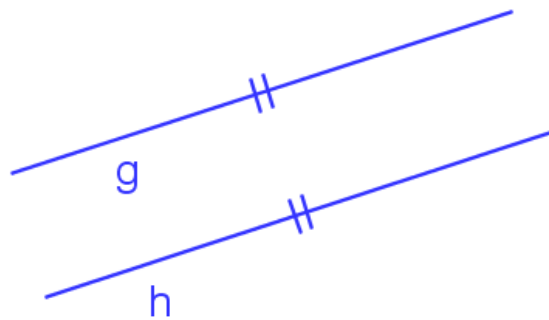


CG-Verfahren für dünnbesetzte Matrizen

*Rafael Dorigo
Sebastian Hirnschall*

*Betreut von:
Markus Wess Dipl.-Ing.*



Inhaltsverzeichnis

1	Einleitung	3
2	Motivation	4
3	CG-Verfahren (Verfahren der konjugierten Gradienten)	6
3.1	Dünnbesetzte Matrizen	9
3.2	Vorkonditionierung	12
A	Verwendete Klassen	15
A.1	Size	15
A.2	Vektor	15
A.3	Vollbesetzte Matrix	22
A.4	Dünnbesetzte Matrix	36

1. Einleitung

Das Projekt beschäftigt sich mit dem Lösen linearer Gleichungssysteme der Form $Ax = b$. Dabei werden verschiedene iterative Verfahren vorgestellt und deren Aufwand verglichen. Außerdem wird eine effiziente Methode gezeigt, dünnbesetzte Matrizen zu speichern. Es folgen Plots zur Veranschaulichung der Konvergenzgeschwindigkeit der Verfahren.

2. Motivation

Um für eine symmetrische positiv definite Koeffizientenmatrix $A \in \mathbb{R}^{n \times n}$ und $b \in \mathbb{R}^n$ die Gleichung $Ax = b$ zu lösen, treten bei der Cholesky-Zerlegung $\frac{1}{3}n^3 + \mathcal{O}(n^2)$ arithmetische Operationen auf.

Zum Testen kann A folgendermaßen generiert werden.

```
1  template<>
2  void linag::DenseMatrix<double>::randSPD(int notZeroPerLine) {
3      assert(isSymmetric() && notZeroPerLine <= dim().cols &&
4             notZeroPerLine%2);
5
6      randLT();
7      double c = 50;
8      linag::DenseMatrix<double> diagM(dim());
9      diagM.randDiag();
10
11     (*this) = (*this) + transpose() + c * diagM;
12
13     for (int i = 0; i < dim().rows; ++i) {
14         for (int j = i+std::ceil((double)notZeroPerLine/2); j <
15             dim().cols; ++j) {
16             at(i,j) = 0;
17         }
18         for (int j = 0; j <= i-std::ceil((double)notZeroPerLine
19             /2); ++j) {
20             at(i,j) = 0;
21         }
22     }
23 }
```

Listing 1: Erstellen einer symmetrisch positiv definiten Zufallsmatrix mit einer fixen Anzahl an Einträgen ungleich 0 pro Zeile in C++

BEMERKUNG. Die Implementierung aller Klassen und Funktionen im *linag*-namespace sind im Anhang zu finden. Zum Vergleich wurde die Eigen-Bibliothek verwendet. (<http://eigen.tuxfamily.org>)

Wie in Abb. 1 zu sehen ist, ist das direkte Lösen bei großen Problemen nicht praktikabel, da die benötigte Zeit kubisch mit der Problemgröße steigt.

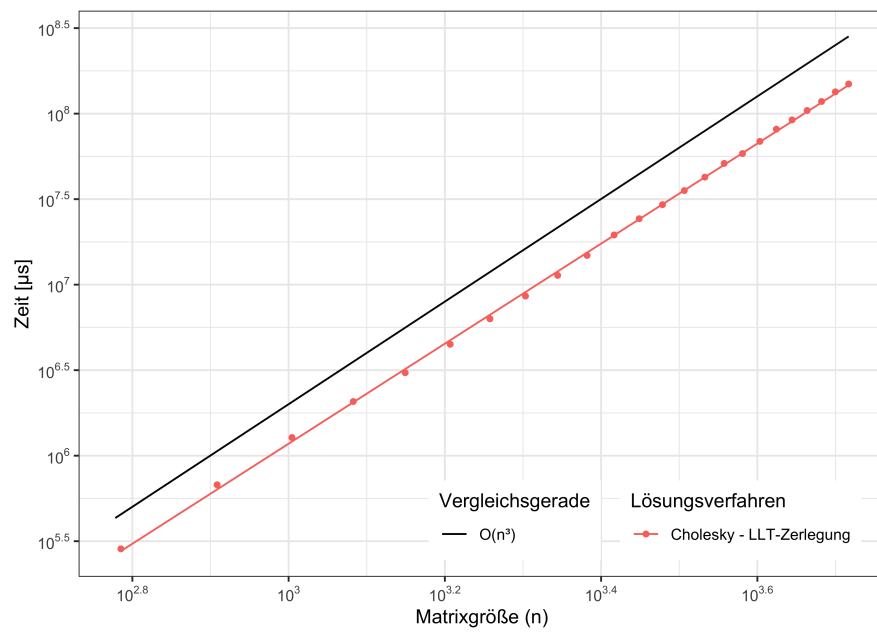


Abbildung 1: Benötigte Zeit in μs um Gleichungssysteme mit verschieden großen dicht besetzten $n \times n$ -Koeffizientenmatrizen mittels Cholesky-Zerlegung zu lösen

3. CG-Verfahren (Verfahren der konjugierten Gradienten)

Ist $A \in \mathbb{R}^{n \times n}$ eine symmetrisch positiv definite Matrix, dann kann die Lösung x mithilfe des CG-Verfahrens beliebig genau approximiert werden.

Das CG-Verfahren ist äquivalent zur Minimierung der Energiefunktion $\phi(x) = \frac{1}{2}x^T Ax - x^T b$, also $\nabla \phi(x) = Ax - b = 0$. Weil A positiv definit ist, ist x ein Minimum. Man rechnet für einen zufälligen Vektor x_0 das Residuum $r_0 = b - Ax_0$ aus und nähert sich dann rekursiv der exakten Lösung x an. Dabei ist nach höchstens n Iterationen $\|x_t - x\| = 0$.

Algorithm 1 CG-Verfahren

Input: Sei $A \in \mathbb{R}^{n \times n}$, $b, x_0 \in \mathbb{R}^n$ und eine Toleranz $\tau > 0$

```

1:  $r_0 = b - Ax_0$ 
2:  $d_0 = r_0$ 
3:  $t = 0$ 
4: while  $\|r_t\| > \tau$  do
5:    $z = Ad_t$ 
6:    $\alpha_t = \frac{r_t^T r_t}{d_t^T z}$ 
7:    $x_{t+1} = x_t + \alpha_t d_t$ 
8:    $r_{t+1} = r_t - \alpha_t z$ 
9:    $\beta_t = \frac{r_{t+1}^T r_{t+1}}{r_t^T r_t}$ 
10:   $d_{t+1} = r_{t+1} + \beta_t d_t$ 
11:   $t = t + 1$ 

```

Output: Näherung x_t an $x = A^{-1}b$ mit $\|Ax_t - b\| < \tau$

Nun stellt sich die Frage, ob Algorithmus 1 äquivalent zum Algorithmus 8.10 (Nannen 2019, S. 101) ist und welcher zu bevorzugen ist.

In Algorithmus 1 werden pro Iteration neben Vektor-Vektor Multiplikationen auch zwei Matrix-Vektor Multiplikation durchgeführt. Dabei ist der Aufwand um eine $n \times n$ -Matrix mit einem Vektor zu multiplizieren $\mathcal{O}(n^2)$ und um einen Vektor mit einem Vektor zu multiplizieren $\mathcal{O}(n)$. Da beide Matrix-Vektor Multiplikationen in Algorithmus 1 gleich sind, kann das Ergebnis gespeichert werden um die Zahl der Matrix-Vektor Multiplikationen pro Iteration auf eine zu senken. In Algorithmus 8.10 werden ebenfalls zwei Matrix-Vektor Multiplikationen pro Iteration durchgeführt. Da diese jedoch unterschiedlich sind, kann das Ergebnis nicht gespeichert werden, wodurch Algorithmus 1 effizienter ist. Es bleibt noch zu zeigen, dass die beiden Algorithmen dasselbe Ergebnis liefern.

Man sieht offensichtlich, dass sich die Algorithmen nur in den While-Schleifen un-

terscheiden. Zuerst sei bei Algorithmus 8.10 angemerkt, dass man Zeile 7 ans Ende der While-Schleife verschieben kann und die t in Zeile 8-10 durch $t + 1$ ersetzen kann.

