Lineare Algebra und Numerische Mathematik

Prof. Ralf Hiptmair

Seminar for Applied Mathematics, ETH Zürich

Vorlesung für D-BAUG Herbstsemester 2014

www.math.ethz.ch/education/bachelor/lectures/hs2014/other/linalgnum_BAUG
www.sam.math.ethz.ch/~hiptmair/tmp/LANM/

V. Numerische lineare Algebra mit MAT(rix) LAB(oratory)

In MATLAB: Alles ist matrix

5.1. MATLAB Grundlagen → Informatik I

Gram-Schmidt Orthonormalisierungsalgorithmus:

Gegeben: Endliche Menge von Vektoren $\{\mathbf{a}^1, \dots, \mathbf{a}^k\} \subset \mathbb{R}^n \setminus \{\mathbf{0}\}$, $k \leqslant n$.

1: $\mathbf{q}^1 := \frac{\mathbf{a}^1}{\|\mathbf{a}^1\|}$ % Erster der orthonormalen Vektoren
2: **for** $j = 2, \dots, k$ **do**
{ % Orthogonale Projektion auf das Erzeugnis der bisher berechneten Vektoren
3: $\mathbf{q}^j := \mathbf{a}^j$
4: **for** $\ell = 1, 2, \dots, j-1$ **do**
5: { $\mathbf{q}^j \leftarrow \mathbf{q}^j - \langle \mathbf{a}^j, \mathbf{q}^\ell \rangle \mathbf{q}^\ell$ }
6: **if** $(\mathbf{q}^j = \mathbf{0})$ **then** Abbruch
7: **else** { $\mathbf{q}^j \leftarrow \frac{\mathbf{q}^j}{\|\mathbf{q}^j\|}$ }
}

transponieren

```
q = A(:,j) - Q*(Q'*A(:,j)); % Orthogonal projection;
```

$$a^j - [q^1, \dots, q^{j-1}] \left(\begin{bmatrix} (q^1)^T \\ \vdots \\ (q^{j-1})^T \end{bmatrix} a^j \right)$$

$$= a^j - \sum_{\ell=1}^{j-1} q^\ell \langle q^\ell, a^j \rangle$$

Code 5.1.3: (gramschmidt.m) Gram-Schmidt-Orthogonalisierung, siehe Unterabschnitt 4.3.4

```
1  function Q = gramschmidt(A)
2  % Gram-Schmidt orthogonalization of column vectors
3  % Arguments: Matrix A passes vectors in its columns
4  % Return values: Matrix Q contains the orthornormal basis in its columns
5  [n,k] = size(A); % Get number k of vectors and dimension n of space
6  Q = A(:,1)/norm(A(:,1)); % First basis vector
7  for j=2:k
8    q = A(:,j) - Q*(Q'*A(:,j)); % Orthogonal projection; loop-free
     implementation
9    nq = norm(q);                % Check premature termination
10   if (nq < (1E-9)*norm(A(:,j))), break; end % Safe check for == 0
11   Q = [Q,q/nq]; % Add new basis vector as another column of Q
12 end
```

↳ Hinzufügen einer Spalte

5.2. Rundungsfehler

Gram-Schmidt in MATLAB

[Q,R] = **qr**(A) (volle QR-Zerlegung, $\mathbf{Q} \in \mathbb{R}^{m,m}$)

[Q,R] = **qr**(A,0) („sparsame" QR-Zerlegung, $\mathbf{Q} \in \mathbb{R}^{m,n}$, $m \geq n$)

```
1  % MATLAB script demonstrating the effect of roundoff on the result of
   Gram-Schmidt orthogonalization
2  format short; % Print only a few digits in outputs
3  % Create special matrix the so-called Hilbert matrix: (A)_{i,j} = (i+j-1)^{-1}
4  A = hilb(10), pause;          % 10x10 Hilbert matrix
5  Q = gramschmidt(A); % Gram-Schmidt orthogonalization of columns of A
6  % Test orthonormality of column of Q, which should be an orthogonal matrix
   according to theory
7  I = Q'*Q, pause; % Should be the unit matrix, but isn't !
8
9  % MATLAB's internal Gram-Schmidt orthogonalization
10 [Q1,R1] = qr(A), pause;
11 D = A - Q1*R1,  pause; % Check whether we get the expected result
12 I1 = Q1'*Q1, pause;   % Test orthonormality
```

