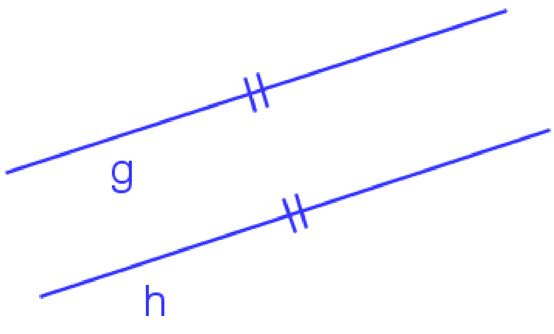
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

CG-Verfahren für dünnbesetzte Matrizen

Rafael Dorigo
Sebastian Hirnschall

Betreut von:
Markus Wess Dipl.-Ing.



29. Januar 2020

Inhaltsverzeichnis

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.

```
template<>
void linag::DenseMatrix<double>::randSPD(int notZeroPerLine) {
    assert(isSymmetric() && notZeroPerLine <= dim().cols && notZeroPerLine%2);

    randLT();
    double c = 50;
    linag::DenseMatrix<double> diagM(dim());
    diagM.randDiag();

    (*this) = (*this) + transpose() + c * diagM;

    for (int i = 0; i < dim().rows; ++i) {
        for (int j = i+std::ceil((double)notZeroPerLine/2); j < dim().cols; ++j) {
            at(i,j) = 0;
        }
        for (int j = 0; j <= i-std::ceil((double)notZeroPerLine/2); ++j) {
            at(i,j) = 0;
        }
    }
}
```

Listing 1: Erstellen einer symmertrisch 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.

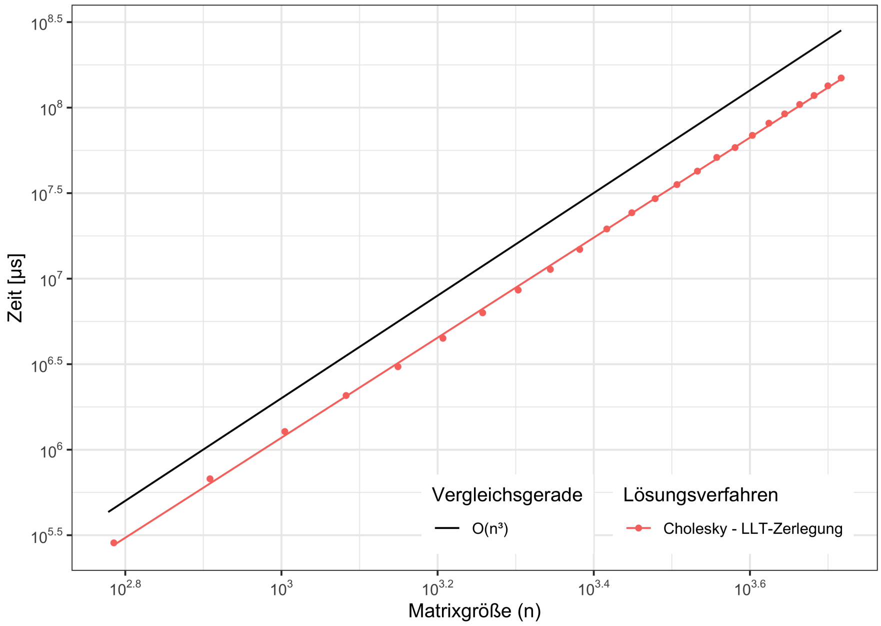


Abbildung 1: Benötigte Zeit in μs um Gleichungssysteme mit verschieden großen dicht besetzten $n \times n$-Koeffizientenmatrizen mittels Cholesky-Zerlegung zu lösen

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 $\tau > 0$

$r_0 = b - Ax_0$
$d_0 = r_0$
$t = 0$
while $\|r_t\| > \tau$ **do**
    $z = Ad_t$
    $\alpha_t = \frac{r_t^T r_t}{d_t^T z}$
    $x_{t+1} = x_t + \alpha_t d_t$
    $r_{t+1} = r_t - \alpha_t z$
    $\beta_t = \frac{r_{t+1}^T r_{t+1}}{r_t^T r_t}$
    $d_{t+1} = r_{t+1} + \beta_t d_t$
    $t = t + 1$

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

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

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

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

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)

