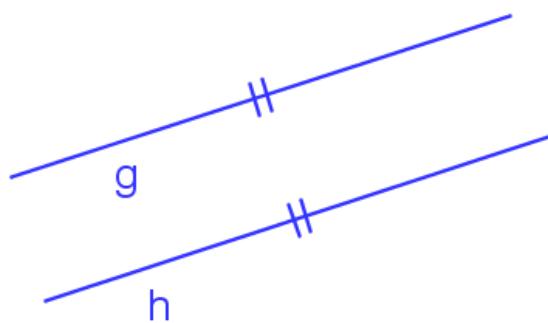


# CG-Verfahren für dünnbesetzte Matrizen

*Rafael Dorigo  
Sebastian Hirnschall*

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



## Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>3</b>
<b>2 Motivation</b>	<b>4</b>
<b>3 CG-Verfahren (Verfahren der konjugierten Gradienten)</b>	<b>6</b>
3.1 Dünnbesetzte Matrizen . . . . .	9
3.2 Vorkonditionierung . . . . .	12
<b>A Verwendete Klassen</b>	<b>15</b>
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 postiv 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.

## 2 MOTIVATION

---

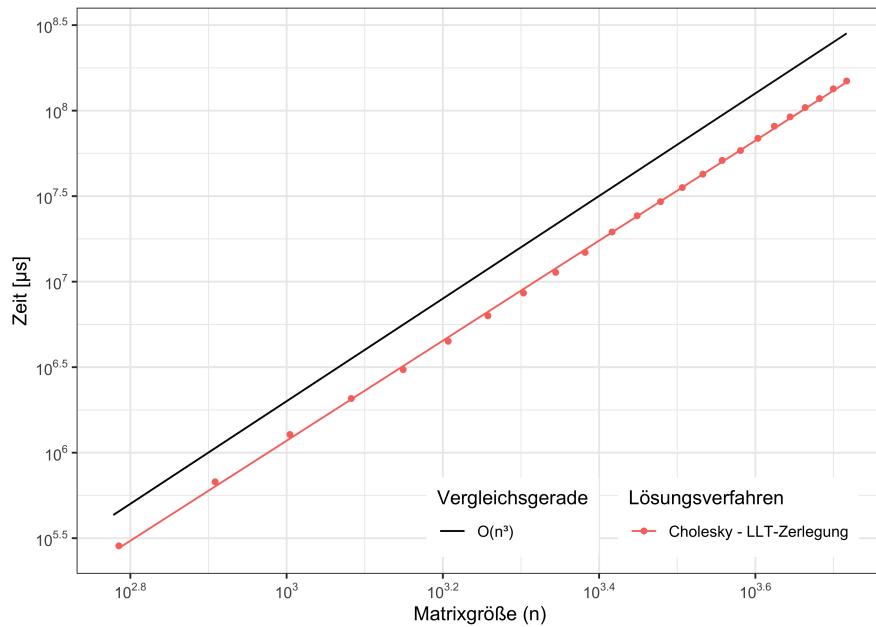


Abbildung 1: Benötigte Zeit in  $\mu s$  um Gleichungssysteme mit verschiedenen großen dicht besetzten  $n \times n$ -Koeffizientenmatrizen mittels Cholesky-Zerlegung zu lösen

### 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-

### 3 CG-VERFAHREN (VERFAHREN DER KONJUGIERTEN GRADIENTEN)

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  $\beta_t$  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 Residuen 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

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

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  $\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1} = 0$ .

### 3 CG-VERFAHREN (VERFAHREN DER KONJUGIERTEN GRADIENTEN)

```
1 template <typename T>
2 linag::Vector<T> linag::DenseMatrix<T>::conjugateGradientSolver(linag
3   ::Vector<T> b, double tau, int* count, linag::Vector<linag::Vector<
4     double>*>* xs, linag::Vector<double>*> rs){
5   assert(tau>0 && dim().rows == b.length());
6   if(xs)
7     assert(xs->length() == dim().rows); //exact result after n
8   iterations
9   if(rs)
10    assert(rs->length() == dim().rows); //exact result after n
11   iterations
12
13   linag::Vector<T> r1(dim().rows);
14   linag::Vector<T> r2(dim().rows);
15   linag::Vector<T> d(dim().rows);
16   linag::Vector<T> x(dim().rows);
17   linag::Vector<T> z(dim().rows);
18   x.rand();
19   T alpha;
20   T betta;
21   unsigned long t = 0;
22   r1 = b - (*this)*x;
23   d = r1;
24   if(xs) {
25     for (int i = 1; i < xs->length(); ++i) {
26       xs->at(i) = nullptr;
27     }
28     xs->at(0) = new linag::Vector<double>(x);
29   }
30   if(rs) {
31     for (int i = 1; i < rs->length(); ++i) {
32       rs->at(i) = 0;
33     }
34     rs->at(0) = r1.l2norm();
35   }
36   do{
37     z = (*this)*d;
38     alpha = (r1*r1)/(d*z);
39     x = x + alpha*d;
40     r2 = r1 - alpha*z;
41     betta = (r2*r2)/(r1*r1);
42     d = r2 + betta*d;
43
44     r1=r2;
45     if(xs && t < xs->length())
46       xs->at(t) = new linag::Vector<double>(x);
47     if(rs && t < rs->length())
48       rs->at(t) = r2.l2norm();
49   }
```

```

45     ++t;
46 }while (r2.l2norm()>tau);
47 if(count)
48     *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>
linag::SparseMatrix<T>::SparseMatrix(const linag::DenseMatrix<T>& rhs
):
I(0),J(0),v(0),dimension(rhs.dim()){
    //calculate array size
    int vc = 0;
    int Ic = rhs.dim().rows+1;
    for (int i = 0; i < rhs.dim().rows; ++i) { //rows
        for (int j = 0; j < rhs.dim().cols; ++j) { //cols
            if(std::fabs(rhs.at(i,j)) > 10e-10){
                ++vc;
            }
        }
    }
    //set array size
    I = linag::Vector<int>(Ic);
    J = linag::Vector<int>(vc);
    v = linag::Vector<T>(vc);
}

//convert dense matrix to sparse matrix