Z 7 : $I = Q^T Q$ → Einheitsmatrix falls kein Abbruch

Z 11 : $D = 0$

Z 12 : $I1 \hat{=}$ Einheitsmatrix

Grund für $I \neq$ Einheitsmatrix in der Rechnung:

Rundungsfehler

> Computer rechnen intern nicht in $\mathbb{R}$, sondern mit endlich vielen Maschinenzahlen
>
> ▶ Unvermeidlich: Rundungsfehler bei '+', '-', '*', '/' und Funktionen

> Gefahr: Verstärkung der kleinen relativen Fehler (Grössenordnung 10^{-16}) bei '+', '-', '*', '/' !

```
1 >> format long;
2 >> a = 4/3; b = a-1; c = 3*b; e = 1-c
3 e = 2.220446049250313e-16
4 >> a = 1012/113; b = a-9; c = 113*b; e = 5+c
5 e = 6.750155989720952e-14
6 >> a = 83810206/6789; b = a-12345; c = 6789*b; e = c-1
7 e = -1.607986632734537e-09
8 >> s = sin(10^16*pi)          ← Fehlerverstärker
9 s = -0.375212890012334
```

▷ Abfrage $= 0$ in numerischen Codes unzulässig für berechnete Resultate

◁ Ariane-Unglück 1996: Folge von Rundungsfehlern in Steuersoftware

Listing 5.8: (intersection.m) Fehlerhafte Berechnung des Schnittpunkts zweier Geraden im Raum

```
1  function x = intersection(p1,d1,p2,d2)
2  % MATLAB function purporting to compute the intersection of two lines in 3D
3  % p1,d1 (p2,d2) pass a point and the direction vector of first (second) line
4  % returns the empty matrix in case the lines do not have a common point
5  % BEWARE: this is FLAWED implementation
6  b = p2-p1; A = [d1,-d2]; xi = A\b;
7  if (norm(b-A*xi) ~= 0), % A numerical crime !
8    x = [];
9    disp('No intersection!');
10 else
11   x = p1+xi(1)*d1;
12 end
```

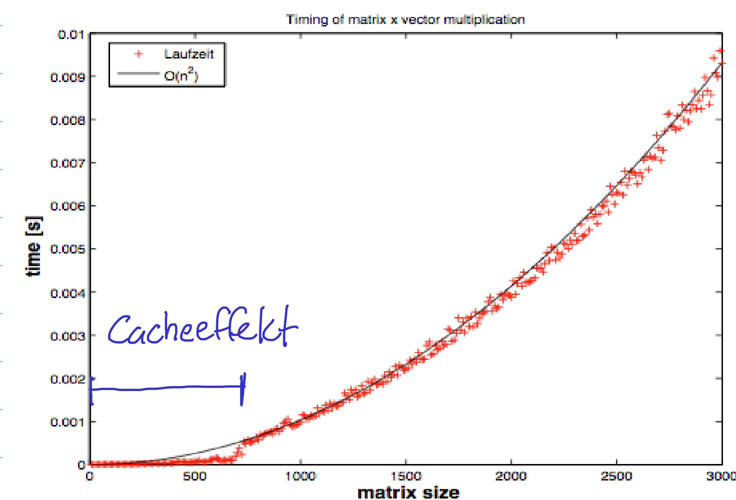
Listing 5.9: (interstbisectors.m) Berechnung des Inkreismittelpunktes eines Raumdreiecks

```
1  % MATLAB script for computing the intersection of the angular bisectors of
2  % a triangle in space
3  T = [1 2 3;1 0 1;4 5 6]; % vertex coordinates in columns of 3x3 matrix
4  % Compute angular bisectors (directions in d1 and d2)
5  p1 = T(:,1); e12 = T(:,2)-T(:,1); e13 = T(:,3)-T(:,1);
6  d1 = e12/norm(e12)+e13/norm(e13);
7  p2 = T(:,2); e21 = T(:,1)-T(:,2); e23 = T(:,3)-T(:,2);
8  d2 = e21/norm(e21)+e23/norm(e23);
9  % Computer intersection (center of incircle of triangle)
10 x = intersection(p1,d1,p2,d2),
```