Durch Multiplizieren von A von links in Zeile 6 im Algorithmus 8.10 folgt

$$Ax_{t+1} = Ax_t + \alpha_t Ad_t$$

und somit

$$r_{t+1} = b - Ax_{t+1} = b - Ax_t - \alpha_t Ad_t = r_t - \alpha_t Ad_t \quad (3.1)$$

Gleichung (3.1) kann man umformen in

$$r_{t+1} = r_t - \alpha_t Ad_t \Leftrightarrow Ad_t = \frac{1}{\alpha_t}(r_t - r_{t+1})$$

Das heißt der Zähler von β_t kann folgendermaßen umgeschrieben werden

$$r_{t+1}^T Ad_t = \frac{1}{\alpha_t} r_{t+1}^T (r_t - r_{t+1}) = -\frac{1}{\alpha_t} r_{t+1}^T r_{t+1}$$

da die Residieen orthogonal sind und der Nenner

$$d_t^T Ad_t = (r_t + \beta_{t-1} d_{t-1})^T Ad_t = \frac{1}{\alpha_t} r_t^T (r_t - r_{t+1}) = \frac{1}{\alpha_t} r_t^T r_t$$

da die Suchrichtungen d_t A -orthogonal sind. Also folgt insgesamt

$$\beta_t = -\frac{r_{t+1}^T Ad_t}{d_t^T Ad_t} = -\frac{-\frac{1}{\alpha_t} r_{t+1}^T r_{t+1}}{\frac{1}{\alpha_t} r_t^T r_t} = \frac{r_{t+1}^T r_{t+1}}{r_t^T r_t}$$

Außerdem folgt durch einsetzen von Zeile 10 in Zeile 5

$$\alpha_t = \frac{r_t^T d_t}{d_t^T Ad_t} = \frac{r_t^T (r_t + \beta_{t-1} d_{t-1})}{d_t^T Ad_t} = \frac{r_t^T r_t}{d_t^T Ad_t}$$

da (vgl. Nannen 2019, S. 100)

$$r_t^T d_j = 0 \quad \forall 0 \leq j < t$$

Für die Iteration des CG-Verfahrens (ohne Abbruchkriterium) gilt die Fehlerabschätzung (vgl. ebd., S. 102)

$$\|x^{(t)} - A^{-1}b\|_A \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^t \|x^{(0)} - A^{-1}b\|_A, \quad t \in \mathbb{N}$$

mit der spektralen Konditionszahl der Matrix $\kappa(A) := \left| \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)} \right|$ und der Energienorm $\|x\|_A := \sqrt{(x, x)_A}$. Das Verfahren konvergiert für $\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} < 1$, was offensichtlich gegeben ist, da $\kappa \geq 1$. Die Konvergenzgeschwindigkeit ist größer, wenn der Bruch minimal ist, also genau dann wenn die Eigenwerte der Matrix eng zusammenliegen. Wenn alle Eigenwerte gleich sind, dann gilt $\kappa = 1$ also $\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} = 0$.

```

1  template <typename T>
   linag::Vector<T> linag::DenseMatrix<T>::conjugateGradientSolver(linag
   ::Vector<T> b, double tau, int* count, linag::Vector<linag::Vector<
   double>*>* xs, linag::Vector<double>* rs){
3   assert(tau>0 && dim().rows == b.length());
   if(xs)
5       assert(xs->length() == dim().rows); //exact result after n
   iterations
   if(rs)
7       assert(rs->length() == dim().rows); //exact result after n
   iterations

9   linag::Vector<T> r1(dim().rows);
   linag::Vector<T> r2(dim().rows);
11  linag::Vector<T> d(dim().rows);
   linag::Vector<T> x(dim().rows);
13  linag::Vector<T> z(dim().rows);
   x.rand();
15  T alpha;
   T betta;
17  unsigned long t = 0;
   r1 = b - (*this)*x;
19  d = r1;
   if(xs) {
21     for (int i = 1; i < xs->length(); ++i) {
         xs->at(i) = nullptr;
23     }
     xs->at(0) = new linag::Vector<double>(x);
25 }
   if(rs) {
27     for (int i = 1; i < rs->length(); ++i) {
         rs->at(i) = 0;
29     }
     rs->at(0) = r1.l2norm();
31 }
   do{
33     z = (*this)*d;
     alpha = (r1*r1)/(d*z);
35     x = x + alpha*d;
     r2 = r1 - alpha*z;
37     betta = (r2*r2)/(r1*r1);
     d = r2 + betta*d;

39     r1=r2;
     if(xs && t < xs->length())
41         xs->at(t) = new linag::Vector<double>(x);
     if(rs && t < rs->length())
43         rs->at(t) = r2.l2norm();

```



```

45     ++t;
    }while (r2.l2norm()>tau);
47     if(count)
        *count = t;
49     return x;
}

```

Listing 2: Implementierung des CG-Verfahrens in C++

3.1. Dünnbesetzte Matrizen

Da man beim Lösen linearer Gleichungssysteme oft mit großen, dünnbesetzten Matrizen arbeitet, ist es sinnvoll, nur Einträge ungleich Null zu speichern. Dafür kann zum Beispiel das sogenannte *compressed sparse row* Format verwendet werden. Bei der Implementierung dieses Formats werden anstelle aller Einträge $A_{i,j}$, $i, j = 1, \dots, n$ einer Matrix $A \in \mathbb{R}^{n \times n}$ ein Vektor $v \in \mathbb{R}^m$ aller Einträge ungleich Null, ein Vektor $J \in \mathbb{N}_0^m$ von Spaltenindizes und ein Vektor $I \in \mathbb{N}_0^{n+1}$ gespeichert. Die i -te Zeile von A ist gegeben durch

$$A_{i,j} = \begin{cases} v_{k(j)}, & \text{falls } j \in \{J_{I_i}, J_{I_i} + 1, \dots, J_{I_{i+1}} - 1\} \\ 0, & \text{sonst} \end{cases}$$

wobei $J_{k(j)} = j$.

Eine Möglichkeit das Format zu implementieren ist wie folgt:

```

template <typename T>
2 linag::SparseMatrix<T>::SparseMatrix(const linag::DenseMatrix<T>& rhs
    ):
    I(0),J(0),v(0),dimension(rhs.dim()){
4     //calculate array size
    int vc = 0;
6     int Ic = rhs.dim().rows+1;
    for (int i = 0; i < rhs.dim().rows; ++i) { //rows
8         for (int j = 0; j < rhs.dim().cols; ++j) { //cols
            if(std::fabs(rhs.at(i,j)) > 10e-10){
10                 ++vc;
            }
12         }
    }
14     //set array size
    I = linag::Vector<int>(Ic);
16     J = linag::Vector<int>(vc);
    v = linag::Vector<T>(vc);
18
    //convert dense matrix to sparse matrix

```

3 CG-VERFAHREN (VERFAHREN DER KONJUGIERTEN GRADIENTEN)

```
20     vc=0;
    Ic=-1;
22     int Jc=0;
    for (int i = 0; i < rhs.dim().rows; ++i) { //rows
24         for (int j = 0; j < rhs.dim().cols; ++j) { //cols
            if(std::fabs(rhs.at(i,j)) > 10e-10){
26                 if(Ic != i){
                    I.at(++Ic) = vc;
28                 }
                    v.at(vc++) = rhs.at(i,j);
30                 J.at(Jc++) = j;
            }
32         }
    }
34     I.at(++Ic) = vc;
}
```

Listing 3: Speicherung einer Matrix im compressed spare row Format

```
template <typename T>
2 linag::DenseMatrix<T>::DenseMatrix(const linag::SparseMatrix<T>& rhs)
    :dimension(rhs.dim()){
    if(dim().rows*dim().cols > 0)
4    {
        data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
6        assert(data != nullptr);

        zeros();
8        for (int i = 0; i < dim().rows; ++i) {
            for (int j = rhs.getI().at(i); j < rhs.getI().at(i+1); ++
10 j) {
                at(i,rhs.getJ().at(j))=rhs.getV().at(j);
12            }
        }
14    }
    else
16    data = (T*) nullptr;
}
```

Listing 4: compressed spare row Format zu vollbesetzter Matrix

Die Implementierung des CG-Verfahrens unterscheidet sich dabei nicht von Listing 2. Die Matrix-Vektor Multiplikation kann für dünnbesetzte Matrizen jedoch effizienter implementiert werden. (Listing 5)

```
template<typename T>
2 const linag::Vector<T> linag::operator*(const linag::SparseMatrix<T>&
    x,const linag::Vector<T>& y){
    assert(x.dim().cols == y.length());
4    linag::Vector<T> res(y.length());
```

3 CG-VERFAHREN (VERFAHREN DER KONJUGIERTEN GRADIENTEN)

```
res.zeros();
6 for (int i = 0; i < res.length(); ++i) {
    for (int j = x.getI().at(i); j < x.getI().at(i + 1); ++j) {
8         res.at(i) += y.at(x.getJ().at(j)) * x.getV().at(j);
    }
10 }
    return res;
12 }
```

Listing 5: Überladen der Matrix-Vektor Multiplikation für dünnbesetzte Matrizen

In Abb. 2 ist zu sehen, dass das CG-Verfahren für dünn besetzte Matrizen im *compressed sparse row* Format wesentlich effizienter berechnet werden kann, als für vollbesetzte Matrizen bzw. als solche gespeicherten Matrizen. Wie in Zeile 5 in Algorithmus 1 zu sehen ist, ist der unterschiedliche Aufwand von einer Matrix-Vektor-Multiplikation abhängig. Der Aufwand um eine vollbesetzte $n \times n$ -Matrix mit einem Vektor zu multiplizieren entspricht $\mathcal{O}(n^2)$. Der Aufwand um eine dünnbesetzte Matrix mit einem Vektor zu multiplizieren ist mit $\mathcal{O}(n)$ linear und davon abhängig wie dicht die Matrix besetzt ist. Außerdem ist zu sehen, wie sich die benötigte Zeit zur Durchführung des CG-Verfahrens mit der Anzahl an Einträgen ungleich Null pro Zeile der Matrix ändert.