```

### 3 CG-VERFAHREN (VERFAHREN DER KONJUGIERTEN GRADIENTEN)

```
20     vc=0;
21     Ic=-1;
22     int Jc=0;
23     for (int i = 0; i < rhs.dim().rows; ++i) { //rows
24         for (int j = 0; j < rhs.dim().cols; ++j) { //cols
25             if(std::fabs(rhs.at(i,j)) > 10e-10){
26                 if(Ic != i){
27                     I.at(++Ic) = vc;
28                 }
29                 v.at(vc++) = rhs.at(i,j);
30                 J.at(Jc++) = j;
31             }
32         }
33     }
34     I.at(++Ic) = vc;
35 }
```

Listing 3: Speicherung einer Matrix im compressed sparse row Format

```
template <typename T>
2 linag::DenseMatrix<T>::DenseMatrix(const linag::SparseMatrix<T>& rhs)
3     :dimension(rhs.dim()){
4         if(dim().rows*dim().cols > 0)
5         {
6             data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
7             assert(data != nullptr);
8
8             zeros();
9             for (int i = 0; i < dim().rows; ++i) {
10                 for (int j = rhs.getI().at(i); j < rhs.getI().at(i+1); ++
11                     j) {
12                     at(i,rhs.getJ().at(j))=rhs.getV().at(j);
13                 }
14             }
15             else
16                 data = (T*) nullptr;
17 }
```

Listing 4: compressed sparse 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>&
3     x,const linag::Vector<T>& y){
4     assert(x.dim().cols == y.length());
5     linag::Vector<T> res(y.length());
```

### 3 CG-VERFAHREN (VERFAHREN DER KONJUGIERTEN GRADIENTEN)

