernt sich weiter von der linearen Algebra und fokussiert eher numerische Themen**

Matrix-Vektor Multiplikation und Gaxpy

- als weitere Anwendung der Doppelpunkt-Notation, betrachten wir nun das <u>äußere Produkt bzw. Vektorprodukt</u>

  $A \epsilon \mathbb{R}^{mxn}, x \epsilon \mathbb{R}^{mx1}, y \epsilon \mathbb{R}^{1xn}$

  $$A = A + xy^T$$

- xy^T ist ein spezielles Matrix-Matrix-Produkt, dessen Einträge sich wie folgt zusammensetzen:

$$a_{ij} = a_{ij} + x_i y_j \ mit \ i = 1{:}\, m \ und \ j = 1{:}\, n$$

Matrix-Vektor Multiplikation und Gaxpy

- **daraus ergeben sich die folgenden zwei Algorithmen:**
  - for i = 1:m

    $A(i,:) = A(i,:) + x(i)y^T$

    end
    - **ein Vielfaches von y^T wird zu jeder Zeile von A addiert**
  - for j = 1:n

    $A(:,j) = A(:,j) + y(i)x$

    end
    - **ein Vielfaches von y^T wird zu jeder Spalte von A addiert**

Matrizenmultiplikation

- nach dem Vorangegangenen gibt es nun 3 verschiedene Herangehensweisen an die Matrizenmultiplikation
  - jeder Eintrag als **Skalarprodukt** interpretiert
    - $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1*5+2*7 & 1*6+2*8 \\ 3*5+4*7 & 3*6+4*8 \end{bmatrix}$
  - Linearkombinationen der Spalten: **Saxpy**
    - $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \left[5 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 7 \begin{bmatrix} 2 \\ 4 \end{bmatrix}, 6 \begin{bmatrix} 1 \\ 3 \end{bmatrix} + 8 \begin{bmatrix} 2 \\ 4 \end{bmatrix} \right]$
  - Summe von **Vektorprodukten**
    - $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \begin{bmatrix} 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 \\ 4 \end{bmatrix} \begin{bmatrix} 7 & 8 \end{bmatrix}$
- numerisch gesehen gibt es einige Unterschiede zwischen den Verfahren z.B. bezüglich des Zugriffs auf Daten bzw. des Speicherplatzbedarfs

Matrizenmultiplikation

- Matrizenmultiplikation: $A \epsilon \mathbb{R}^{mxp}, B \epsilon \mathbb{R}^{pxn}, C \epsilon \mathbb{R}^{mxn}$ $\mathrm{C} = \mathrm{AB} + \mathrm{C}$
- gewöhnlich Interpretation als Array von Skalarprodukten, wobei Einträge nacheinander von rechts nach links, von oben nach unten berechnet werden
- dafür wird klassisch ein triple loop benötigt, diese können auf 3!=6 Varianten angeordnet werden

Loop Order	Inner Loop	Middle Loop	Inner Loop Data Access
ijk	Skalarprodukt	Vektor x Matrix	A Zeile, B Spalte
jik	Skalarprodukt	Matrix x Vektor	A Zeile, B Spalte
ikj	Saxpy	Gaxpy Zeile	B Zeile, C Zeile
jki	Saxpy	Gaxpy Spalte	A Spalte, C Spalte
kij	Saxpy	Vektorprodukt Zeile	B Zeile, C Zeile
kji	Saxpy	Vektorprodukt Spalte	A Spalte, C Spalte

Matrizenmultiplikation

- **Algorithmus 1.1.5.**

(Matrixmultiplikation: ijk-Variante)

```
for i = 1:m
    for j = 1:n
        for k = 1:p
            C(i,j) = A(i,k)B(k,j) + C(i,j)
        end
    end
end
```

- innere Schleife: Skalarprodukt
- mittlere Schleife: Matrix x Vektor
- Datenzugriff: A als Zeile, B als Spalte

Matrizenmultiplikation

- **Algorithmus 1.1.5.**

  (Matrixmultiplikation: jki-Variante)

```
for j = 1:n
    for k = 1:p
        for i = 1:m
            C(i,j) = A(i,k)B(k,j) + C(i,j)
        end
    end
end
```

  - **innere Schleife: Saxpy**
  - **mittlere Schleife: Gaxpy Spalte**
  - **Datenzugriff: A als Spalte, C als Spalte**

Matrizenmultiplikation

- jede Variante benötigt den gleichen Aufwand an Gleitkommaarithmetik, greift jedoch auf die Daten von A, B und C unterschiedlich zu