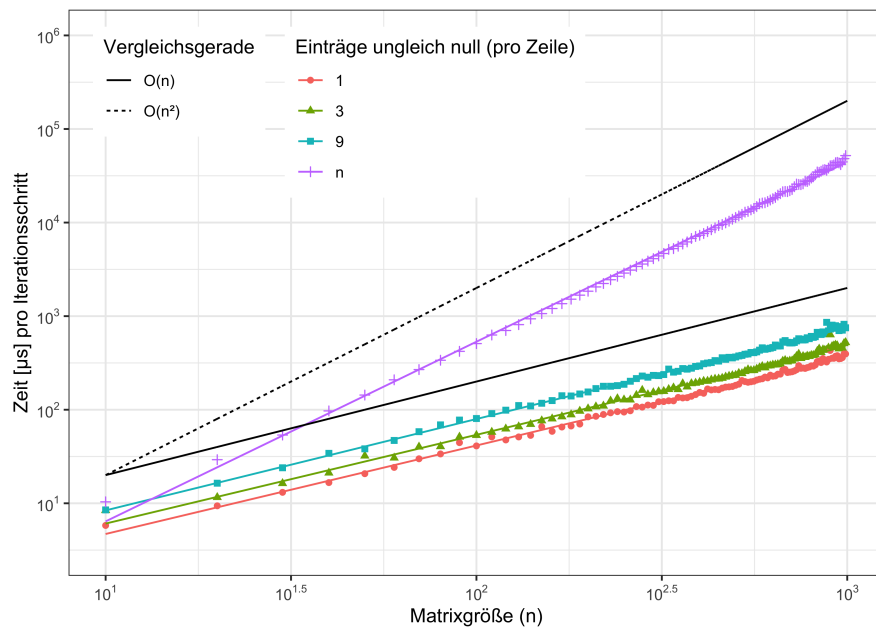


Abbildung 2: Benötigte Zeit in μs für verschieden dicht besetzte $n \times n$ -Matrizen im *compressed sparse row*-Format

3.2. Vorkonditionierung

Die spektrale Konditionszahl definiert die Konvergenzgeschwindigkeit des CG-Verfahrens. Durch Lösen des vorkonditionierten Systems

$$D^{-1}AD^{-T}y = D^{-1}b$$

kann die Konvergenz beschleunigt werden und man bekommt eine Lösung x der Form

$$x = D^{-T}y$$

Man wählt die Matrix D so, dass für beliebige $z \in \mathbb{R}^n$ der Vektor $D^{-T}D^{-1}z$ einfach zu berechnen ist und zugleich $\text{cond}(D^{-1}AD^{-T}) < \text{cond}(A)$ gilt. Das heißt, mithilfe der Matrix D liegen die Eigenwerte enger zusammen und die spektrale Konditionszahl wird kleiner.

Algorithm 2 Vorkonditioniertes CG-Verfahren

Input: Sei $A \in \mathbb{R}^{n \times n}$, $x_0 \in \mathbb{R}^n$, $P := DD^T$ und eine Toleranz $\tau > 0$

```

1:  $r_0 = b - Ax_0$ 
2:  $z_0 = P^{-1}r_0$ 
3:  $d_0 = z_0$ 
4:  $t = 0$ 
5: while  $\|r_t\| > \tau$  do
6:    $z = Ad_t$ 
7:    $\alpha_t = \frac{r_t^T z_t}{d_t^T z}$ 
8:    $x_{t+1} = x_t + \alpha_t d_t$ 
9:    $r_{t+1} = r_t - \alpha_t z$ 
10:   $z_{t+1} = P^{-1}r_{t+1}$ 
11:   $\beta_t = \frac{z_{t+1}^T r_{t+1}}{z_t^T r_t}$ 
12:   $d_{t+1} = z_{t+1} + \beta_t d_t$ 
13:   $t = t + 1$ 

```

Output: Näherung x_t an $x = A^{-1}b$ mit $\|Ax_t - b\| < \tau$

Im Gegensatz zu Algorithmus 1 sind in Algorithmus 2 pro Iteration zwei Matrix-Vektor Multiplikationen notwendig. Für dünnbesetzte Matrizen bleibt der Aufwand mit $\mathcal{O}(n)$ also linear, wird jedoch, um eine von der Koeffizientenmatrix abhängige multiplikative Konstante größer. Die für einen Iterationsschritt benötigte Zeit ist zusammen mit einer Vergleichsgerade für $\mathcal{O}(n)$ in Abb. 3 zu sehen.

3 CG-VERFAHREN (VERFAHREN DER KONJUGIERTEN GRADIENTEN)

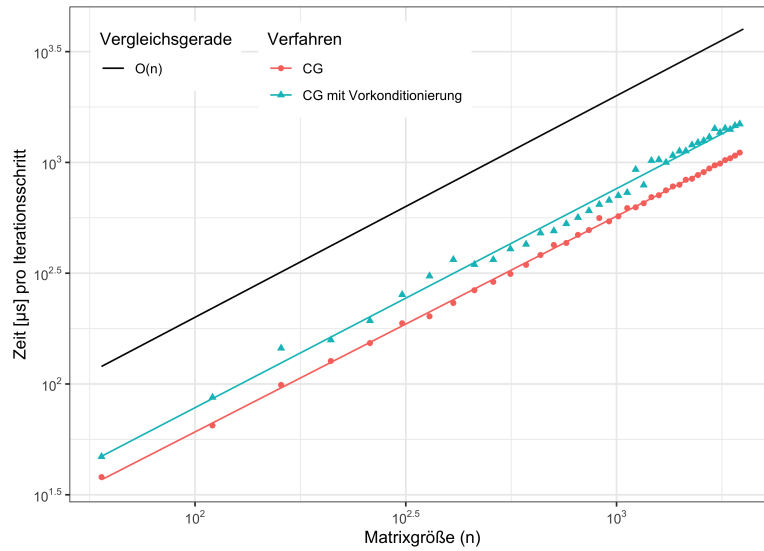
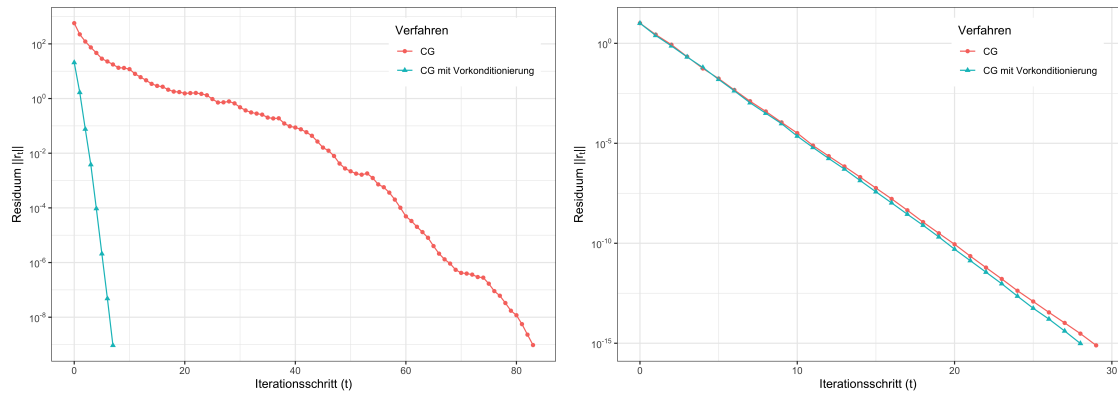


Abbildung 3: Residuum nach Iterationsschritt t für vorkonditioniertes CG- und CG-Verfahren

In Abb. 4 ist zu sehen, dass das Vorkonditionieren des Problems je nach Problem unterschiedlich gut funktioniert. Für strikt diagonaldominante Koeffizientenmatrizen wie in Abb. 4a sind mit $P = \text{diag}(A_{11}, \dots, A_{nn})$ aufgrund der kleineren Konditionszahl wesentlich weniger Iterationsschritte nötig, bzw. das vorkonditionierte CG-Verfahren konvergiert schneller. Dies funktioniert wie in Abb. 4b nicht für alle Koeffizientenmatrizen.

3 CG-VERFAHREN (VERFAHREN DER KONJUGIERTEN GRADIENTEN)



(a) Koeffizientenmatrix der Form $A = B + B^T + c \cdot \text{diag}(b)$ ¹ (b) Koeffizientenmatrix der Form $A = B + B^T + c \cdot I_n$

Abbildung 4: Residuum nach Iterationsschritt t für vorkonditioniertes CG- und CG-Verfahren

Für Matrizen bei denen das Vorkonditionieren gut funktioniert, steigt die Iterationszahl für größer werdende Matrizen mit Vorkonditionierung wesentlich langsamer und die Streuung ist geringer als ohne, wie an den Trendlinien in Abb. 5 zu sehen ist.