```

6     res.zeros();
7     for (int i = 0; i < res.length(); ++i) {
8         for (int j = x.getI().at(i); j < x.getI().at(i + 1); ++j) {
9             res.at(i) += y.at(x.getJ().at(j)) * x.getV().at(j);
10        }
11    }
12    return res;
}

```

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.

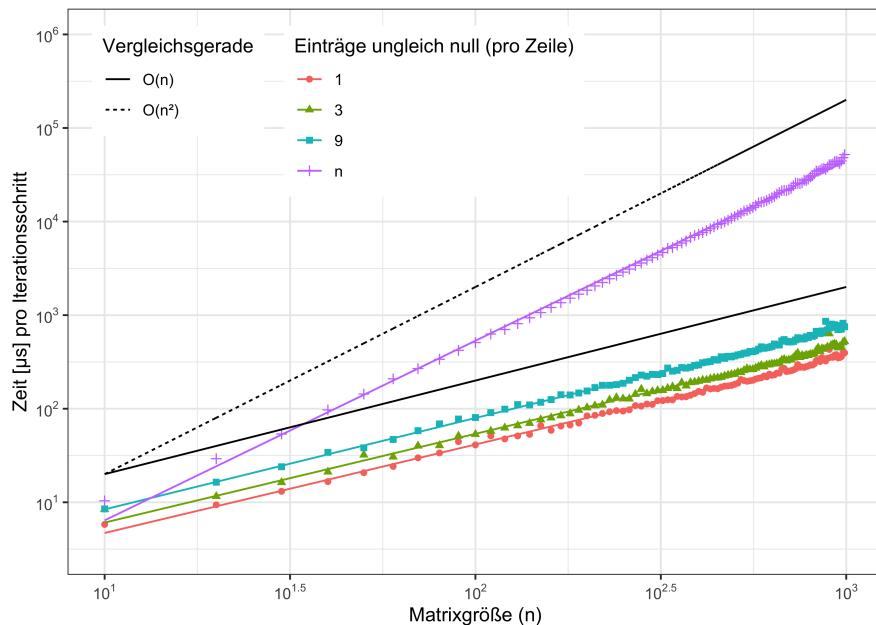


Abbildung 2: Benötigte Zeit in  $\mu s$  für verschieden dicht besetzte  $n \times n$ -Matrizen im *compressed sparse row*-Format

### 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)

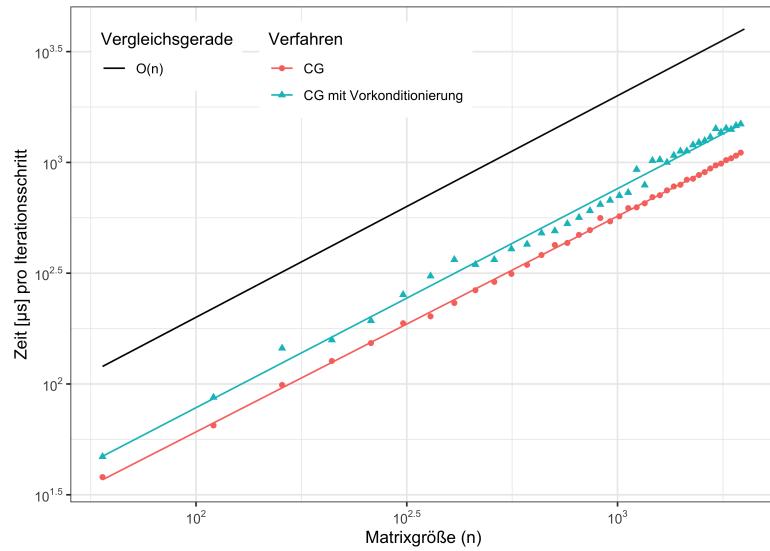


Abbildung 3: Residuum nach Iterationsschritt  $t$  für vorkonditioniertes CG- und CG-Verfahren

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)

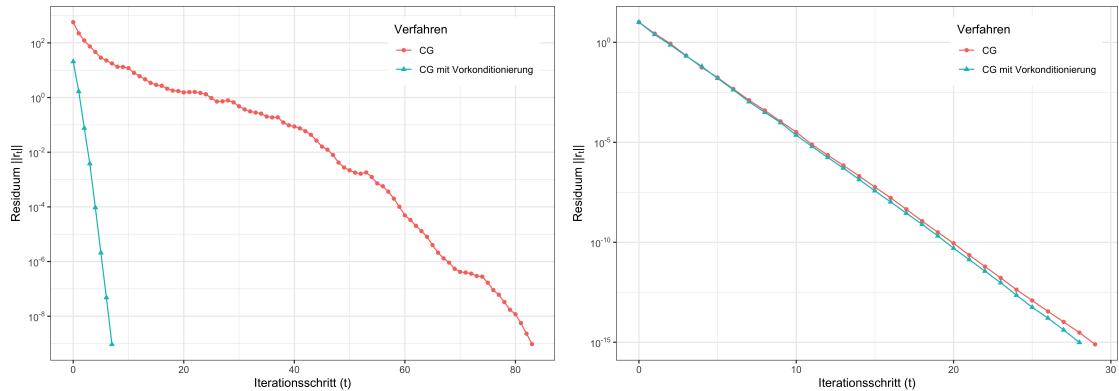


Abbildung 4: Residuum nach Iterationsschritt  $t$  für vorkonditioniertes CG- und CG-Verfahren

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.

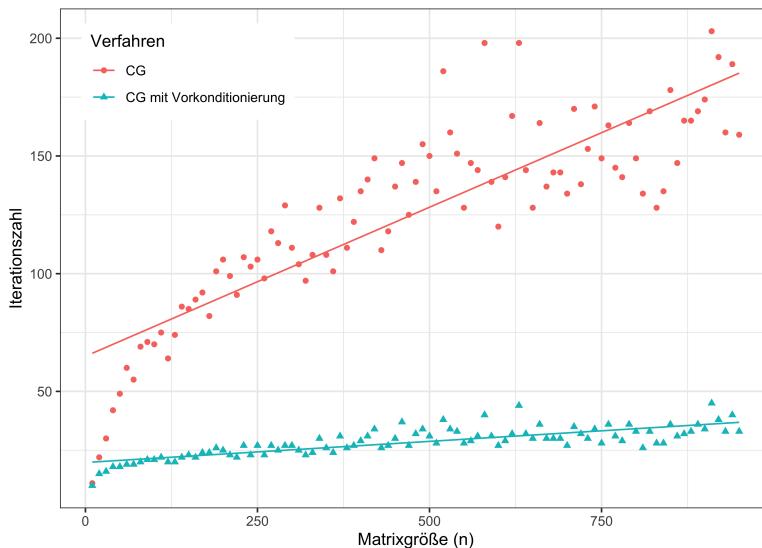


Abbildung 5: Iterationszahlen für verschiedene große Koeffizientenmatrizen der Form  $A = B + B^T + c \cdot \text{diag}(b)$

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

## 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     bool operator==(const Size& lhs, const Size& rhs)
12     {
13         return lhs.rows == rhs.rows && lhs.cols == rhs.cols;
14     }
15
16     bool operator!=(const Size& lhs, const Size& rhs)
17     {
18         return !(lhs == rhs);
19     }
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 {
```

## A VERWENDETE KLASSEN

---

```
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);
```

## A VERWENDETE KLASSEN

---

```
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<T>& y){
83     linag::Vector<T> res(y.dim());
84     for (int i = 0; i < y.dim().rows * y.dim().cols; ++i) {
85         res.at(i) = x*y.at(i);
86     }
87     return res;
88 }
89
90 template<typename T>
91 const linag::Vector<T> linag::operator*(const linag::Vector<T>& x,
92     const T y){
93     linag::Vector<T> res(x.dim());
94     for (int i = 0; i < x.dim().rows * x.dim().cols; ++i) {
95         res.at(i) = x.at(i) * y;
96     }
97     return res;
98 }
99
100 template<typename T>
101 T linag::operator*(const linag::Vector<T>& x, const linag::Vector<T>&
102     y){
103     assert(x.dim().rows*x.dim().cols == y.dim().rows*y.dim().cols);
104     T res = (T)0;
105     for (int i = 0; i < x.dim().rows*x.dim().cols; ++i) {
106         res+= y.at(i) *x.at(i);
107     }
108     return res;
```