Loop Order	Inner Loop	Middle Loop	Inner Loop Data Access
ijk	Skalarprodukt	Vektor x Matrix	A Zeile, B Spalte
jik	Skalarprodukt	Matrix x Vektor	A Zeile, B Spalte
ikj	Saxpy	Gaxpy Zeile	B Zeile, C Zeile
jki	Saxpy	Gaxpy Spalte	A Spalte, C Spalte
kij	Saxpy	Vektorprodukt Zeile	B Zeile, C Zeile
kji	Saxpy	Vektorprodukt Spalte	A Spalte, C Spalte

Matrizenmultiplikation

- Skalarproduktversion, verschiedene Interpretationen
- **Algorithmus 1.1.6**

  - for i = 1:m
        for j = 1:n
            C(i,j) = A(i,:)B(:,j) + C(i,j)
        end
    end
  - for i = 1:m
        for j = 1:n
            $C_{ij} = a_i^T b_j + c_j$
        end
    end

Matrizenmultiplikation

- **Algorithmus 1.1.6**
  - for i = 1:m

    $c_i^T = a_i^T B + c_i^T$

    end
  - for i = 1:m

    C(i,:) = A(i,:)B + C(i,:)

    end

Matrizenmultiplikation

- Saxpy Darstellung, A und C sind in Spalten aufgeteilt
  - $C = AB + C \Rightarrow c_j = \sum_{k=1}^{p} b_{kj} a_k + c_j \, , j = 1{:}n$
  - dies kann mit einer Abfolge von Saxpy Updates zusammengefasst werden

- **Algorithmus 1.1.7**
  - for j = 1:n

```
for j = 1:n
    for k = 1:p
        C(:,j) = A(:,k)B(k,j) + C(:,j)
    end
end
```

```
for j = 1:n
    C(:,j) = AB(:,j) + C(:,j)
end
```

Matrizenmultiplikation

- Vektorprodukt Darstellung
  - A wird als Ansammlung von Spalten und B als Sammlung von Zeilen betrachtet
  - interpretiert man AB als Summe von p äußeren Produkten ergibt sich:

- **Algorithmus 1.1.8.**
  - for k = 1:p

```
for k = 1:p
    C = A(:,k)B(k,:) + C
end
```

Matrizenmultiplikation

- Komplexe Matrizen
  - Vektor/Matrix Addition und Multiplikation sind im Komplexen identisch zum Realen
  - transponieren wird zu konjugiert transponieren
    - $C = A^H \implies c_{ij} = \bar{a}_{ji}\ , C \epsilon \mathbb{C}^{mxn}$
  - Skalarprodukt
    - $s = x^H y = \sum_{i=1}^{n} \bar{x}_i\, y_i$
  - $A = B + iC\ \epsilon \mathbb{C}^{mxn}$, dann bezeichnet Re(A)=B den Realteil von A, sowie Im(A)=C den Imaginärteil von A

Matrizenmultiplikation

- „Level"
  - Level-1 Operation:
    - Komplexität linear, d.h. arithmetischer Aufwand, sowie Datenmenge linear
    - Beispiele: Skalarprodukt, Saxpy $\longrightarrow je\ \mathcal{O}(n)$
  - Level-2 Operation:
    - Komplexität quadratisch, Arbeitsaufwand sowie Datenmenge quadratisch
    - Beispiele: äußeres Produkt, Gaxpy $\longrightarrow je\ \mathcal{O}(mn)$

Matrizenmultiplikation

- „Level"
  - Level-3 Operation:
    - Speicherplatz für die Daten ist quadratisch, Arbeitsaufwand jedoch kubisch
    - Beispiel: Matrixupdate (C=AB+C) $\longrightarrow f\ddot{u}r\ nxn\ Matrizen$: $\mathcal{O}(n^2)\ Matrixeintr\ddot{a}ge, \mathcal{O}(n^3)\ arithmetische\ Operationen$

Matrixstrukturen ausnutzen

- Effizienz bzw. Leistungsfähigkeit eines Algorithmus hängt von vielen Dingen ab
  - die zwei wichtigsten Punkte sind jedoch:
    - Menge an arithmetischen Operationen
    - Speicherplatzbedarf

- als Beispiele zur Effizienzsteigerung dieser zwei Argumente werden Bandmatrizen sowie symmetrische Matrizen betrachtet

Flops

- floating point operations per second
  - Anzahl der Gleitkommaoperationen, die von Rechnersystemen oder Prozessoren pro Sekunde ausgeführt werden können
  - dienen hier als Maß für die Effizienz eines Algorithmus

- aber: Flops sind nur eine kurze Bilanz, die nur einen Aspekt der Effizienz aufgreifen

Flops

- Anzahl der Flops berechnet man durch Aufsummieren der arithmetischen Operationen zusammen mit dem meist verschachtelten Befehl im Algorithmus
  - Beispiel: Matrixmultiplikation
    - Befehl: $C(i,j) = A(i,k)B(k,j) + C(i,j)$
    - beinhaltet 2 Flops, die mnp mal ausgeführt werden $\longrightarrow$ normale Matrixmultiplikation benötigt 2mnp Flops