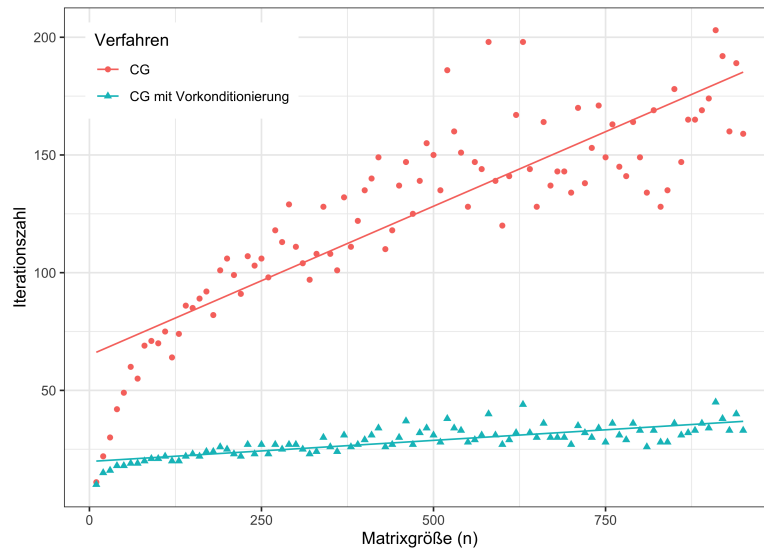


Abbildung 5: Iterationszahlen für verschieden große Koeffizientenmatrizen der Form $A = B + B^T + c \cdot \text{diag}(b)$

¹ $A, B \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n, c \in \mathbb{R}, n \in \mathbb{N}$

A. Verwendete Klassen

A.1. Size

```
1  #ifndef AUFGABE1_SIZE_H
2  #define AUFGABE1_SIZE_H
3
4  namespace linag {
5      class Size {
6      public:
7          int rows, cols;
8          //default operator=, shallow copy
9
10         };
11
12         bool operator==(const Size& lhs, const Size& rhs)
13         {
14             return lhs.rows == rhs.rows && lhs.cols == rhs.cols;
15         }
16
17         bool operator!=(const Size& lhs, const Size& rhs)
18         {
19             return !(lhs == rhs);
20         }
21     }
22
23     #endif //AUFGABE1_SIZE_H
```

Listing 6: size.h

A.2. Vektor

```
1  #ifndef AUFGABE1_VECTOR_H
2  #define AUFGABE1_VECTOR_H
3
4  #include <iostream>
5  #include <cstdlib>
6  #include <time.h>
7  #include <cmath>
8  #include <cassert>
9  #include <cstring>
10 #include "Eigen/Dense"
11 #include "size.h"
12 // #include "Eigen/src/Core/Matrix.h"
13
14 namespace linag {
```

```
16 //say class exists without defining it
17 template <typename T> class DenseMatrix;
18
19
20 template<typename T>
21 class Vector{
22 private:
23     Size dimension;
24     T *data;
25 public:
26     ~Vector();
27
28     Vector(const Vector<T> &rhs);
29     Vector<T> &operator=(const Vector<T> &);
30     explicit Vector(int rows,int cols = 1);
31     explicit Vector(const Size dimension);
32
33     Vector(std::initializer_list<T> init);
34
35     Eigen::VectorXd toEigen() const;
36     const Vector<T> operator-() const;
37
38     T& at(int index);
39     const T &at(int index) const;
40
41     const Size dim() const;
42     unsigned long length() const;
43
44     double l2norm();
45     double Anorm(const linag::DenseMatrix<T>& A) const;
46
47     void zeros();
48     void ones();
49     void rand();
50
51 };
52
53 template<typename T>
54 const Vector<T> operator+(const Vector<T>& x,const Vector<T>& y);
55 template<typename T>
56 const Vector<T> operator-(const Vector<T>& x,const Vector<T>& y);
57 template<typename T>
58 T operator*(const Vector<T>& x,const Vector<T>& y);
59 template<typename T>
60 const Vector<T> operator*(const Vector<T>& x,const T y);
61 template<typename T>
62 const Vector<T> operator/(const Vector<T>& x,const T y);
63 template<typename T>
64 const Vector<T> operator*(const T x,const Vector<T>& y);
```



```
64     template<typename T>
65     std::ostream& operator<< (std::ostream& output, const Vector<T>& x)
66     ;
67 }
68
69 template<typename T>
70 std::ostream& linag::operator<< (std::ostream& output, const linag::
71     Vector<T>& x){
72     for (int i = 0; i < x.dim().rows; ++i) {
73         for (int j = 0; j < x.dim().cols; ++j) {
74             output << x.at(i+j) << ",\t";
75         }
76         output << '\n';
77     }
78     return output;
79 }
80
81 template<typename T>
82 const linag::Vector<T> linag::operator* (const T x, const linag::Vector
83     <T>& y){
84     linag::Vector<T> res(y.dim());
85     for (int i = 0; i < y.dim().rows * y.dim().cols; ++i) {
86         res.at(i) = x*y.at(i);
87     }
88     return res;
89 }
90
91 template<typename T>
92 const linag::Vector<T> linag::operator* (const linag::Vector<T>& x,
93     const T y){
94     linag::Vector<T> res(x.dim());
95     for (int i = 0; i < x.dim().rows * x.dim().cols; ++i) {
96         res.at(i) = x.at(i) * y;
97     }
98     return res;
99 }
100
101 template<typename T>
102 T linag::operator* (const linag::Vector<T>& x, const linag::Vector<T>&
103     y){
104     assert(x.dim().rows*x.dim().cols == y.dim().rows*y.dim().cols);
105     T res = (T)0;
106     for (int i = 0; i < x.dim().rows*x.dim().cols; ++i) {
107         res+= y.at(i) *x.at(i);
108     }
109     return res;
110 }
```

```
108 }
110
112 template<typename T>
113 const linag::Vector<T> operator/(const linag::Vector<T>& x, const T y)
114 {
115     linag::Vector<T> res(x);
116     for (int i = 0; i < res.dim().cols*res.dim().rows; ++i) {
117         res.at(i)/=y;
118     }
119     return res;
120 }
121
122 template<typename T>
123 const linag::Vector<T> linag::operator-(const linag::Vector<T>& x,
124     const linag::Vector<T>& y){
125     return x + (-y);
126 }
127
128 template<typename T>
129 const linag::Vector<T> linag::operator+(const linag::Vector<T>& x,
130     const linag::Vector<T>& y){
131     assert(x.dim() == y.dim());
132     linag::Vector<T> res(x.dim());
133     for (int i = 0; i < x.dim().rows*x.dim().cols; ++i) {
134         res.at(i) = x.at(i) + y.at(i);
135     }
136     return res;
137 }
138
139 template <typename T>
140 void linag::Vector<T>::zeros(){
141     for (int i = 0; i < length(); ++i) {
142         at(i) = 0;
143     }
144 }
145
146 template <typename T>
147 void linag::Vector<T>::ones(){
148     for (int i = 0; i < length(); ++i) {
149         at(i) = 1;
150     }
151 }
152
153 template <typename T>
154 void linag::Vector<T>::rand(){
155     for (int i = 0; i < dim().rows*dim().cols; ++i) {
156         at(i) = (T)std::rand()/RAND_MAX;
157     }
158 }
```

```
154     }
155 }
156
157 template<typename T>
158 double linag::Vector<T>::l2norm(){
159     double sum = 0;
160     for (int i = 0; i < length(); ++i) {
161         sum += std::fabs(double((at(i)*at(i))));
162     }
163     return std::sqrt(sum);
164 }
165
166 template<typename T>
167 double linag::Vector<T>::Anorm(const linag::DenseMatrix<T>& A) const{
168     return std::sqrt((*this) * A * (*this));
169 }
170
171 template <typename T>
172 unsigned long linag::Vector<T>::length() const{
173     return dim().rows*dim().cols;
174 }
175
176 template <typename T>
177 const linag::Size linag::Vector<T>::dim() const{
178     return dimension;
179 }
180
181 template <typename T>
182 const T &linag::Vector<T>::at(int index) const{
183     assert(index>=0 && index <dim().rows*dim().cols);
184     return data[index];
185 }
186
187 template <typename T>
188 T &linag::Vector<T>::at(int index){
189     assert(index>=0 && index <dim().rows*dim().cols);
190     return data[index];
191 }
192
193 template <typename T>
194 const linag::Vector<T> linag::Vector<T>::operator-() const{
195     return (T)-1* (*this);
196 }
197
198 template <typename T>
199 linag::Vector<T>::Vector(std::initializer_list<T> init){
200     dimension.rows = init.size();
201     dimension.cols = 1;
202     if(dim().rows*dim().cols > 0)
```

```
204     {
205         data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
206         assert(data != nullptr);
207         //copy
208         int i=0;
209         for(auto item:init) {
210             at(i) = item;
211             ++i;
212         }
213     }
214     else
215         data = (T*) nullptr;
216 }
217
218 template <typename T>
219 linag::Vector<T>::Vector(const linag::Size dimension):dimension(
220     dimension){
221     assert(dimension.rows == 1 || dimension.cols == 1);
222     if(dim().rows*dim().cols > 0)
223     {
224         data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
225         assert(data != nullptr);
226     }
227     else
228         data = (T*) nullptr;
229 }
230
231 template <typename T>
232 linag::Vector<T>::Vector(int rows,int cols){
233     assert(rows == 1 || cols == 1);
234     dimension.rows=rows;
235     dimension.cols=cols;
236     if(rows*cols > 0)
237     {
238         data = (T*) malloc(rows * cols * sizeof(T));
239         assert(data != nullptr);
240     }
241     else
242         data = (T*) nullptr;
243 }
244
245 template <typename T>
246 linag::Vector<T> &linag::Vector<T>::operator=(const linag::Vector<T>
247     &rhs){
248     if(this != &rhs){
249         if(dimension!=rhs.dim()) {
250             dimension = rhs.dim();
251             if(dim().rows*dim().cols > 0)
252             {
```

```
250         if(data == nullptr){
                data = (T *) malloc(rhs.dim().rows * rhs.dim().
cols * sizeof(T));
252         assert(data != nullptr);
        }
254         else {
                data = (T *) realloc(data, rhs.dim().rows * rhs.
dim().cols * sizeof(T));
256         assert(data != nullptr);
        }
258     }
        else
260         data = (T*) nullptr;
    }
262     //memcpy is a "dumb" function that only copies bytes
    std::memcpy(data, rhs.data, rhs.dim().rows * rhs.dim().cols *
sizeof(T));
264 }
    return *this;
266 }

268 template <typename T>
linag::Vector<T>::Vector(const linag::Vector<T> &rhs){
270     dimension = rhs.dim();
    if(dimension.rows*dimension.cols > 0)
272     {
        data = (T*) malloc(rhs.dim().rows * rhs.dim().cols * sizeof(T
));
274         assert(data != nullptr);
        //memcpy is a "dumb" function that only copies bytes
276         std::memcpy(data, rhs.data, rhs.dim().rows * rhs.dim().cols *
sizeof(T));
    }
    else
278         data = (T*) nullptr;
280 }

282 template <typename T>
linag::Vector<T>::~~Vector(){
284     if(data!= nullptr)
        free(data);
286 }

288 template <>
Eigen::VectorXd linag::Vector<double>::toEigen () const{
290     Eigen::VectorXd res = Eigen::VectorXd(length());

    for (int i = 0; i < length(); ++i) {
292         res(i)=at(i);
    }
```

```
294     }
295     return res;
296 }
297
298 #endif //AUFGABE1_VECTOR_H
```