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

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

```
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){
    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

    linag::Vector<T> r1(dim().rows);
    linag::Vector<T> r2(dim().rows);
    linag::Vector<T> d(dim().rows);
    linag::Vector<T> x(dim().rows);
    linag::Vector<T> z(dim().rows);
    x.rand();
    T alpha;
    T betta;
    unsigned long t = 0;
    r1 = b - (*this)*x;
    d = r1;
    if(xs) {
        for (int i = 1; i < xs->length(); ++i) {
            xs->at(i) = nullptr;
        }
        xs->at(0) = new linag::Vector<double>(x);
    }
    if(rs) {
        for (int i = 1; i < rs->length(); ++i) {
            rs->at(i) = 0;
        }
        rs->at(0) = r1.l2norm();
    }
    do{
        z = (*this)*d;
        alpha = (r1*r1)/(d*z);
        x = x + alpha*d;
        r2 = r1 - alpha*z;
        betta = (r2*r2)/(r1*r1);
        d = r2 + betta*d;

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

```
        ++t;
    }while (r2.l2norm()>tau);
    if(count)
        *count = t;
    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 spare row* Format verwendet werden. Bei der Implementierung dieses Formats werden anstelle aller Einträge $A_{i,j}, i, j = 1, \ldots, n$ 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

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

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
```

```
    vc=0;
    Ic=-1;
    int Jc=0;
    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){
                if(Ic != i){
                    I.at(++Ic) = vc;
                }
                v.at(vc++) = rhs.at(i,j);
                J.at(Jc++) = j;
            }
        }
    }
    I.at(++Ic) = vc;
}
```

Listing 3: Speicherung einer Matrix im compressed spare row Format

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

        zeros();
        for (int i = 0; i < dim().rows; ++i) {
            for (int j = rhs.getI().at(i); j < rhs.getI().at(i+1); ++
   j) {
                at(i,rhs.getJ().at(j))=rhs.getV().at(j);
            }
        }
    }
    else
        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>
const linag::Vector<T> linag::operator*(const linag::SparseMatrix<T>&
    x,const linag::Vector<T>& y){
    assert(x.dim().cols == y.length());
    linag::Vector<T> res(y.length());
```

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

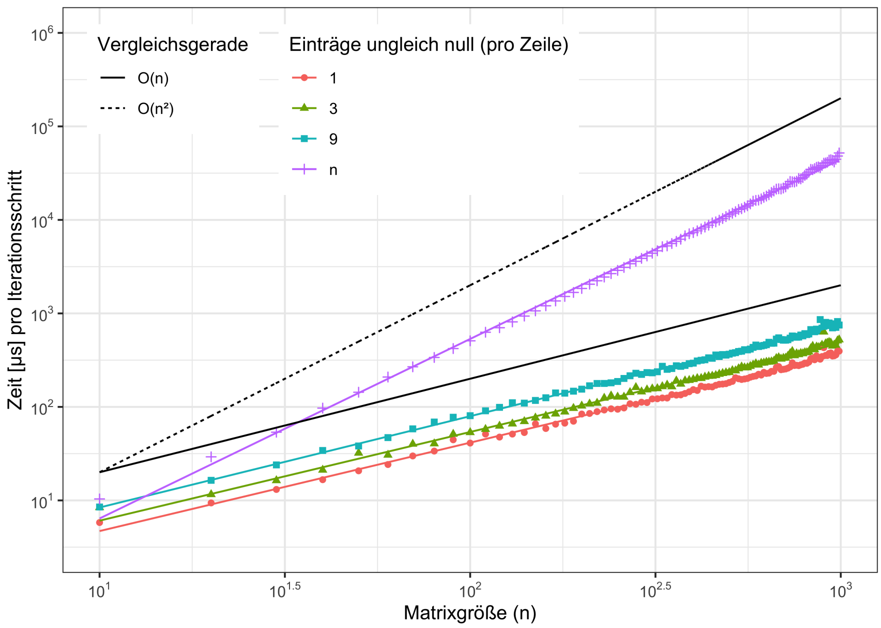


Abbildung 2: Benötigte Zeit in μs für verschieden dicht besetzte $n \times n$-Matrizen im *compressed sparse row*-Format

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 $cond(D^{-1}AD^{-T}) < cond(A)$ 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 $\tau > 0$

$r_0 = b - Ax_0$
$z_0 = P^{-1}r_0$
$d_0 = z_0$
$t = 0$
while $\|r_t\| > \tau$ **do**
    $z = Ad_t$
    $\alpha_t = \frac{r_t^T z_t}{d_t^T z}$
    $x_{t+1} = x_t + \alpha_t d_t$
    $r_{t+1} = r_t - \alpha_t z$
    $z_{t+1} = P^{-1}r_{t+1}$
    $\beta_t = \frac{z_{t+1}^T r_{t+1}}{z_t r_t}$
    $d_{t+1} = z_{t+1} + \beta_t d_t$
    $t = t + 1$

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.