## A VERWENDETE KLASSEN

---

```
108 }
110
111 template<typename T>
112 const linag::Vector<T> operator/(const linag::Vector<T>& x, const T y)
113 {
114     linag::Vector<T> res(x);
115     for (int i = 0; i < res.dim().cols*res.dim().rows; ++i) {
116         res.at(i)/=y;
117     }
118     return res;
119 }

120
121 template<typename T>
122 const linag::Vector<T> linag::operator-(const linag::Vector<T>& x,
123 const linag::Vector<T>& y){
124     return x + (-y);
125 }

126 template<typename T>
127 const linag::Vector<T> linag::operator+(const linag::Vector<T>& x,
128 const linag::Vector<T>& y){
129     assert(x.dim()== y.dim());
130     linag::Vector<T> res(x.dim());
131     for (int i = 0; i < x.dim().rows*x.dim().cols; ++i) {
132         res.at(i) = x.at(i) + y.at(i);
133     }
134     return res;
135 }

136
137 template <typename T>
138 void linag::Vector<T>::zeros(){
139     for (int i = 0; i < length(); ++i) {
140         at(i) = 0;
141     }
142 }
143 template <typename T>
144 void linag::Vector<T>::ones(){
145     for (int i = 0; i < length(); ++i) {
146         at(i) = 1;
147     }
148 }

149 template <typename T>
150 void linag::Vector<T>::rand(){
151     for (int i = 0; i < dim().rows*dim().cols; ++i) {
152         at(i) = (T)std::rand()/RAND_MAX;
153     }
154 }
```

## A VERWENDETE KLASSEN

---

```
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)
```

## A VERWENDETE KLASSEN

---

```
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){
251                 data = (T *) malloc(rhs.dim().rows * rhs.dim().
252                                     cols * sizeof(T));
253                 assert(data != nullptr);
254             }
255             else {
256                 data = (T *) realloc(data, rhs.dim().rows * rhs.
257                                     dim().cols * sizeof(T));
258                 assert(data != nullptr);
259             }
260         }
261         else
262             data = (T*) nullptr;
263     }
264     //memcpy is a "dumb" function that only copies bytes
265     std::memcpy(data,rhs.data,rhs.dim().rows * rhs.dim().cols *
266     sizeof(T));
267 }
268 return *this;
269 }

270 template <typename T>
271 linag::Vector<T>::Vector(const linag::Vector<T> &rhs){
272     dimension = rhs.dim();
273     if(dimension.rows*dimension.cols > 0)
274     {
275         data = (T*) malloc(rhs.dim().rows * rhs.dim().cols * sizeof(T));
276         assert(data != nullptr);
277         //memcpy is a "dumb" function that only copies bytes
278         std::memcpy(data,rhs.data,rhs.dim().rows * rhs.dim().cols *
279         sizeof(T));
280     }
281     else
282         data = (T*) nullptr;
283 }

284 template <typename T>
285 linag::Vector<T>::~Vector(){
286     if(data!= nullptr)
287         free(data);
288 }

289 template <>
290 Eigen::VectorXd linag::Vector<double>::toEigen () const{
291     Eigen::VectorXd res = Eigen::VectorXd(length());
292
293     for (int i = 0; i < length(); ++i) {
294         res(i)=at(i);
295     }
296 }
```

```
294     }
295     return res;
296 }
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);
```

## A VERWENDETE KLASSEN

---

```
37
39     Eigen::MatrixXd toEigen() const;
41
43     const DenseMatrix<T> operator-() const;
45
47     T& at(int row,int col);
49     const T& at(int row,int col) const;
50     Vector<T> colToVector(int col);
51     Vector<T> rowToVector(int row);
52     const Size dim() const;
53
54     void zeros();
55     void id();
56     void diag(T value);
57     void diag(const DenseMatrix<T>& rhs);
58     void rand();
59     //upper triangular matrix
60     void randLT();
61
62     void randDiag();
63
64     //rand sym, pos def
65     void randSPD(int notZeroPerLine);
66
67     char isSymmetric() const;
68
69     Vector<T> conjugateGradientSolver(linag::Vector<T> b, double tau,
70     int* count = nullptr,Vector<linag::Vector<double>*>* xs = nullptr,
71     linag::Vector<double>* rs = nullptr);
72
73     double cond();
74
75 };
76
77 template<typename T>
78 const DenseMatrix<T> operator+(const DenseMatrix<T>& x,const
79     DenseMatrix<T>& y);
80
81 template<typename T>
82 const DenseMatrix<T> operator-(const DenseMatrix<T>& x,const
83     DenseMatrix<T>& y);
84
85 template<typename T>
86 const DenseMatrix<T> operator*(const DenseMatrix<T>& x,const
87     DenseMatrix<T>& y);
88
89 template <typename T>
```