Listing 7: vector.h

A.3. Vollbesetzte Matrix

```
1  #ifndef AUFGABE1_DENSEMATRIX_H
2  #define AUFGABE1_DENSEMATRIX_H
3
4  //eigen lib
5  #include "Eigen/Dense"
6  #include <iostream>
7  #include <cstring>
8  #include "size.h"
9  #include <thread>
10 #define THREAD_COUNT 8
11
12 namespace linag {
13     template <typename T> class SparseMatrix;
14     template <typename T> class Vector;
15
16     template <typename T>
17     class DenseMatrix{
18     private:
19         Size dimension;
20         T* data;
21     public:
22         DenseMatrix(int rows = 0, int cols = 0);
23         explicit DenseMatrix(linag::Size dimension);
24         ~DenseMatrix();
25
26         DenseMatrix(std::initializer_list<std::initializer_list<T>> init)
27             ;
28
29         DenseMatrix(const DenseMatrix<T> &rhs);
30         DenseMatrix<T> &operator=(const DenseMatrix<T> &rhs);
31
32         explicit DenseMatrix(const SparseMatrix<T>& rhs);
33         DenseMatrix<T> &operator=(const SparseMatrix<T> &rhs);
34
35     };
```

```
37
39     Eigen::MatrixXd toEigen() const;
41
42     const DenseMatrix<T> operator-() const;
43
44     const DenseMatrix<T> inverse() const;
45     const DenseMatrix<T> transpose() const;
46
47     T& at(int row,int col);
48     const T& at(int row,int col) const;
49     Vector<T> colToVector(int col);
50     Vector<T> rowToVector(int row);
51     const Size dim() const;
52
53     void zeros();
54     void id();
55     void diag(T value);
56     void diag(const DenseMatrix<T>& rhs);
57     void rand();
58     //upper tirangular matrix
59     void randLT();
60
61     void randDiag();
62
63     //rand sym,pos def
64     void randSPD(int notZeroPerLine);
65
66     char isSymmetric() const;
67
68     Vector<T> conjugateGradientSolver(linag::Vector<T> b, double tau,
69     int* count = nullptr,Vector<linag::Vector<double>*>* xs = nullptr
70     , linag::Vector<double>* rs = nullptr);
71
72     double cond();
73 };
74
75 template<typename T>
76 const DenseMatrix<T> operator+(const DenseMatrix<T>& x,const
77     DenseMatrix<T>& y);
78 template<typename T>
79 const DenseMatrix<T> operator-(const DenseMatrix<T>& x,const
80     DenseMatrix<T>& y);
81 template<typename T>
82 const DenseMatrix<T> operator*(const DenseMatrix<T>& x,const
83     DenseMatrix<T>& y);
84 template <typename T>
```

```
81 void mult_DenseMatrix_DenseMatrix(linag::DenseMatrix<T>& res, const
    DenseMatrix<T>& x, const DenseMatrix<T>& y, int idThread, int
    numThreads);

83

85 template<typename T>
    const DenseMatrix<T> operator*(const DenseMatrix<T>& x, const T y);
87 template<typename T>
    const DenseMatrix<T> operator*(const T x, const DenseMatrix<T>& y);
89 template<typename T>
    const Vector<T> operator*(const Vector<T>& x, const DenseMatrix<T>& y)
        ;
91 template<typename T>
    const Vector<T> operator*(const DenseMatrix<T>& x, const Vector<T>& y)
        ;
93

95 template<typename T>
    std::ostream& operator<<(std::ostream& output, const DenseMatrix<T>& x
        );
97

//template spezialication
99
101 template <>
    void linag::DenseMatrix<double >::rand(){
        for (int i = 0; i < dim().rows; ++i) {
103             for (int j = 0; j < dim().cols; ++j) {
                at(i,j) = (double)std::rand()/RAND_MAX;
105             }
        }
107    }

109    template<>
    void linag::DenseMatrix<double>::randLT() {
111        for (int i = 0; i < dim().cols; ++i) {
            for (int j = 0; j < i+1; ++j) {
113                at(i,j) = (double)std::rand()/RAND_MAX;
            }
            for (int j = i+1; j < dim().rows; ++j) {
115                at(i,j) = 0;
            }
117        }
    }
119

121    template <>
    void linag::DenseMatrix<double>::randDiag(){
123        int n = dim().cols<dim().rows?dim().cols:dim().rows;
        for (int i = 0; i < dim().rows; ++i) {
```



```
125         for (int j = 0; j < dim().cols; ++j) {
126             if(i == j)
127                 at(i,j) = (double)std::rand()/RAND_MAX;
128             else
129                 at(i,j) = 0;
130         }
131     }
132 }
133
134 template<>
135 void linag::DenseMatrix<double>::randSPD(int notZeroPerLine) {
136     assert(isSymmetric() && notZeroPerLine <= dim().cols &&
137 notZeroPerLine%2);
138
139     randLT();
140     double c = 50;
141     linag::DenseMatrix<double> diagM(dim());
142     diagM.randDiag();
143
144     (*this) = (*this) + transpose() + c * diagM;
145
146     for (int i = 0; i < dim().rows; ++i) {
147         for (int j = i+std::ceil((double)notZeroPerLine/2); j <
148 dim().cols; ++j) {
149             at(i,j) = 0;
150         }
151         for (int j = 0; j <= i-std::ceil((double)notZeroPerLine
152 /2); ++j) {
153             at(i,j) = 0;
154         }
155     }
156 }
157
158 template<typename T>
159 std::ostream& linag::operator<<(std::ostream& output, const linag::
160 DenseMatrix<T>& x){
161     for (int i = 0; i < x.dim().rows; ++i) {
162         for (int j = 0; j < x.dim().cols; ++j) {
163             output << x.at(i,j) << ",\t";
164         }
165         output << '\n';
166     }
167     return output;
168 }
169
170 template<typename T>
```

```
const linag::Vector<T> linag::operator*(const linag::DenseMatrix<T>&
x, const linag::Vector<T>& y){
171     assert(x.dim().cols == y.dim().cols*y.dim().rows);

173     linag::Vector<T> res(x.dim().rows);

175     for (int i = 0; i < res.dim().rows*res.dim().cols; ++i) {
        res.at(i) = 0;
177         for (int j = 0; j < x.dim().cols; ++j) {
            res.at(i) += x.at(i,j) * y.at(j);
179         }
        }
181     return res;
}

183
template<typename T>
185 const linag::Vector<T> linag::operator*(const linag::Vector<T>& x,
    const linag::DenseMatrix<T>& y){
    assert(y.dim().rows == x.dim().cols*x.dim().rows);

187     linag::Vector<T> res(y.dim().cols);

189     for (int i = 0; i < res.dim().rows*res.dim().cols; ++i) {
        res.at(i) = 0;
191         for (int j = 0; j < y.dim().rows; ++j) {
            res.at(i) += x.at(j) * y.at(i,j);
193         }
        }
195     return res;
197 }

199 template<typename T>
const linag::DenseMatrix<T> linag::operator*(const T x, const linag::
DenseMatrix<T>& y){
201     linag::DenseMatrix<T> res(y.dim());
    for (int i = 0; i < y.dim().rows; ++i) {
203         for (int j = 0; j < y.dim().cols; ++j) {
            res.at(i,j) = x*y.at(i,j);
205         }
        }
207     return res;
}

209
template<typename T>
211 const linag::DenseMatrix<T> linag::operator*(const linag::DenseMatrix
<T>& x, const T y){
    return y*x;
213 }
```

```
215 template<typename T>
const linag::DenseMatrix<T> linag::operator*(const linag::DenseMatrix
<T>& x,const linag::DenseMatrix<T>& y){
217     assert(x.dim().cols==y.dim().rows);

219     linag::DenseMatrix<T> res(x.dim().rows,y.dim().cols);

221     //multithreading
    linag::Vector<std::thread*> threads(THREAD_COUNT>res.dim().cols?
res.dim().cols:THREAD_COUNT);
223     for (int l = 0; l < threads.length(); ++l) {
        //create threads
225         threads.at(l) = new std::thread(linag::
mult_DenseMatrix_DenseMatrix<T>,std::ref(res),std::ref(x),std::ref
(y),l,threads.length());
    }
227     for (int l = 0; l < threads.length(); ++l) {
        threads.at(l)->join();
229     }
    //std::thread test(std::thread(linag::mult<T>,std::ref(res),std::
ref(x),std::ref(y),0,1));
231     //test.join();
    for (int l = 0; l < threads.length(); ++l) {
233         delete threads.at(l);
    }

235     return res;
237 }

239 template <typename T>
void linag::mult_DenseMatrix_DenseMatrix(linag::DenseMatrix<T>& res,
const linag::DenseMatrix<T>& x,const linag::DenseMatrix<T>& y,int
idThread,int numThreads){
241     for (int i = idThread; i < res.dim().cols; i+=numThreads) {
        for (int j = 0; j < x.dim().rows; ++j) {
243             res.at(j,i)=0;
            for (int k = 0; k < x.dim().cols; ++k) {
245                 res.at(j,i) += x.at(j,k) * y.at(k,i);
            }
247         }
    }
249 }

251 template<typename T>
253 const linag::DenseMatrix<T> linag::operator-(const linag::DenseMatrix
<T>& x,const linag::DenseMatrix<T>& y){
    assert(x.dim().cols==y.dim().cols && x.dim().rows==y.dim().rows);
255 }
```

```
linag::DenseMatrix<T> res(x.dim().rows,x.dim().cols);
257
    for (int i = 0; i < x.dim().rows; ++i) {
259        for (int j = 0; j < x.dim().cols; ++j) {
            res.at(i,j) = x.at(i,j)-y.at(i,j);
261        }
    }
263    return res;
}
265
template<typename T>
267 const linag::DenseMatrix<T> linag::operator+(const linag::DenseMatrix
    <T>& x,const linag::DenseMatrix<T>& y){
    return x-(-y);
269 }

template<typename T>
271 const linag::Size linag::DenseMatrix<T>::dim() const{
273     return dimension;
}
275
template<typename T>
277 linag::Vector<T> linag::DenseMatrix<T>::rowToVector(int row){
    assert(dim().row >= 0 && dim().cols < dim().rows);
279
    linag::Vector<T> res(dim().cols);
281
    for (int i = 0; i < res.dim(); ++i) {
283        res.at(i) = at(row,i);
    }
285    return res;
}
287
template<typename T>
289 linag::Vector<T> linag::DenseMatrix<T>::colToVector(int col){
    assert(col >= 0 && col < dim().cols);
291
    linag::Vector<T> res(dim().rows);
293
    for (int i = 0; i < res.dim(); ++i) {
295        res.at(i) = at(i,col);
    }
297    return res;
}
299
template<typename T>
301 const T& linag::DenseMatrix<T>::at(int row,int col) const{
    assert(row >= 0 && col >= 0 && row < dim().rows && col < dim().
        cols);
```

```
303     return data[row + col*dim().rows];
305 }
307 template<typename T>
308 T& linag::DenseMatrix<T>::at(int row, int col){
309     assert(row >= 0 && col >= 0 && row < dim().rows && col < dim().
310           cols);
311     return data[row + col*dim().rows];
312 }
313
314 template<typename T>
315 const linag::DenseMatrix<T> linag::DenseMatrix<T>::transpose() const{
316     linag::DenseMatrix<T> res(dim().cols, dim().rows);
317
318     for (int i = 0; i < dim().rows; ++i) {