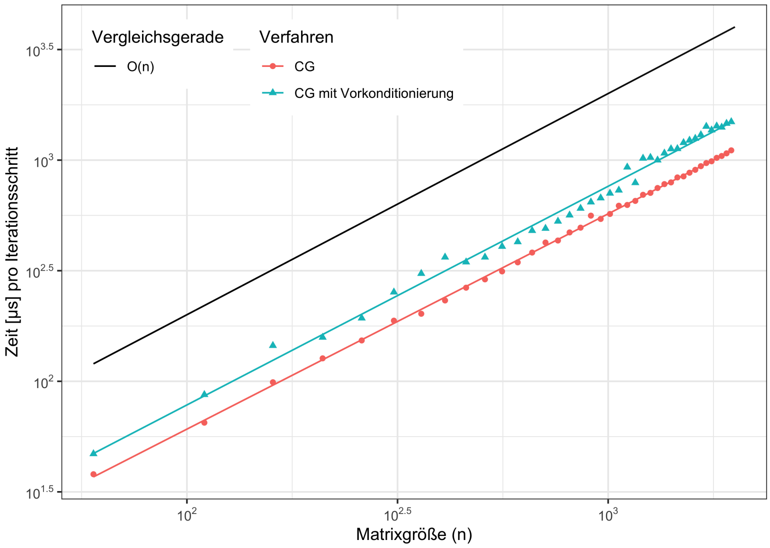
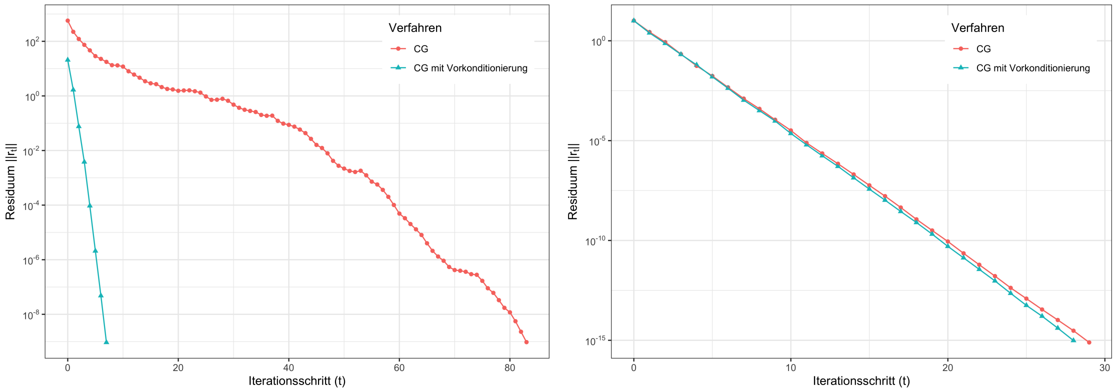


Abbildung 3: Residuum nach Iterationsschritt t für vorkonditioniertes CG- und CG-Verfahren

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 $P = diag(A_{11}, \dots, A_{nn})$ 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.



(a) Koeffizientenmatrix der Form $A = B + B^T + c \cdot diag(b)$[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.