5.3. Rechenaufwand

```
1  % Timing of matrix×vector operation for dense matrices
2  N = 3000; % maximum size of matrix
3  B = rand(N,N); % initialize random matrix
4  v = rand(N,1); % initialize random column vector
5
6  res = []; % matrix for collecting the results
7  for n=10:10:N
8      A = B(1:n,1:n); % extract submatrix
9      x = v(1:n);
10     t = realmax;
11     for j=1:3, tic; z = A*x; t = min(toc,t); end
12     res = [res; n, t];
13 end
14
15 figure; plot(res(:,1),res(:,2),'r+',...
16     res(:,1),res(:,1).^2/(res(end,1)^2)*res(end,2),'k-');
17 xlabel('{\bf matrix size}','fontsize',14);
18 ylabel('{\bf time [s]}','fontsize',14);
19 title('Timing of matrix x vector multiplication');
20 legend('Laufzeit','O(n^2)','location','best');
21
```

plot(x,y) → Zeichnet Polygon durch $\begin{bmatrix} x_i \\ y_i \end{bmatrix}$



Rechenzeit ~ Vektorlänge²

$(A\underline{x})_i = \underbrace{\sum_{j=1}^{n} (A)_{i,j} \cdot x_j}_{\text{Schleife Länge } n}$ n Mal

$\Rightarrow n^2$ Mult. & $n(n-1)$ Additionen $\sim n^2$

Definition V.3.0.B (Rechenaufwand/Kosten einer Funktion).
Der Rechenaufwand für die Ausführung einer numerischen Funktion ist die Anzahl der elementaren Rechenoperationen '+', '-', '', '/' zuzüglich der Anzahl der Auswertungen von Grundfunktionen wie* **sqrt**, **exp**, **cos**, **sin**, *etc., die während der Auswertung der Funktion ausgeführt werden.*

Definition V.3.0.D ((Polynomialer) asymptotischer Rechenaufwand).
Eine numerische Funktion hat asymptotischen Rechenaufwand oder Komplexität $O(n^q)$, $q > 0$, im Problemgrösenparameter $n \to \infty$ ($n \in \mathbb{N}$), falls es zwei Konstanten $0 < \underline{C} \leqslant \overline{C} < \infty$ und ein $n_0 \in \mathbb{N}$ so gibt, dass

$$\underline{C} n^q \leqslant W(n) \leqslant \overline{C} n^q \quad \forall n \geqslant n_0 ,$$

wobei $W(n)$ der Rechenaufwand der Funktion für ein Problem der Grösse n ist.

Für Matrix x Vektor: Problemgrossenparameter $\Leftrightarrow$ Vektorlänge

↳ $W(n) = 2n^2 - n$

$\Rightarrow n^2 \leq W(n) \leq 2n^2 \Rightarrow O(n^2)$

↑ quadratische Komplexität

Beispiel: Matrixprodukt

```
1  function C = loopmatprod(A,B)
2  % Nested loop implementation of matrix multiplication
3    [n,k] = size(A); % A is an n×k-matrix
4    [kb,m] = size(B); % B is an k×m-matrix
5    if (k ~= kb), error('Mismatch of matrix dimensions'); end
6    C = zeros(n,m);
7    for i=1:n
8       for j=1:m
9          C(i,j) = 0;
10         for l=1:k
11            C(i,j) = C(i,j)+ A(i,l)*B(l,j);
12         end
13      end
14   end
15 end
```

$\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{n,k}$ ▶ Rechenaufwand(A*B) = $O(mnk)$ *