## A VERWENDETE KLASSEN

---

```
81 void mult_DenseMatrix_DenseMatrix(linag::DenseMatrix<T>& res, const
82     DenseMatrix<T>& x, const DenseMatrix<T>& y, int idThread, int
83     numThreads);
84
85 template<typename T>
86 const DenseMatrix<T> operator*(const DenseMatrix<T>& x, const T y);
87 template<typename T>
88 const DenseMatrix<T> operator*(const T x, const DenseMatrix<T>& y);
89 template<typename T>
90 const Vector<T> operator*(const Vector<T>& x, const DenseMatrix<T>& y)
91     ;
92 template<typename T>
93 const Vector<T> operator*(const DenseMatrix<T>& x, const Vector<T>& y)
94     ;
95
96 template<typename T>
97 std::ostream& operator<<(std::ostream& output, const DenseMatrix<T>& x
98 );
99
100 //template spezialication
101
102 template <>
103 void linag::DenseMatrix<double>::rand(){
104     for (int i = 0; i < dim().rows; ++i) {
105         for (int j = 0; j < dim().cols; ++j) {
106             at(i,j) = (double)std::rand()/RAND_MAX;
107         }
108     }
109
110     template<>
111     void linag::DenseMatrix<double>::randLT() {
112         for (int i = 0; i < dim().cols; ++i) {
113             for (int j = 0; j < i+1; ++j) {
114                 at(i,j) = (double)std::rand()/RAND_MAX;
115             }
116             for (int j = i+1; j < dim().rows; ++j) {
117                 at(i,j) = 0;
118             }
119         }
120
121         template <>
122         void linag::DenseMatrix<double>::randDiag(){
123             int n = dim().cols<dim().rows?dim().cols:dim().rows;
124             for (int i = 0; i < dim().rows; ++i) {
```

## A VERWENDETE KLASSEN

---

```
125         for (int j = 0; j < dim().cols; ++j) {
127             if(i == j)
128                 at(i,j) = (double)std::rand()/RAND_MAX;
129             else
130                 at(i,j) = 0;
131         }
133     }
135     template<>
136     void linag::DenseMatrix<double>::randSPD(int notZeroPerLine) {
137         assert(isSymmetric() && notZeroPerLine <= dim().cols &&
138             notZeroPerLine%2);
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     template<typename T>
158     std::ostream& linag::operator<<(std::ostream& output, const linag::
159     DenseMatrix<T>& x){
160         for (int i = 0; i < x.dim().rows; ++i) {
161             for (int j = 0; j < x.dim().cols; ++j) {
162                 output << x.at(i,j) << ",\t";
163             }
164             output << '\n';
165         }
166         return output;
167     }
168     template<typename T>
```

## A VERWENDETE KLASSEN

---

```
const linag::Vector<T> linag::operator*(const linag::DenseMatrix<T>&
171   x, const linag::Vector<T>& y){
    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);
      }
    }
181   return res;
}

183 template<typename T>
185 const linag::Vector<T> linag::operator*(const linag::Vector<T>& x,
186   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) {
193     res.at(i) += x.at(j) * y.at(i,j);
      }
    }
195   return res;
}

197 }

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

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

213 }
```

## A VERWENDETE KLASSEN

---

```
215 template<typename T>
216 const linag::DenseMatrix<T> linag::operator*(const linag::DenseMatrix
217 <T>& x, const linag::DenseMatrix<T>& y){
218     assert(x.dim().cols==y.dim().rows);
219
220     linag::DenseMatrix<T> res(x.dim().rows,y.dim().cols);
221
222     //multithreading
223     linag::Vector<std::thread*> threads(THREAD_COUNT>res.dim().cols?
224     res.dim().cols:THREAD_COUNT);
225     for (int l = 0; l < threads.length(); ++l) {
226         //create threads
227         threads.at(l) = new std::thread(linag::
228             mult_DenseMatrix_DenseMatrix<T>,std::ref(res),std::ref(x),std::ref
229             (y),l,threads.length());
230     }
231     for (int l = 0; l < threads.length(); ++l) {
232         threads.at(l)->join();
233     }
234     //std::thread test(std::thread(linag::mult<T>,std::ref(res),std::
235     ref(x),std::ref(y),0,1));
236     //test.join();
237     for (int l = 0; l < threads.length(); ++l) {
238         delete threads.at(l);
239     }
240
241     return res;
242 }
243
244 template <typename T>
245 void linag::mult_DenseMatrix_DenseMatrix(linag::DenseMatrix<T>& res,
246 const linag::DenseMatrix<T>& x,const linag::DenseMatrix<T>& y,int
247 idThread,int numThreads){
248     for (int i = idThread; i < res.dim().cols; i+=numThreads) {
249         for (int j = 0; j < x.dim().rows; ++j) {
250             res.at(j,i)=0;
251             for (int k = 0; k < x.dim().cols; ++k) {
252                 res.at(j,i) += x.at(j,k) * y.at(k,i);
253             }
254         }
255     }
256 }
257
258 template<typename T>
259 const linag::DenseMatrix<T> linag::operator-(const linag::DenseMatrix
260 <T>& x, const linag::DenseMatrix<T>& y){
261     assert(x.dim().cols==y.dim().cols && x.dim().rows==y.dim().rows);
262 }
```