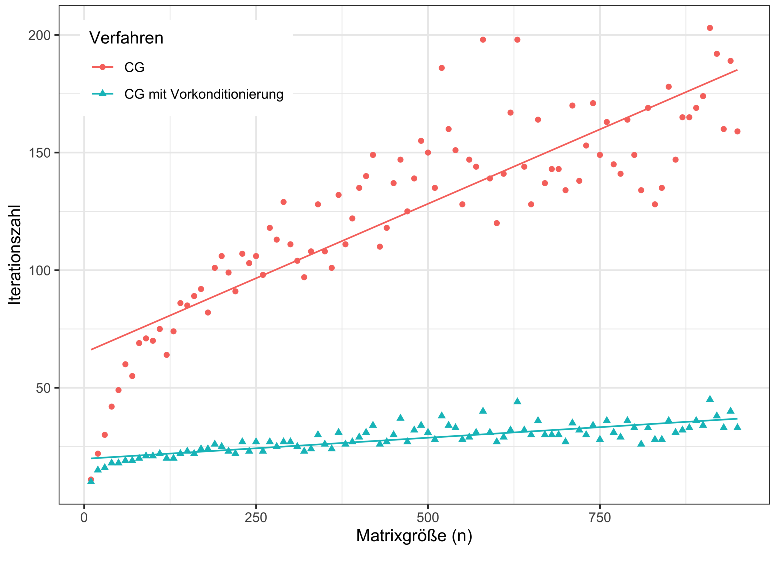


Abbildung 5: Iterationszahlen für verschieden große Koeffizientenmatrizen der Form $A = B + B^T + c \cdot diag(b)$

[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

```
#ifndef AUFGABE1_SIZE_H
#define AUFGABE1_SIZE_H

namespace linag {
    class Size {
    public:
        int rows, cols;
        //default operator=, shallow copy

    };

    bool operator==(const Size& lhs, const Size& rhs)
    {
        return lhs.rows == rhs.rows && lhs.cols == rhs.cols;
    }

    bool operator!=(const Size& lhs, const Size& rhs)
    {
        return !(lhs == rhs);
    }
}


#endif //AUFGABE1_SIZE_H
```

Listing 6: size.h

A.2. Vektor

```
#ifndef AUFGABE1_VECTOR_H
#define AUFGABE1_VECTOR_H

#include <iostream>
#include <cstdlib>
#include <time.h>
#include <cmath>
#include <cassert>
#include <cstring>
#include "Eigen/Dense"
#include "size.h"
//#include "Eigen/src/Core/Matrix.h"

namespace linag {
```

```
//say class exists without defining it
template <typename T> class DenseMatrix;


template<typename T>
class Vector{
private:
    Size dimension;
    T *data;
public:
    ~Vector();

    Vector(const Vector<T> &rhs);
    Vector<T> &operator=(const Vector<T> &);
    explicit Vector(int rows,int cols = 1);
    explicit Vector(const Size dimension);

    Vector(std::initializer_list<T> init);

    Eigen::VectorXd toEigen() const;
    const Vector<T> operator-() const;

    T& at(int index);
    const T &at(int index) const;

    const Size dim() const;
    unsigned long length() const;

    double l2norm();
    double Anorm(const linag::DenseMatrix<T>& A) const;

    void zeros();
    void ones();
    void rand();

};

template<typename T>
const Vector<T> operator+(const Vector<T>& x,const Vector<T>& y);
template<typename T>
const Vector<T> operator-(const Vector<T>& x,const Vector<T>& y);
template<typename T>
T operator*(const Vector<T>& x,const Vector<T>& y);
template<typename T>
const Vector<T> operator*(const Vector<T>& x,const T y);
template<typename T>
const Vector<T> operator/(const Vector<T>& x,const T y);
template<typename T>
const Vector<T> operator*(const T x,const Vector<T>& y);
```

```

    template<typename T>
    std::ostream& operator<<(std::ostream& output,const Vector<T>& x);
}

template<typename T>
std::ostream& linag::operator<<(std::ostream& output,const linag::Vector<T>& x){
    for (int i = 0; i < x.dim().rows; ++i) {
        for (int j = 0; j < x.dim().cols; ++j) {
            output << x.at(i+j) << ",\t";
        }
        output << '\n';
    }
    return output;
}

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


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


template<typename T>
T linag::operator*(const linag::Vector<T>& x,const linag::Vector<T>& y){
    assert(x.dim().rows*x.dim().cols == y.dim().rows*y.dim().cols);
    T res = (T)0;
    for (int i = 0; i < x.dim().rows*x.dim().cols; ++i) {
        res+= y.at(i) *x.at(i);
    }
    return res;
```

```
}


template<typename T>
const linag::Vector<T> operator/(const linag::Vector<T>& x,const T y)
    {
     linag::Vector<T> res(x);
     for (int i = 0; i < res.dim().cols*res.dim().rows; ++i) {
         res.at(i)/=y;
     }
     return res;
}


template<typename T>
const linag::Vector<T> linag::operator-(const linag::Vector<T>& x,
    const linag::Vector<T>& y){
     return x + (-y);
}

template<typename T>
const linag::Vector<T> linag::operator+(const linag::Vector<T>& x,
    const linag::Vector<T>& y){
     assert(x.dim()== y.dim());
     linag::Vector<T> res(x.dim());
     for (int i = 0; i < x.dim().rows*x.dim().cols; ++i) {
             res.at(i) = x.at(i) + y.at(i);
     }
     return res;
}


template <typename T>
void linag::Vector<T>::zeros(){
     for (int i = 0; i < length(); ++i) {
         at(i) = 0;
     }
}
template <typename T>
void linag::Vector<T>::ones(){
     for (int i = 0; i < length(); ++i) {
         at(i) = 1;
     }
}

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