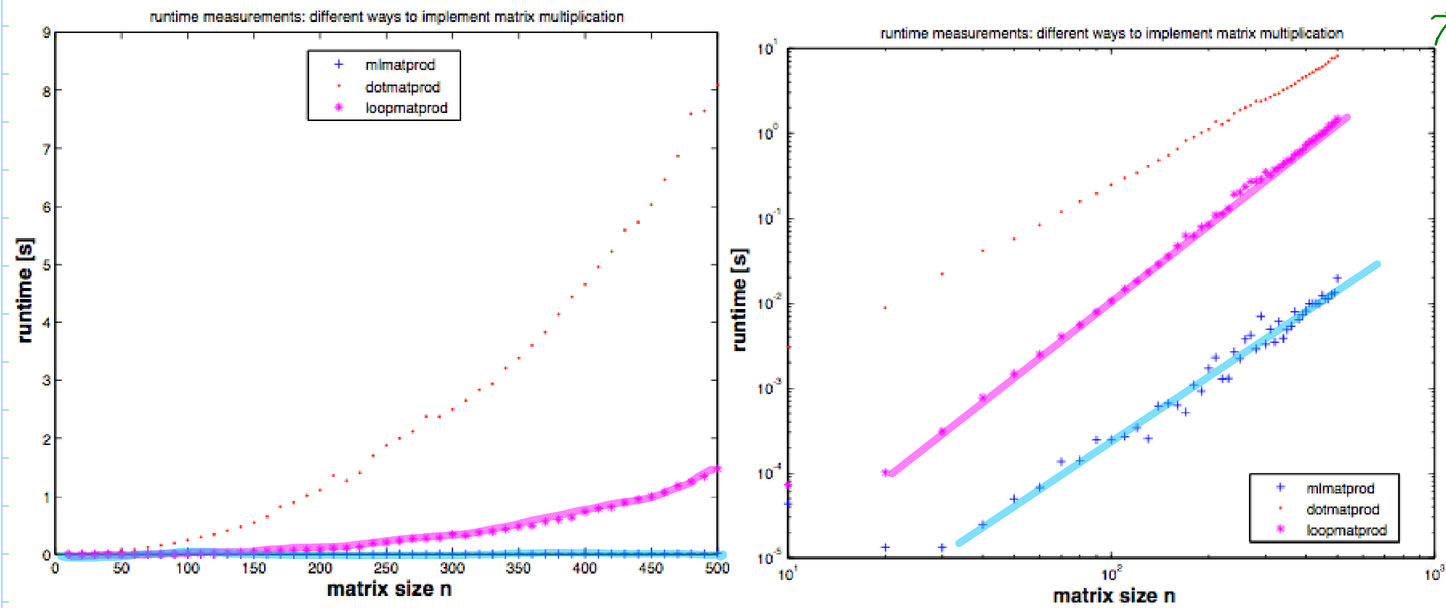
Speziell: $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,n}$ ▶ Rechenaufwand(A*B) = $O(n^3)$

$\underline{C}\, mnk \leq W(m,n,k) \leq \overline{C}\, mnk$

kubische Komplexität

```
1  function C = mlmatprod(A,B)
2  % Standard matrix product in MATLAB
3     C = A*B;
4  end
```

Gleicher Rechenaufwand für beide Implementierung



∇ Rechenaufwand $\not\propto$ Rechenzeit für verschieden Fkt.

Komplexität $\Rightarrow$ Zunahme der Rechenzeit bei Anwachsen der Problemgrösse

* Bem: $W(n) = C n^q$

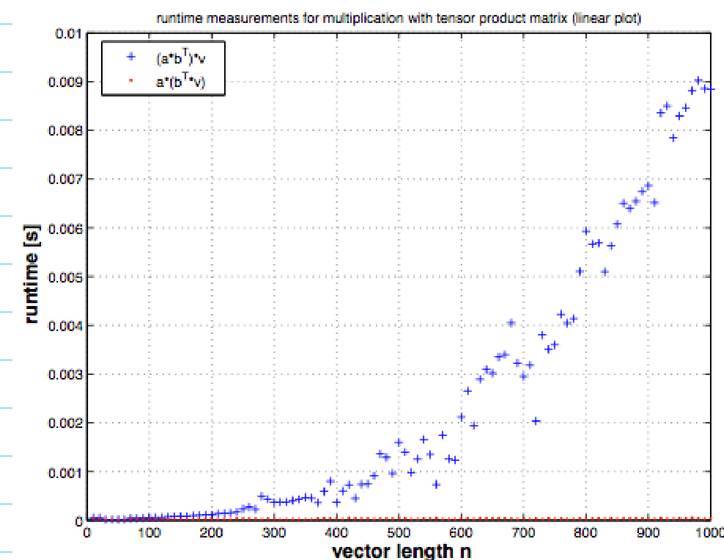
$\Rightarrow$ $\underbrace{\log W(n) = q \log n + \log C}$