## A VERWENDETE KLASSEN

---

```
257     linag::DenseMatrix<T> res(x.dim().rows,x.dim().cols);
258
259     for (int i = 0; i < x.dim().rows; ++i) {
260         for (int j = 0; j < x.dim().cols; ++j) {
261             res.at(i,j) = x.at(i,j)-y.at(i,j);
262         }
263     }
264     return res;
265 }
266
267 template<typename T>
268 const linag::DenseMatrix<T> linag::operator+(const linag::DenseMatrix<T>& x,const linag::DenseMatrix<T>& y){
269     return x-(-y);
270 }
271
272 template<typename T>
273 const linag::Size linag::DenseMatrix<T>::dim() const{
274     return dimension;
275 }
276
277 template<typename T>
278 linag::Vector<T> linag::DenseMatrix<T>::rowToVector(int row){
279     assert(dim().row >= 0 && dim().cols < dim().rows);
280
281     linag::Vector<T> res(dim().cols);
282
283     for (int i = 0; i < res.dim(); ++i) {
284         res.at(i) = at(row,i);
285     }
286     return res;
287 }
288
289 template<typename T>
290 linag::Vector<T> linag::DenseMatrix<T>::colToVector(int col){
291     assert(col >= 0 && col < dim().cols);
292
293     linag::Vector<T> res(dim().rows);
294
295     for (int i = 0; i < res.dim(); ++i) {
296         res.at(i) = at(i,col);
297     }
298     return res;
299 }
300
301 template<typename T>
302 const T& linag::DenseMatrix<T>::at(int row,int col) const{
303     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
312     return data[row + col*dim().rows];
313 }
314
315 template<typename T>
316 const linag::DenseMatrix<T> linag::DenseMatrix<T>::transpose() const{
317     linag::DenseMatrix<T> res(dim().cols,dim().cols);
318
319     for (int i = 0; i < dim().rows; ++i) {
320         for (int j = 0; j < dim().cols; ++j) {
321             res.at(j,i) = at(i,j);
322         }
323     }
324     return res;
325 }
326
327 template<typename T>
328 const linag::DenseMatrix<T> linag::DenseMatrix<T>::inverse() const{
329     assert(dim().rows == dim().cols);
330
331     linag::DenseMatrix<T> cpy(*this);
332     linag::DenseMatrix<T> res(dim());
333
334     for (int i = 0; i < dim().rows; ++i) {
335         for (int j = 0; j < dim().cols; ++j) {
336             if(i==j)
337                 res.at(i,j)=1;
338             else
339                 res.at(i,j)=0;
340         }
341     }
342
343     //gauss-jordan
344     for (int k = 0; k < dim().cols; ++k) {
345         //int k = 2;
346         T diagValue = cpy.at(k,k);
347         for (int i = 0; i < dim().cols; ++i) {
348             cpy.at(k,i) /= diagValue;
349             res.at(k,i) /= diagValue;
350         }
351         for (int i = 0; i < dim().rows; ++i) {
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2
```

## A VERWENDETE KLASSEN

---

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

## A VERWENDETE KLASSEN

---

```
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 template <typename T>
408 linag::DenseMatrix<T>::DenseMatrix(const DenseMatrix<T> &rhs):
409     dimension(rhs.dim()){
410     if(dimension.rows*dimension.cols > 0)
411     {
412         data = (T*) malloc(rhs.dim().rows * rhs.dim().cols * sizeof(T));
413         assert(data != nullptr);
414         //memcpy is a "dumb" function that only copies bytes
415         std::memcpy(data,rhs.data,rhs.dim().rows * rhs.dim().cols *
416         sizeof(T));
417     }
418     else
419         data = (T*) nullptr;
420 }
421

422 template <typename T>
423 linag::DenseMatrix<T>::DenseMatrix(std::initializer_list<std::
424     initializer_list<T>> init){
425     dimension.rows = init.size();
426     dimension.cols = init.begin()>>size();
427     //check if all rows have same length
428     for(auto row : init){
429         assert(dim().cols == row.size());
430     }

431     if(dim().rows*dim().cols > 0)
432     {
433         data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
434         assert(data != nullptr);
435         //copy
436         int i=0,j;
437         for(auto row:init) {
438             j=0;
439             for (auto item:row) {
440                 at(i,j) = item;
441                 ++j;
442             }
443         }
444     }
445 }
```

## A VERWENDETE KLASSEN

---

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

## A VERWENDETE KLASSEN

---

```
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 }
498
499 template <typename T>
500 linag::Vector<T> linag::DenseMatrix<T>::conjugateGradientSolver(linag
501   ::Vector<T> b, double tau, int* count, linag::Vector<linag::Vector<
502   double>*>* xs, linag::Vector<double>* rs){
503     assert(tau>0 && dim().rows == b.length());
504     if(xs)
505       assert(xs->length() == dim().rows); //exact result after n
506       iterations
507     if(rs)
508       assert(rs->length() == dim().rows); //exact result after n
509       iterations
510
511     linag::Vector<T> r1(dim().rows);
512     linag::Vector<T> r2(dim().rows);
513     linag::Vector<T> d(dim().rows);
514     linag::Vector<T> x(dim().rows);
515     linag::Vector<T> z(dim().rows);
516     x.rand();
517     T alpha;
518     T betta;
519     unsigned long t = 0;
520     r1 = b - (*this)*x;
521     d = r1;
522     if(xs) {
523         for (int i = 1; i < xs->length(); ++i) {
524             xs->at(i) = nullptr;
525         }
526         xs->at(0) = new linag::Vector<double>(x);
527     }
528     if(rs) {
529         for (int i = 1; i < rs->length(); ++i) {
530             rs->at(i) = 0;
531         }
532         rs->at(0) = r1.l2norm();
533     }
534     do{
535         z = (*this)*d;
536         alpha = (r1*r1)/(d*z);
537         x = x + alpha*d;
```