Bandmatrizen

- gehören zu den dünnbesetzen Matrizen: neben Hauptdiagonale nur bestimmte Anzahl an Nebendiagonalen ungleich Null
  - $A \epsilon \mathbb{R}^{mxn}$, dann bezeichnet man mit der Bandbreite folgendes:
    - p ist untere Bandbreite, wenn $a_{ij} = 0 \; für \; i > j + p$
    - q ist obere Bandbreite, wenn $a_{ij} = 0 \; für \; j > i + p$

- Bandstrukturen von häufig auftretenden mxn Matrizen

Matrixart	Untere Bandb.	Obere Bandb.
Diagonalm.	0	0
obere Dreiecksm.	0	n-1
obere Bidiagonalm.	0	1
untere Hessenbergm.	m-1	1

Bandmatrizen

- **x-0 Notation:** $\mathrm{A} = \begin{bmatrix} x & x & 0 \\ 0 & x & x \\ 0 & 0 & x \end{bmatrix}, \mathrm{x} \epsilon \mathbb{R}, A \epsilon \mathbb{R}^{3x3}$

- **Diagonalmatrizen multiplizieren**
  - $D \epsilon \mathbb{R}^{mxn},\ dann\ D = diag(d_1, \dots, d_q),\ q = \min(m, n) \Leftrightarrow d_i = d_{ii}$
  - $D \epsilon \mathbb{R}^{mxn}\ diagonal, A \epsilon \mathbb{R}^{mxn}$, dann ist DA ein row scaling von A $und\ AD\ ein\ column\ scaling\ von\ A$

Bandmatrizen

- **Dreiecksmatrizen multiplizieren**
  - **Beispiel:** $A, B\ Dreiecksmatrizen\ \epsilon \mathbb{R}^{3x3}, dann\ ist\ C = AB$
  - $C = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{11} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ 0 & a_{22}b_{22} & a_{22}b_{23} + a_{23}b_{33} \\ 0 & 0 & a_{33}b_{33} \end{bmatrix}$
  - **es ist** $a_{ik}b_{kj} = 0, wenn\ k < i\ oder\ j < k$, **damit erhalten wir:**

$$c_{ij} = \sum_{k=i}^{j} a_{ik}b_{kj}$$

Bandmatrizen

- **Algorithmus 1.2.1.**
  (Dreicksmatrixmultiplikation, $A, B \in \mathbb{R}^{nxn}, C = AB$**)**

  - C=0

```
C=0
  for i=1:n
     for j=i:n
               for k=i:j
                         C(i,j) = A(i,k)B(k,j) + C(i,j)
               end
     end
end
```

  - dieser Algorithmus benötigt nur $^1/_6$ des Arbeitsaufwandes einer normalen Matrizenmultiplikation

Bandmatrizen

- erweitert man die Doppelpunktnotation, lässt sich der Algorithmus weiter verkürzen
  - $A \epsilon \mathbb{R}^{mxn}, \quad p, q, r \epsilon \mathbb{Z}$ mit $1 \leq p \leq q \leq n \ und \ 1 \leq r \leq m$
  - $A(r, p: q) = [a_{rp} \quad \dots \quad a_{rq}] \epsilon \mathbb{R}^{1x(q-p+1)}$
  - $A \epsilon \mathbb{R}^{mxn}, \quad p, q, c \epsilon \mathbb{Z}$ mit $1 \leq p \leq q \leq m \ und \ 1 \leq c \leq n$
  - $A(p: q, c) = \begin{bmatrix} a_{pc} \\ \vdots \\ a_{qc} \end{bmatrix} \epsilon \mathbb{R}^{(q-p+1)}$