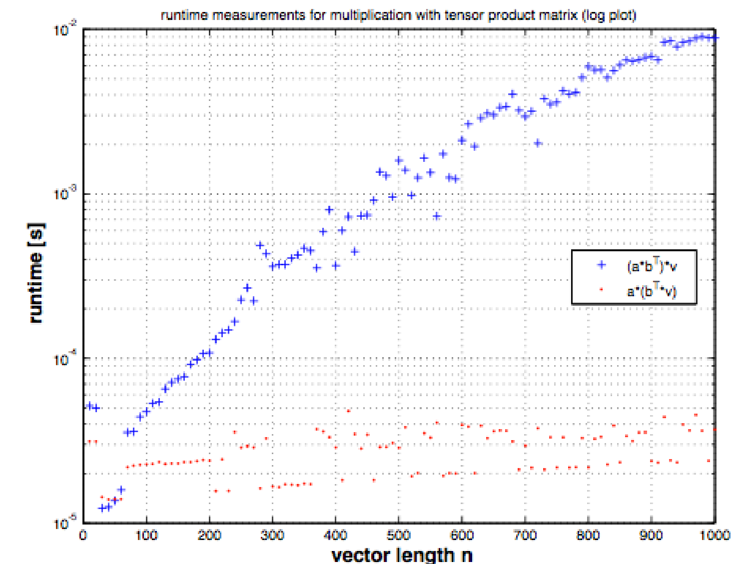
Gerade im doppeltlogarithmischen Plot!

Beispiel V.3.F (Komplexitätsreduktion durch geschickte Organisation von Berechnung)

$v, a, b \in \mathbb{R}^n$ gegeben: $w := \underbrace{a \cdot b^T}_{\in \mathbb{R}^{n,n}} \cdot v \in \mathbb{R}^n$ Tensorprodukt

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





Wenn a,b,v $\leftrightarrow$ $\mathbf{a}, \mathbf{b}, \mathbf{v} \in \mathbb{R}^n$, $n \in \mathbb{N}$:

`w = (a*b')*v`	`w = a*(b'*v);`
Asymptotische Komplexität $O(n^2)$	Asymptotische Komplexität $O(n)$
	[Nur Skalarprodukt & Skalar * Vektor!]

Übersicht: Klassifikation von Operation in numerischer LA nach Komplexität: ($n \triangleq$ Vektorlänge)

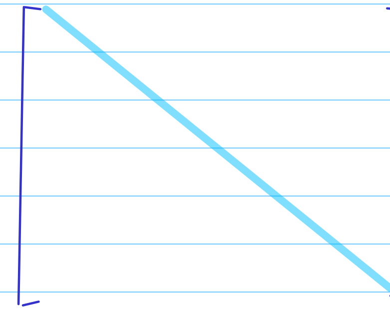
$O(n)$	$O(n^2)$	$O(n^3)$
Vektoraddition	Matrix × Vektor	Matrixprodukt (quadratische Matrizen)
Skalar × Vektor	Tensorprodukt	qr für quadratische Matrizen
Skalarprodukt		Gausselimination

drei geschachtelte Schleifen der Länge n

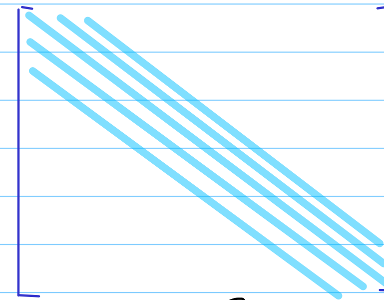
5.4. Dünnbesetzte Matrizen

↳ Matrizen, für die "fast alle" Einträge $= 0$ sind

Beispiele:



Diagonalmatrizen

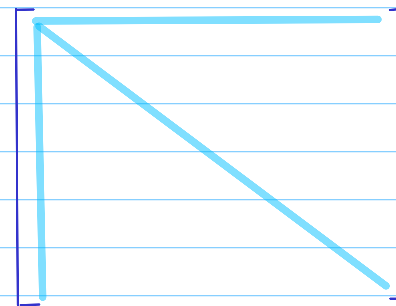


$(A)_{i,j} = 0$, wenn $|i-j| \geq p$, $p \in \mathbb{N}$

Bandmatrix mit Bandbreite

$p = 2 \triangleq$ Tridiagonalmatrix

$p = 3$



Pfeilmatrix

Alle praktisch vorkommenden dünnbesetzten Matrizen haben speziell Struktur.

Implementierung: Spezielle Datenstrukturen

Dünnbesetzte Matrizen in MATLAB:

Initialisierung einer dünnbesetzten $m \times n$-Matrix in MATLAB

```
A = sparse(I,J,a,m,n)
```

$\in \mathbb{R}^{m,n}$

- I ≙ Array von Zeilenindices potentieller Nicht-Null-Einträge
- J ≙ Array von Spaltenindices potentieller Nicht-Null-Einträge
- a ≙ Array von Werten potentieller Nicht-Null-Einträge
- m,n ≙ Anzahl von Zeilen und Spalten

} gleiche Länge k

$1 \leq I(\ell) \leq m$

$1 \leq J(\ell) \leq n$

```
function A = mysparse(I,J,a,m,n)
  k = numel(I);
  if ((numel(J) ~= k) || (numel(a) ~= k))
    error('Length mismatch'); end
  if ((min(I) < 1) || (max(I) > m) || (min(J) < 1) || (max(J) > n))
    error('Index out of range'); end
  A = zeros(m,n);
  for l=1:k,  A(I(l),J(l)) = A(I(l),J(l)) + a(l); end
```

Benutzung der sparse-Datenstruktur ist obligatorisch
(→ sagt dem Rechner, welche Matrixeinträge garantiert = 0 sind)

Beispiel: Multiplikation mit Tridiagonalmatrix

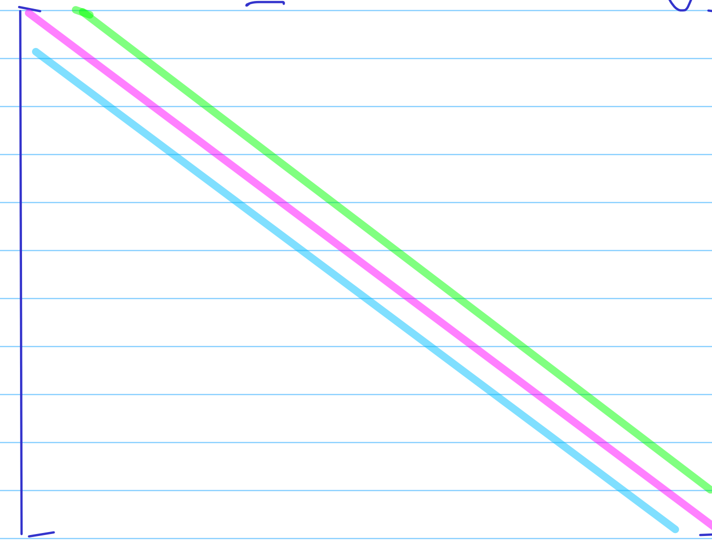
```
function sparsemvtiming
% Measurement of runtimes for matrix×vector multiplication with
% a sparse matrix.
N = 2000;                        % Maximal size of matrix
% Initialize random column vectors of length N and N-1, respectively.
d = rand(N,1); dl = rand(N-1,1); du = rand(N-1,1); v = rand(N,1);
% Initialize dense triadiagonal matrix, see the documntation of the MATLAB
  command
% diag for details.
T_dense = diag(d)+diag(dl,-1)+diag(du,1);
% Initialize sparse triadiagonal matrix
T_sparse = sparse([1:N,1:N-1,2:N],[1:N,2:N,1:N-1],[d;du;dl],N,N);
```