319         for (int j = 0; j < dim().cols; ++j) {
320             res.at(j, i) = at(i, j);
321         }
322     }
323     return res;
324 }
325
326 template<typename T>
327 const linag::DenseMatrix<T> linag::DenseMatrix<T>::inverse() const{
328     assert(dim().rows == dim().cols);
329
330     linag::DenseMatrix<T> cpy(*this);
331     linag::DenseMatrix<T> res(dim());
332
333     for (int i = 0; i < dim().rows; ++i) {
334         for (int j = 0; j < dim().cols; ++j) {
335             if(i==j)
336                 res.at(i, j)=1;
337             else
338                 res.at(i, j)=0;
339         }
340     }
341
342     //gauss-jordan
343     for (int k = 0; k < dim().cols; ++k) {
344         //int k = 2;
345         T diagValue = cpy.at(k, k);
346         for (int i = 0; i < dim().cols; ++i) {
347             cpy.at(k, i) /= diagValue;
348             res.at(k, i) /= diagValue;
349         }
350         for (int i = 0; i < dim().rows; ++i) {
```

```
351         if(i==k)
352             continue;
353         T rowMult = cpy.at(i,k);
354         for (int j = 0; j < dim().cols; ++j) {
355             cpy.at(i,j) -= rowMult * cpy.at(k,j);
356             res.at(i,j) -= rowMult * res.at(k,j);
357         }
358     }
359 }
360 //std::cout << cpy << std::endl << res << std::endl;
361
362 return res;
363 }
364
365 template<typename T>
366 const linag::DenseMatrix<T> linag::DenseMatrix<T>::operator-() const{
367     return (T)-1* (*this);
368 }
369
370 template <>
371 Eigen::MatrixXd linag::DenseMatrix<double>::toEigen () const{
372     Eigen::MatrixXd res = Eigen::MatrixXd(dim().rows,dim().cols);
373
374     for (int i = 0; i < dim().rows; ++i) {
375         for (int j = 0; j < dim().cols; ++j) {
376             res(i,j)=at(i,j);
377         }
378     }
379     return res;
380 }
381
382 template <typename T>
383 linag::DenseMatrix<T> & linag::DenseMatrix<T>::operator=(const linag
384 ::DenseMatrix<T> &rhs){
385     if(this != &rhs){
386         if(dimension!=rhs.dim()) {
387             dimension = rhs.dim();
388             if(dim().rows*dim().cols > 0)
389             {
390                 if(!data)
391                 {
392                     data = (T*) malloc (rhs.dim().rows * rhs.dim().
393 cols * sizeof(T));
394                     assert(data != nullptr);
395                 }else {
396                     data = (T *) realloc(data, rhs.dim().rows * rhs.
397 dim().cols * sizeof(T));
398                     assert(data != nullptr);
399                 }
400             }
401         }
402     }
403 }
```

```
397         }
398         else
399             data = (T*) nullptr;
400     }
401     //memcpy is a "dumb" function that only copies bytes
402     std::memcpy(data, rhs.data, rhs.dim().rows * rhs.dim().cols *
403     sizeof(T));
404 }
405 return *this;
406 }
407
408 template <typename T>
409 linag::DenseMatrix<T>::DenseMatrix(const DenseMatrix<T> &rhs):
410     dimension(rhs.dim()){
411     if(dimension.rows*dimension.cols > 0)
412     {
413         data = (T*) malloc(rhs.dim().rows * rhs.dim().cols *
414         sizeof(T));
415         assert(data != nullptr);
416         //memcpy is a "dumb" function that only copies bytes
417         std::memcpy(data, rhs.data, rhs.dim().rows * rhs.dim().cols *
418         sizeof(T));
419     }
420     else
421         data = (T*) nullptr;
422 }
423
424 template <typename T>
425 linag::DenseMatrix<T>::DenseMatrix(std::initializer_list<std::
426     initializer_list<T>> init){
427     dimension.rows = init.size();
428     dimension.cols = init.begin()->size();
429     //check if all rows have same length
430     for(auto row : init){
431         assert(dim().cols == row.size());
432     }
433
434     if(dim().rows*dim().cols > 0)
435     {
436         data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
437         assert(data != nullptr);
438         //copy
439         int i=0, j;
440         for(auto row: init) {
441             j=0;
442             for (auto item: row) {
443                 at(i, j) = item;
444                 ++j;
445             }
446             ++i;
447         }
448     }
449 }
```

```
441         ++i;
442     }
443 }
444 else
445     data = (T*) nullptr;
446 }
447
448 template <typename T>
449 linag::DenseMatrix<T>::~DenseMatrix(){
450     if(data!= nullptr)
451         free(data);
452 }
453
454 template <typename T>
455 linag::DenseMatrix<T>::DenseMatrix(linag::Size dimension):dimension(
456     dimension){
457     if(dim().rows*dim().cols > 0)
458     {
459         data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
460         assert(data != nullptr);
461     }
462     else
463         data = (T*) nullptr;
464 }
465
466 template <typename T>
467 linag::DenseMatrix<T>::DenseMatrix(int rows,int cols){
468     dimension.rows = rows;
469     dimension.cols = cols;
470     if(rows*cols > 0)
471     {
472         data = (T*) malloc(rows * cols * sizeof(T));
473         assert(data != nullptr);
474     }
475     else
476         data = (T*) nullptr;
477 }
478
479 template <typename T>
480 void linag::DenseMatrix<T>::zeros(){
481     for (int i = 0; i < dim().rows; ++i) {
482         for (int j = 0; j < dim().cols; ++j) {
483             at(i,j) = (T)0;
484         }
485     }
486 }
487
488 template <typename T>
489 void linag::DenseMatrix<T>::id(){
```



```
489     for (int i = 0; i < dim().rows; ++i) {
490         for (int j = 0; j < dim().cols; ++j) {
491             if(i==j)
492                 at(i,j) = 1;
493             else
494                 at(i,j) = (T)0;
495         }
496     }
497 }

499 template <typename T>
500 linag::Vector<T> linag::DenseMatrix<T>::conjugateGradientSolver(linag
::Vector<T> b, double tau, int* count, linag::Vector<linag::Vector<
double>*>* xs, linag::Vector<double>* rs){
501     assert(tau>0 && dim().rows == b.length());
502     if(xs)
503         assert(xs->length() == dim().rows); //exact result after n
iterations
504     if(rs)
505         assert(rs->length() == dim().rows); //exact result after n
iterations

507     linag::Vector<T> r1(dim().rows);
508     linag::Vector<T> r2(dim().rows);
509     linag::Vector<T> d(dim().rows);
510     linag::Vector<T> x(dim().rows);
511     linag::Vector<T> z(dim().rows);
512     x.rand();
513     T alpha;
514     T betta;
515     unsigned long t = 0;
516     r1 = b - (*this)*x;
517     d = r1;
518     if(xs) {
519         for (int i = 1; i < xs->length(); ++i) {
520             xs->at(i) = nullptr;
521         }
522         xs->at(0) = new linag::Vector<double>(x);
523     }
524     if(rs) {
525         for (int i = 1; i < rs->length(); ++i) {
526             rs->at(i) = 0;
527         }
528         rs->at(0) = r1.l2norm();
529     }
530     do{
531         z = (*this)*d;
532         alpha = (r1*r1)/(d*z);
533         x = x + alpha*d;
```

```
535     r2 = r1 - alpha*z;
    betta = (r2*r2)/(r1*r1);
    d = r2 + betta*d;
537
    r1=r2;
539     if(xs && t < xs->length())
        xs->at(t) = new linag::Vector<double>(x);
541     if(rs && t < rs->length())
        rs->at(t) = r2.l2norm();
543     ++t;
    }while (r2.l2norm()>tau);
545     if(count)
        *count = t;
547     return x;
    }
549
    template <typename T>
551     char linag::DenseMatrix<T>::isSymmetric() const{
        return dim().cols == dim().rows?1:0;
553     }

555     template <typename T>
    void linag::DenseMatrix<T>::diag(T value){
557         int n = dim().cols<dim().rows?dim().cols:dim().rows;
        for (int i = 0; i < n; ++i) {
559             at(i,i) = value;
        }
561     }

563     template <typename T>
    void linag::DenseMatrix<T>::diag(const linag::DenseMatrix<T>& rhs){
565         assert(dim() == rhs.dim());
        zeros();
567         for (int i = 0; i < rhs.dim().rows; ++i) {
            for (int j = 0; j < rhs.dim().cols; ++j) {
569                 if(i == j)
                    at(i,j) = rhs.at(i,j);
571             }
        }
573     }

575     template <typename T>
577     linag::DenseMatrix<T>::DenseMatrix(const linag::SparseMatrix<T>& rhs)
        :dimension(rhs.dim()){
        if(dim().rows*dim().cols > 0)
579     {
        data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
581        assert(data != nullptr);
```

```
583     zeros();
584     for (int i = 0; i < dim().rows; ++i) {
585         for (int j = rhs.getI().at(i); j < rhs.getI().at(i+1); ++
j) {
586             at(i, rhs.getJ().at(j))=rhs.getV().at(j);
587         }
588     }
589 }
590 else
591     data = (T*) nullptr;
592 }
593
594 template <typename T>
595 linag::DenseMatrix<T> &linag::DenseMatrix<T>::operator=(const linag::
SparseMatrix<T> &rhs){
596     dimension = rhs.dim();
597     if (rhs.dim().rows * rhs.dim().cols > 0) {
598         if (!data) {
599             data = (T *) malloc(rhs.dim().rows * rhs.dim().
cols * sizeof(T));
600             assert(data != nullptr);
601         } else {
602             data = (T *) realloc(data, rhs.dim().rows * rhs.
dim().cols * sizeof(T));
603             assert(data != nullptr);
604         }
605     }else
606         data = (T *) nullptr;
607
608     zeros();
609     for (int i = 0; i < dim().rows; ++i) {
610         for (int j = rhs.getI().at(i); j < rhs.getI().at(i + 1);
++j) {
611             at(i, rhs.getJ().at(j)) = rhs.getV().at(j);
612         }
613     }
614     return *this;
615 }
616
617 template <typename T>
618 double linag::DenseMatrix<T>::cond(){
619     Eigen::VectorXcd eigenvalues = toEigen().eigenvalues();
620
621     double min=std::fabs(eigenvalues(0).real()),max = std::fabs(
eigenvalues(0).real());
622     for (int i = 1; i < eigenvalues.size(); ++i) {
623         if(std::fabs(eigenvalues(i).real()) > max)
```

```
625         max = std::fabs(eigenvalues(i).real());
627         if(std::fabs(eigenvalues(i).real()) < min)
629             min = std::fabs(eigenvalues(i).real());
631     }
633     return max/min;
635 }
#endif //AUGABE1_DENSEMATRIX_H
```