## A VERWENDETE KLASSEN

---

```
535     r2 = r1 - alpha*z;
536     betta = (r2*r2)/(r1*r1);
537     d = r2 + betta*d;
538
539     r1=r2;
540     if(xs && t < xs->length())
541         xs->at(t) = new linag::Vector<double>(x);
542     if(rs && t < rs->length())
543         rs->at(t) = r2.l2norm();
544     ++t;
545     }while (r2.l2norm()>tau);
546     if(count)
547         *count = t;
548     return x;
549 }
550
551 template <typename T>
552 char linag::DenseMatrix<T>::isSymmetric() const{
553     return dim().cols == dim().rows?1:0;
554 }
555
556 template <typename T>
557 void linag::DenseMatrix<T>::diag(T value){
558     int n = dim().cols<dim().rows?dim().cols:dim().rows;
559     for (int i = 0; i < n; ++i) {
560         at(i,i) = value;
561     }
562 }
563
564 template <typename T>
565 void linag::DenseMatrix<T>::diag(const linag::DenseMatrix<T>& rhs){
566     assert(dim() == rhs.dim());
567     zeros();
568     for (int i = 0; i < rhs.dim().rows; ++i) {
569         for (int j = 0; j < rhs.dim().cols; ++j) {
570             if(i == j)
571                 at(i,j) = rhs.at(i,j);
572         }
573     }
574 }
575
576 template <typename T>
577 linag::DenseMatrix<T>::DenseMatrix(const linag::SparseMatrix<T>& rhs)
578 :dimension(rhs.dim()){
579     if(dim().rows*dim().cols > 0)
580     {
581         data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
582         assert(data != nullptr);
```

```

583     zeros();
585     for (int i = 0; i < dim().rows; ++i) {
586         for (int j = rhs.getI().at(i); j < rhs.getI().at(i+1); ++
587             j) {
588             at(i, rhs.getJ().at(j))=rhs.getV().at(j);
589         }
590     }
591     else
592         data = (T*) nullptr;
593 }
594
595 template <typename T>
596 linag::DenseMatrix<T> &linag::DenseMatrix<T>::operator=(const linag::
597     SparseMatrix<T> &rhs){
598     dimension = rhs.dim();
599     if (rhs.dim().rows * rhs.dim().cols > 0) {
600         if (!data) {
601             data = (T *) malloc(rhs.dim().rows * rhs.dim() *
602                 cols * sizeof(T));
603             assert(data != nullptr);
604         } else {
605             data = (T *) realloc(data, rhs.dim().rows * rhs.dim() *
606                 cols * sizeof(T));
607             assert(data != nullptr);
608         }
609     }
610     else
611         data = (T *) nullptr;
612
613     zeros();
614     for (int i = 0; i < dim().rows; ++i) {
615         for (int j = rhs.getI().at(i); j < rhs.getI().at(i + 1);
616             ++j) {
617             at(i, rhs.getJ().at(j)) = rhs.getV().at(j);
618         }
619     }
620     return *this;
621 }
622
623 template <typename T>
624 double linag::DenseMatrix<T>::cond(){
625     Eigen::VectorXcd eigenvalues = toEigen().eigenvalues();
626
627     double min=std::fabs(eigenvalues(0).real()),max = std::fabs(
628     eigenvalues(0).real());
629     for (int i = 1; i < eigenvalues.size(); ++i) {
630         if(std::fabs(eigenvalues(i).real()) > max)

```

```

625         max = std::fabs(eigenvalues(i).real());
627
628     if(std::fabs(eigenvalues(i).real()) < min)
629         min = std::fabs(eigenvalues(i).real());
630     }
631
632     return max/min;
633 }
634
635 #endif //AUFGABE1_DENSEMATRIX_H

```

Listing 8: densematrix.h

#### A.4. Dünnbesetzte Matrix

```

#ifndef AUFGABE1_SPARSEMATRIX_H
#define AUFGABE1_SPARSEMATRIX_H

#include "cmath"
#include "iostream"
#include "vector.h"

#define THREAD_COUNT 8

namespace linag {
    //say class exists without defining it
    template <typename T> class DenseMatrix;

    template <typename T>
    class SparseMatrix {
private:
    Vector<T> v;
    Vector<int> I;
    Vector<int> J;

    Size dimension;

public:
    SparseMatrix()= default;
    ~SparseMatrix() = default;

    explicit SparseMatrix(const DenseMatrix<T>& rhs);
    SparseMatrix<T> & operator=(const DenseMatrix<T>& rhs);
}

```

## A VERWENDETE KLASSEN

---

```
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
46     tau, int* count = nullptr, linag::Vector<linag::Vector<double>*>*
47     xs = nullptr, linag::Vector<double>* rs = nullptr);
48     Vector<T> preCondConjugateGradientSolver(const linag::
49     SparseMatrix<T>& P, const linag::Vector<T> b, double tau, int*
50     count = nullptr, linag::Vector<linag::Vector<double>*>* xs =
51     nullptr, linag::Vector<double>* rs = nullptr);
52
53     };
54
55
56     template<typename T>
57     const SparseMatrix<T> operator*(const SparseMatrix<T>& x, const T
58     y);
59     template<typename T>
60     const SparseMatrix<T> operator*(const T x, const SparseMatrix<T>&
61     y);
62     template<typename T>
63     const Vector<T> operator*(const Vector<T>& x, const SparseMatrix<T
64     >& y);
65     template<typename T>
66     const Vector<T> operator*(const SparseMatrix<T>& x, const Vector<T
67     >& y);
68     //template <typename T>
69     //void mult(linag::Vector<T> &res, const linag::SparseMatrix<T>&
70     x, const linag::Vector<T>& y, int idThread, int numThreads);
71
72 }
73
74 template<typename T>
75 const linag::SparseMatrix<T> linag::operator*(const linag::
76     SparseMatrix<T>& x, const T y){
77     linag::SparseMatrix<T> res(x);
78     for (int i = 0; i < res.v.length(); ++i) {
79         res.v.at(i) *= y;
```