Bandmatrizen

- **Algorithmus 1.2.1.** (verkürzt)
  - ```
    C(1:n,1:n) = 0
      for i = 1:n
        for j = 1:n
              C(i,j) = A(i,i:j)B(i:j,j) + C(i,j)
        end
    end
    ```
    ```

Bandmatrizen

- **Bandspeicherung**
  - $A \epsilon \mathbb{R}^{nxn}$ mit unterer Bandbreite p und oberer Bandbreite q, mit p,q≪n
  - solche Matrizen können in einem $(p+q+1)xn$-Array gespeichert werden
    - $a_{ij} = A.band(i-j+q+1, j)$
  - Beispiel:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & a_{54} & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix} \quad A = \begin{bmatrix} 0 & 0 & a_{13} & a_{24} & a_{35} & a_{46} \\ 0 & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & 0 \end{bmatrix}$$

Bandmatrizen

- **Algorithmus 1.2.2.**

  (**Band Gaxpy**, $A \epsilon \mathbb{R}^{nxn}$ *mit unterer Bandbreite p und oberer Bandbreite q, gespeichert als A. band,* x,y$\epsilon\mathbb{R}^n$, $y = Ax + y$)

  - for j = 1:n

    $y_{top} = max(1, j-q)$

    $y_{bot} = min(n, j+p)$

    $a_{top} = max(1, q+2-j)$

    $a_{bot} = a_{top} + y_{bot} - y_{top}$

    $y(y_{top}:y_{bot}) = x(j)A.band(a_{top}:a_{bot}, j) + y(y_{top}:y_{bot})$

  end

  - **benötigt 2n(p+q+1) Flops**

Symmetrische Matrizen

- eine Matrix ist symmetrisch, falls $A^T = A, A \epsilon \mathbb{R}^{nxn}$
  - Speicherbedarf kann halbiert werden, indem nur das untere Dreieck gespeichert wird
    - $a_{ij} = A.vec\left((j-1)n - \frac{j(j-1)}{2} + i\right) \qquad (i \geq j)$
    - Beispiel: $A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix} \rightarrow A.vec = [1 \;\; 2 \;\; 3 \;\; 4 \;\; 5 \;\; 6\;]$

Symmetrische Matrizen

- **Algorithmus 1.2.3.**
  (**Symmetric Storage Gaxpy,** $A \epsilon \mathbb{R}^{nxn}$ $symmetrisch\ und\ als\ A.vec$ $gespeichert,\ x, y \epsilon \mathbb{R}^n,\ y = Ax + y$)
  - for j = 1:n

```
    for i = 1:j–1
        y(i) = A.vec((i–1)n–i(i–1)/2+j)x(j) + y(i)
    end
    for i = j:n
        y(i) = A.vec((j–1)n–j(j–1)/2+i)x(j) + y(i)
    end
end
```

  - **benötigt $2n^2$ Flops**

Symmetrische Matrizen

- **symmetrische Matrizen können auch diagonal gespeichert werden**
  - $i \geq j, \quad a_{i+k,i} = A.diag\left(i + nk - \frac{k(k-1)}{2}\right), (k \geq 0)$
  - $A \epsilon \mathbb{R}^{mxn}, \; D(A,k) \epsilon \mathbb{R}^{mxn}, \; bezeichnet\ die\ k-te\ Diagonale$ $von\ A$:
  - $[D(A,k)]_{ij} = \begin{cases} a_{ij} & j = i + k, 1 \leq i \leq m, 1 \leq j \leq n \\ 0 & sonst \end{cases}$
- **Beispiel:**
  - $A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}, \; A.diag = [1 \;\; 4 \;\; 6 \;\; 2 \;\; 5 \;\; 3]$

Symmetrische Matrizen

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 6 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 5 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

- beim A.diag Schema werden die Diagonalen aufeinanderfolgend gespeichert
- daraus ergibt sich für das Gaxpy y = Ax + y
  - $y = D(A,0)x + \sum_{k=1}^{n-1} (D(A,k) + D(A,k)^T)x + y$

Symmetrische Matrizen

- **Algorithmus 1.2.4.**
(Store-By-Diagonal Gaxpy, $A \epsilon \mathbb{R}^{nxn}, symmetrisch, als\ A.diag\ gespeichert, x, y \epsilon \mathbb{R}^n, y = Ax + y$**)**

```
for i = 1:n
    y(i) = A.diag(i)x(i) + y(i)
end
for k = 1:n−1
    t = nk−k(k−1)/2
    {y = D(A,k)x + y}
    for i = 1:n−k
        y(i) = A.diag(i+t)x(i+k) + y(i)
    end
    {y= D(A,k)^Tx + y}
    for i = 1:n−k
        y(i+k) = A.diag(i+t)x(i) + y(i+k)
    end
end
```

Anmerkung zum Arbeitsspeicher

- andere Möglichkeit den Arbeitsspeicheraufwand bei numerischen Berechnungen von Matrizen zu verringern ist, die zu überschreibenden Input-Daten zu kontrollieren
  - betrachten Matrixmultiplikation C = AB unter Bedingung, dass Eingabematrix B mit Ausgabematrix C überschrieben werden soll
  - **Algorithmus 1.1.7.** kann jedoch <u>nicht</u> folgendermaßen modifiziert werden:
  - for j = 1:n

```
for j = 1:n
    for k = 1:n
        B(:,j) = B(:,j) + A(:,k)B(k,j)
    end
end
```

Symmetrische Matrizen

- **jedoch reicht ein linearer Arbeitsspeicher, der die j-te Spalte des Produktes sichert, solange sie mit B(:,j) überschrieben werden kann:**
  - ```
    for j = 1:n
        w(1:n) = 0
        for k = 1:n
                w(:) = w(:) + A(:,k)B(k,j)
        end
        B(:,j) = w(:)
    end
    ```
    ```

Vielen Dank für Eure Aufmerksamkeit!