Listing 8: densematrix.h

A.4. Dünnbesetzte Matrix

```
#ifndef AUGABE1_SPARSEMATRIX_H
2 #define AUGABE1_SPARSEMATRIX_H

4 #include "cmath"
5 #include "iostream"
6 #include "vector.h"

8 #define THREAD_COUNT 8

10 namespace linag {
12     //say class exists without defining it
13     template <typename T> class DenseMatrix;
14
16     template <typename T>
17     class SparseMatrix {
18     private:
19         Vector<T> v;
20         Vector<int> I;
21         Vector<int> J;
22
23         Size dimension;
24
25     public:
26         SparseMatrix()= default;
27         ~SparseMatrix() = default;
28
29         explicit SparseMatrix(const DenseMatrix<T>& rhs);
30         SparseMatrix<T> & operator=(const DenseMatrix<T>& rhs);
```

```
32     SparseMatrix(const SparseMatrix<T>& rhs);
33     SparseMatrix<T> &operator=(const SparseMatrix<T> &rhs);
34
35     const SparseMatrix<T> operator-() const;
36
37     const Size dim() const;
38
39     char isSymmetric() const;
40
41     const Vector<T>& getV() const{ return v;};
42     const Vector<int>& getI() const{ return I;};
43     const Vector<int>& getJ() const{ return J;};
44
45     Vector<T> conjugateGradientSolver(linag::Vector<T> b, double
tau, int* count = nullptr, linag::Vector<linag::Vector<double>*>*
xs = nullptr, linag::Vector<double>* rs = nullptr);
46     Vector<T> preCondConjugateGradientSolver(const linag::
SparseMatrix<T>& P, const linag::Vector<T> b, double tau, int*
count = nullptr, linag::Vector<linag::Vector<double>*>* xs =
nullptr, linag::Vector<double>* rs = nullptr);
48
49 };
50
51 template<typename T>
52 const SparseMatrix<T> operator*(const SparseMatrix<T>& x, const T
y);
53 template<typename T>
54 const SparseMatrix<T> operator*(const T x, const SparseMatrix<T>&
y);
55 template<typename T>
56 const Vector<T> operator*(const Vector<T>& x, const SparseMatrix<T
>& y);
57 template<typename T>
58 const Vector<T> operator*(const SparseMatrix<T>& x, const Vector<T
>& y);
59 //template <typename T>
60 //void mult(linag::Vector<T> &res, const linag::SparseMatrix<T>&
x, const linag::Vector<T>& y, int idThread, int numThreads);
61
62 }
63
64 template<typename T>
65 const linag::SparseMatrix<T> linag::operator*(const linag::
SparseMatrix<T>& x, const T y){
66     linag::SparseMatrix<T> res(x);
67     for (int i = 0; i < res.v.length(); ++i) {
68         res.v.at(i) *= y;
69     }
```

```
70     }
71     return res;
72 }
73
74 template<typename T>
75 const linag::SparseMatrix<T> linag::operator*(const T x,const linag::
76     SparseMatrix<T>& y){
77     return y*x;
78 }
79
80 template<typename T>
81 const linag::Vector<T> linag::operator*(const linag::Vector<T>& x,
82     const linag::SparseMatrix<T>& y){
83     std::cout << "undefined" << std::endl;
84 }
85
86 template<typename T>
87 const linag::Vector<T> linag::operator*(const linag::SparseMatrix<T>&
88     x,const linag::Vector<T>& y){
89     assert(x.dim().cols == y.length());
90     linag::Vector<T> res(y.length());
91     res.zeros();
92     for (int i = 0; i < res.length(); ++i) {
93         for (int j = x.getI().at(i); j < x.getI().at(i + 1); ++j) {
94             res.at(i) += y.at(x.getJ().at(j)) * x.getV().at(j);
95         }
96     }
97     return res;
98 }
99
100 //template <typename T>
101 //void linag::mult(linag::Vector<T> &res, const linag::SparseMatrix<T
102 //    & x,const linag::Vector<T>& y,int idThread,int numThreads){
103 //}
104
105 template <typename T>
106 linag::Vector<T> linag::SparseMatrix<T>::conjugateGradientSolver(
107     linag::Vector<T> b, double tau, int* count,linag::Vector<linag::
108     Vector<double>*>* xs, linag::Vector<double>*> rs){
109     assert(tau>0 && dim().rows == b.length());
110     if(xs)
111         assert(xs->length() == dim().rows);//exact result after n
112         iterations
113     if(rs)
114         assert(rs->length() == dim().rows);//exact result after n
115         iterations
116
117     linag::Vector<T> r1(dim().rows);
```

```
112 linag::Vector<T> r2(dim().rows);
113 linag::Vector<T> d(dim().rows);
114 linag::Vector<T> x(dim().rows);
115 linag::Vector<T> z(dim().rows);
116 x.rand();
117 T alpha;
118 T betta;
119 unsigned long t = 0;
120 r1 = b - (*this)*x;
121 d = r1;
122 if(count)
123     *count = 0;
124 if(xs) {
125     for (int i = 1; i < xs->length(); ++i) {
126         xs->at(i) = nullptr;
127     }
128     xs->at(0) = new linag::Vector<double>(x);
129 }
130 if(rs) {
131     for (int i = 1; i < rs->length(); ++i) {
132         rs->at(i) = 0;
133     }
134     rs->at(0) = r1.l2norm();
135 }
136 do{
137     z = (*this)*d;
138     alpha = (r1*r1)/(d*z);
139     x = x + alpha*d;
140     r2 = r1 - alpha*z;
141     betta = (r2*r2)/(r1*r1);
142     d = r2 + betta*d;
143
144     r1=r2;
145     if(count)
146         ++*count;
147     if(xs && t < xs->length())
148         xs->at(t) = new linag::Vector<double>(x);
149     if(rs && t < rs->length())
150         rs->at(t) = r2.l2norm();
151     ++t;
152 }while (r2.l2norm()>tau);
153
154 return x;
155 }
156
157
158 template<typename T>
```

```
linag::Vector<T> linag::SparseMatrix<T>::
preCondConjugateGradientSolver(const linag::SparseMatrix<T>& Pinv,
    const linag::Vector<T> b, double tau, int* count, linag::Vector<
160 linag::Vector<double>*> xs, linag::Vector<double>* rs){
    assert(tau>0 && dim().rows == b.length());
    if(xs)
162     assert(xs->length() == dim().rows); //exact result after n
iterations
    if(rs)
164     assert(rs->length() == dim().rows); //exact result after n
iterations

    linag::Vector<T> r1(dim().rows);
    linag::Vector<T> r2(dim().rows);
168    linag::Vector<T> d(dim().rows);
    linag::Vector<T> x(dim().rows);