## A VERWENDETE KLASSEN

---

```
70     }
71     return res;
72 }

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 template<typename T>
80 const linag::Vector<T> linag::operator*(const linag::Vector<T>& x,
81     const linag::SparseMatrix<T>& y){
82     std::cout << "undefined" << std::endl;
83 }

84 template<typename T>
85 const linag::Vector<T> linag::operator*(const linag::SparseMatrix<T>&
86     x, const linag::Vector<T>& y){
87     assert(x.dim().cols == y.length());
88     linag::Vector<T> res(y.length());
89     res.zeros();
90     for (int i = 0; i < res.length(); ++i) {
91         for (int j = x.getI().at(i); j < x.getI().at(i + 1); ++j) {
92             res.at(i) += y.at(x.getJ().at(j)) * x.getV().at(j);
93         }
94     }
95     return res;
96 }

97 //template <typename T>
98 //void linag::mult(linag::Vector<T> &res, const linag::SparseMatrix<T
99 //    >& x, const linag::Vector<T>& y, int idThread, int numThreads){

100 //}

101

102 template <typename T>
103 linag::Vector<T> linag::SparseMatrix<T>::conjugateGradientSolver(
104     linag::Vector<T> b, double tau, int* count, linag::Vector<linag::
105     Vector<double>*>* xs, linag::Vector<double>* rs){
106     assert(tau>0 && dim().rows == b.length());
107     if(xs)
108         assert(xs->length() == dim().rows); //exact result after n
109     iterations
110     if(rs)
111         assert(rs->length() == dim().rows); //exact result after n
112     iterations

113     linag::Vector<T> r1(dim().rows);
```

## A VERWENDETE KLASSEN

---

```
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 template<typename T>
```

## A VERWENDETE KLASSEN

---

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

166    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) {
            xs->at(i) = nullptr;
        }
        xs->at(0) = new linag::Vector<double>(x);
    }
188    if(rs) {
        for (int i = 1; i < rs->length(); ++i) {
            rs->at(i) = 0;
        }
        rs->at(0) = r1.l2norm();
    }
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 return x;
}

216

218 template <typename T>
219 char linag::SparseMatrix<T>::isSymmetric() const{
220     return dim().cols == dim().rows?1:0;
}
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 template<typename T>
231 const linag::Size linag::SparseMatrix<T>::dim() const{
232     return dimension;
}
233

234 template <typename T>
235 linag::SparseMatrix<T>::SparseMatrix(const linag::SparseMatrix<T>&
236 rhs){
237     dimension = rhs.dim();
238     v = rhs.v;
239     I = rhs.I;
240     J = rhs.J;
}
241

242 template <typename T>
243 linag::SparseMatrix<T> &linag::SparseMatrix<T>::operator=(const linag
244 ::SparseMatrix<T> &rhs){
245     if(this != &rhs) {
246         dimension = rhs.dim();
247         v = rhs.v;
248         I = rhs.I;
}

```

## A VERWENDETE KLASSEN

---

```
250     J = rhs.J;
251 }
252 return *this;
253 }

254 template <typename T>
255 linag::SparseMatrix<T>::SparseMatrix(const linag::DenseMatrix<T>& rhs
256 ):
257 I(0),J(0),v(0),dimension(rhs.dim()){
258     //calculate array size
259     int vc = 0;
260     int Ic = rhs.dim().rows+1;
261     for (int i = 0; i < rhs.dim().rows; ++i) { //rows
262         for (int j = 0; j < rhs.dim().cols; ++j) { //cols
263             if(std::fabs(rhs.at(i,j)) > 10e-10){
264                 ++vc;
265             }
266         }
267     }
268     //set array size
269     I = linag::Vector<int>(Ic);
270     J = linag::Vector<int>(vc);
271     v = linag::Vector<T>(vc);

272     //convert dense matrix to sparse matrix
273     vc=0;
274     Ic=-1;
275     int Jc=0;
276     for (int i = 0; i < rhs.dim().rows; ++i) { //rows
277         for (int j = 0; j < rhs.dim().cols; ++j) { //cols
278             if(std::fabs(rhs.at(i,j)) > 10e-10){
279                 if(Ic != i){
280                     I.at(++Ic) = vc;
281                 }
282                 v.at(vc++) = rhs.at(i,j);
283                 J.at(Jc++) = j;
284             }
285         }
286     }
287     I.at(++Ic) = vc;
288 }

289 template <typename T>
290 linag::SparseMatrix<T> & linag::SparseMatrix<T>::operator=(const
291 linag::DenseMatrix<T>& rhs){
292     //calculate array size
293     int vc=0;
294     int Ic =rhs.dim().rows+1;
295     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     }
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);

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 }
326
328 #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 postiv 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 $\mu s$ um Gleichungssysteme mit verschiedenen großen dicht besetzten $n \times n$ -Koeffizientenmatrizen mittels Cholesky-Zerlegung zu lösen . . . . .	5
2	Benötigte Zeit in $\mu s$ 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 $A = B + B^T + c \cdot diag(b)$ . . . . .	14