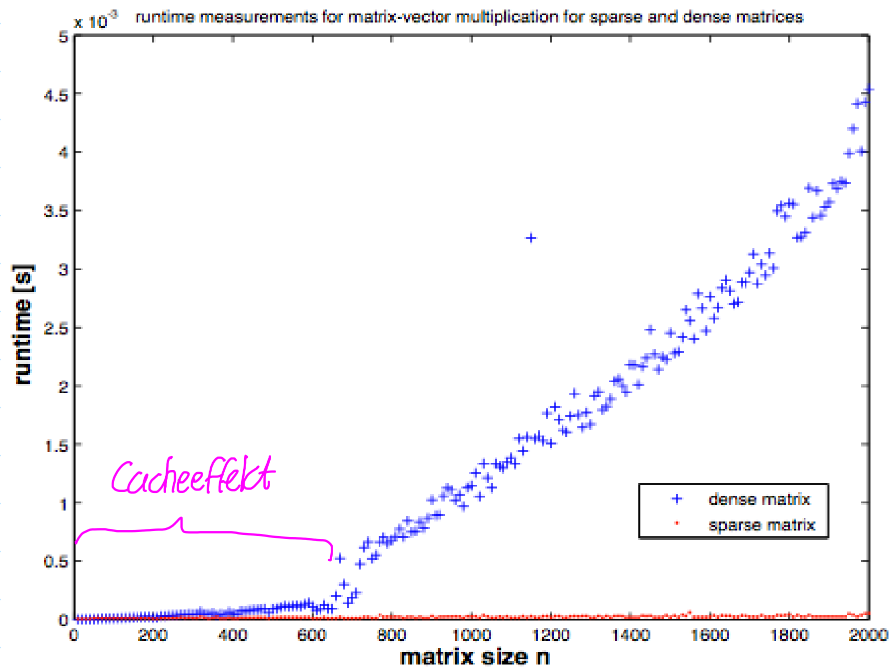
→ Std.-Matrix

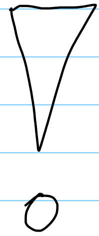
```
res = []; % matrix for recording times
% conduct timings for vectors of different size n
for n=10:10:N
  % Extract sub-matrices, which will be sparse and dense matrices again
  Td = T_dense(1:n,1:n); Ts = T_sparse(1:n,1:n);
  t1 = realmax; for j=1:3, tic; w1 = Td*v(1:n); t1 = min(toc,t1); end
  t2 = realmax; for j=1:3, tic; w2 = Ts*v(1:n); t2 = min(toc,t2); end
  norm(w1-w2), % Check for agreement of results
  res = [res; n, t1, t2];
end
```

```
sparse([1:N,1:N-1,2:N],[1:N,2:N,1:N-1],[d;du;dl],N,N)
```

I, J, a







Bem: "Falle" diag(v) → vollbesetzte Diagonalmatrix

Initialisierung spezieller dünnbesetzter Matrizen in MATLAB:

speye: Einheitsmatrix als dünnbesetzte Matrix

sparse(m,n): $m \times n$-Nullmatrix als dünnbesetzte Matrix

MATLAB-Matrixbaukasten: Matrixblock sparse ⟹ Resultat sparse

```
I = [1:n,1:n-1,2:n,1:(n/2),(n/2+1):n];
J = [1:n,2:n,1:n-1,(n/2+1):n,1:(n/2)];
A = sparse(I,J,[4*ones(n,1);ones(3*n-2,1)],n,n);
```

= [ones(n-1,1); ones(n-1,1); ones(n/2,1); ones(n/2,1)]

5.5. LGS / Lineare Ausgleichsprobleme in MATLAB

→ "Allzweckwaffe" \ - Operator

❶ $\mathbf{A} \in \mathbb{R}^{n,n}$ invertierbar: `x = A\b` berechnet $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ für $\mathbf{b} \in \mathbb{R}^n$