170    linag::Vector<T> z(dim().rows);
    linag::Vector<T> z1(dim().rows);
172    linag::Vector<T> z2(dim().rows);
    x.rand();
174    T alpha;
    T betta;
176    unsigned long t = 0;
    r1 = b - (*this)*x;
178    z1 = Pinv * r1;
    d = z1;
180    if(count)
        *count = 0;
182    if(xs) {
        for (int i = 1; i < xs->length(); ++i) {
184            xs->at(i) = nullptr;
        }
        xs->at(0) = new linag::Vector<double>(x);
    }
188    if(rs) {
        for (int i = 1; i < rs->length(); ++i) {
190            rs->at(i) = 0;
        }
        rs->at(0) = r1.l2norm();
192    }
194    do{
        z = (*this)*d;
196        alpha = (r1*z1)/(d*z);
        x = x + alpha*d;
198        r2 = r1 - alpha*z;
        z2 = Pinv*r2;
200        betta = (z2*r2)/(z1*r1);
        d = z2 + betta*d;
202    }
```



```
204     r1=r2;
205     z1=z2;
206     if(count)
207         ++*count;
208     if(xs && t < xs->length())
209         xs->at(t) = new linag::Vector<double>(x);
210     if(rs && t < rs->length())
211         rs->at(t) = r2.l2norm();
212     ++t;
213 }while (r2.l2norm()>tau);
214
215 return x;
216 }
217
218 template <typename T>
219 char linag::SparseMatrix<T>::isSymmetric() const{
220     return dim().cols == dim().rows?1:0;
221 }
222
223 template <typename T>
224 const linag::SparseMatrix<T> linag::SparseMatrix<T>::operator-()
225 const{
226     linag::SparseMatrix<T> res(*this);
227     res.v = res.v *(-1);
228     return res;
229 }
230
231 template<typename T>
232 const linag::Size linag::SparseMatrix<T>::dim() const{
233     return dimension;
234 }
235
236 template <typename T>
237 linag::SparseMatrix<T>::SparseMatrix(const linag::SparseMatrix<T>&
238 rhs){
239     dimension = rhs.dim();
240     v = rhs.v;
241     I = rhs.I;
242     J = rhs.J;
243 }
244
245 template <typename T>
246 linag::SparseMatrix<T> &linag::SparseMatrix<T>::operator=(const linag
247 ::SparseMatrix<T> &rhs){
248     if(this != &rhs) {
249         dimension = rhs.dim();
250         v = rhs.v;
251         I = rhs.I;
```

```
        J = rhs.J;
250    }
        return *this;
252 }

254 template <typename T>
linag::SparseMatrix<T>::SparseMatrix(const linag::DenseMatrix<T>& rhs
    ):
256 I(0),J(0),v(0),dimension(rhs.dim()){
    //calculate array size
258    int vc = 0;
    int Ic = rhs.dim().rows+1;
260    for (int i = 0; i < rhs.dim().rows; ++i) { //rows
        for (int j = 0; j < rhs.dim().cols; ++j) { //cols
262            if(std::fabs(rhs.at(i,j)) > 10e-10){
                ++vc;
264            }
        }
266    }
    //set array size
268    I = linag::Vector<int>(Ic);
    J = linag::Vector<int>(vc);
270    v = linag::Vector<T>(vc);

    //convert dense matrix to sparse matrix
    vc=0;
274    Ic=-1;
    int Jc=0;
276    for (int i = 0; i < rhs.dim().rows; ++i) { //rows
        for (int j = 0; j < rhs.dim().cols; ++j) { //cols
278            if(std::fabs(rhs.at(i,j)) > 10e-10){
                if(Ic != i){
280                    I.at(++Ic) = vc;
                }
                v.at(vc++) = rhs.at(i,j);
                J.at(Jc++) = j;
284            }
        }
286    }
    I.at(++Ic) = vc;
288 }

290 template <typename T>
linag::SparseMatrix<T> & linag::SparseMatrix<T>::operator=(const
    linag::DenseMatrix<T>& rhs){
292    //calculate array size
    int vc=0;
294    int Ic =rhs.dim().rows+1;
    for (int i = 0; i < rhs.dim().rows; ++i) { //rows
```

```
296         for (int j = 0; j < rhs.dim().cols; ++j) { //cols
297             if(std::fabs(rhs.at(i,j)) > 10e-10){
298                 ++vc;
299             }
300         }
301     }
302     //rows/cols
303     dimension.rows=rhs.dim().rows;
304     dimension.cols=rhs.dim().cols;
305     //set array size
306     I = linag::Vector<int>(Ic);
307     J = linag::Vector<int>(vc);
308     v = linag::Vector<T>(vc);
309
310     //convert dense matrix to sparse matrix
311     vc=0;
312     Ic=0;
313     int Jc=0;
314     for (int i = 0; i < rhs.dim().rows; ++i) { //rows
315         for (int j = 0; j < rhs.dim().cols; ++j) { //cols
316             if(std::fabs(rhs.at(i,j)) > 10e-10){
317                 if(!Ic || Ic != i){
318                     I.at(Ic++) = vc;
319                 }
320                 v.at(vc++) = rhs.at(i,j);
321                 J.at(Jc++) = j;
322             }
323         }
324     }
325 }
326
327 #endif //AUFGABE1_SPARSEMATRIX_H
```

Listing 9: sparsematrix.h

Literatur

Nannen, Lothar (2019). *Numerische Mathematik A*. URL: <https://tiss.tuwien.ac.at/education/course/documents.xhtml?dswid=3351&dsrid=39&courseNr=101313&semester=2019W#> (besucht am 03.12.2019) (siehe S. 6 f.).

Listings

1	Erstellen einer symmetrisch positiv definiten Zufallsmatrix mit einer fixen Anzahl an Einträgen ungleich 0 pro Zeile in C++	4
2	Implementierung des CG-Verfahrens in C++	8
3	Speicherung einer Matrix im compressed sparse row Format	9
4	compressed sparse row Format zu vollbesetzter Matrix	10
5	Überladen der Matrix-Vektor Multiplikation für dünnbesetzte Matrizen	10
6	size.h	15
7	vector.h	15
8	densematrix.h	22
9	sparsematrix.h	36

Abbildungsverzeichnis

1	Benötigte Zeit in μs um Gleichungssysteme mit verschieden großen dicht besetzten $n \times n$ -Koeffizientenmatrizen mittels Cholesky-Zerlegung zu lösen	5
2	Benötigte Zeit in μs für verschieden dicht besetzte $n \times n$ -Matrizen im <i>compressed sparse row</i> -Format	11
3	Residuum nach Iterationsschritt t für vorkonditioniertes CG- und CG-Verfahren	13
4	Residuum nach Iterationsschritt t für vorkonditioniertes CG- und CG-Verfahren	14
5	Iterationszahlen für verschieden große Koeffizientenmatrizen der Form $A = B + B^T + c \cdot \text{diag}(b)$	14