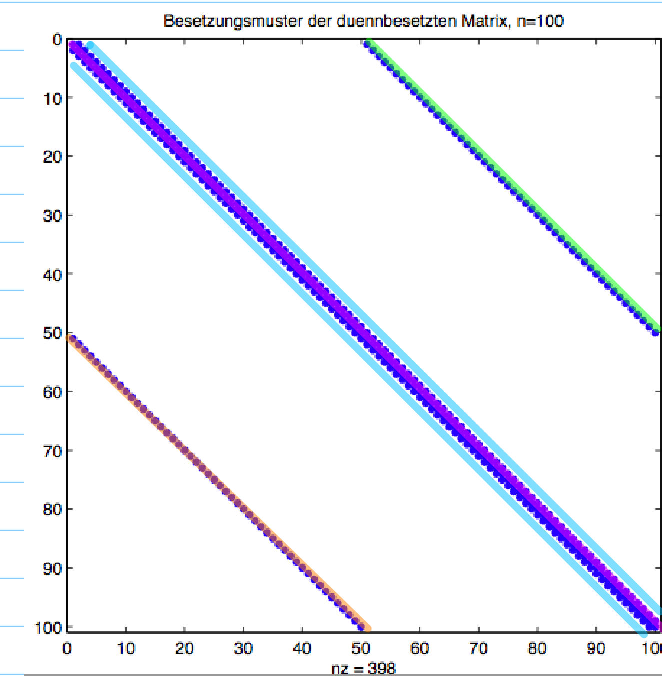
↳ Gausselimination

Rechenaufwand für `x = A\b`: i.a. $O(n^3)$ → für allgemeine Matrizen

$B \in \mathbb{R}^{n,k}$: `A\B` berechnet $A^{-1}B$

Geht i.a. schneller für dünnbesetzte Matrizen:

Bsp V.5.0.B: Dünnbesetztes LGS

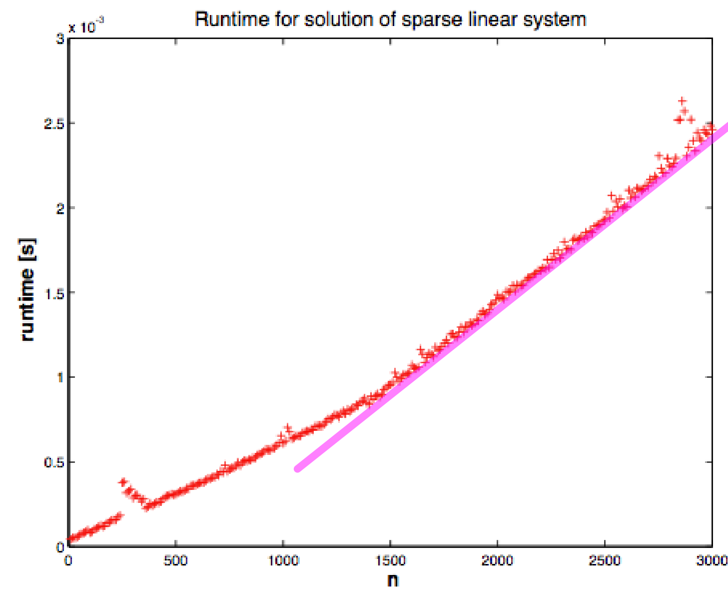


$\mathbf{A} \in \mathbb{R}^{n,n}$, n gerade mit

$$(\mathbf{A})_{i,j} = \begin{cases} 4 & \text{, falls } i = j\,, \\ 1 & \text{, falls } i = j \pm 1\,, \\ 1 & \text{, falls } i = j \pm \frac{n}{2}\,, \\ 0 & \text{sonst.} \end{cases}$$

◁ "spy-plot" der Matrix für $n = 100$.

(erzeugt mit MATLAB-Funktion `spy`)



▷ Asymptotische Komplexität $O(n)$ für \ -Löser

❷ $\mathbf{A} \in \mathbb{R}^{m,n}$, $m > n$: `x = A\b` berechnet Kleinste-Quadrate-Lösung von $\mathbf{Ax} = \mathbf{b}$.

Rechenaufwand für `x = A\b`: i.a. $O(mn^2)$

↑ lineare Komplexität in der Anzahl der Zeilen