```
    }
}

template<typename T>
double linag::Vector<T>::l2norm(){
    double sum = 0;
    for (int i = 0; i < length(); ++i) {
            sum += std::fabs(double((at(i)*at(i))));
    }
    return std::sqrt(sum);
}

template<typename T>
double linag::Vector<T>::Anorm(const linag::DenseMatrix<T>& A) const{
    return std::sqrt((*this) * A * (*this));
}

template <typename T>
unsigned long linag::Vector<T>::length() const{
    return dim().rows*dim().cols;
}

template <typename T>
const linag::Size linag::Vector<T>::dim() const{
    return dimension;
}

template <typename T>
const T &linag::Vector<T>::at(int index) const{
    assert(index>=0 && index <dim().rows*dim().cols);
    return data[index];
}

template <typename T>
T &linag::Vector<T>::at(int index){
    assert(index>=0 && index <dim().rows*dim().cols);
    return data[index];
}

template <typename T>
const linag::Vector<T> linag::Vector<T>::operator-() const{
    return (T)-1* (*this);
}

template <typename T>
linag::Vector<T>::Vector(std::initializer_list<T> init){
    dimension.rows = init.size();
    dimension.cols = 1;
    if(dim().rows*dim().cols > 0)
```

```
    {
        data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
        assert(data != nullptr);
        //copy
        int i=0;
        for(auto item:init) {
                at(i) = item;
                ++i;
        }
    }
    else
        data = (T*) nullptr;
}

template <typename T>
linag::Vector<T>::Vector(const linag::Size dimension):dimension(
    dimension){
    assert(dimension.rows == 1 || dimension.cols == 1);
    if(dim().rows*dim().cols > 0)
    {
        data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
        assert(data != nullptr);
    }
    else
        data = (T*) nullptr;
}

template <typename T>
linag::Vector<T>::Vector(int rows,int cols){
    assert(rows == 1 || cols == 1);
    dimension.rows=rows;
    dimension.cols=cols;
    if(rows*cols > 0)
    {
        data = (T*) malloc(rows * cols * sizeof(T));
        assert(data != nullptr);
    }
    else
        data = (T*) nullptr;
}

template <typename T>
linag::Vector<T> &linag::Vector<T>::operator=(const linag::Vector<T>
    &rhs){
    if(this != &rhs){
        if(dimension!=rhs.dim()) {
            dimension = rhs.dim();
            if(dim().rows*dim().cols > 0)
            {
```

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

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

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

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

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

```
    }
    return res;
}

#endif //AUFGABE1_VECTOR_H
```

Listing 7: vector.h

A.3. Vollbesetzte Matrix

```
#ifndef AUFGABE1_DENSEMATRIX_H
#define AUFGABE1_DENSEMATRIX_H

//eigen lib
#include "Eigen/Dense"
#include <iostream>
#include <cstring>
#include "size.h"
#include <thread>
#define THREAD_COUNT 8


namespace linag {
    template <typename T> class SparseMatrix;
    template <typename T> class Vector;

template<typename T>
class DenseMatrix{
private:
    Size dimension;
    T* data;
public:
    DenseMatrix(int rows = 0,int cols = 0);
    explicit DenseMatrix(linag::Size dimension);
    ~DenseMatrix();

    DenseMatrix(std::initializer_list<std::initializer_list<T>> init)
    ;


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


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

```


    Eigen::MatrixXd toEigen() const;

    const DenseMatrix<T> operator-() const;

    const DenseMatrix<T> inverse() const;
    const DenseMatrix<T> transpose() const;


    T& at(int row,int col);
    const T& at(int row,int col) const;
    Vector<T> colToVector(int col);
    Vector<T> rowToVector(int row);
    const Size dim() const;

    void zeros();
    void id();
    void diag(T value);
    void diag(const DenseMatrix<T>& rhs);
    void rand();
    //upper tirangular matrix
    void randLT();

    void randDiag();

    //rand sym,pos def
    void randSPD(int notZeroPerLine);

    char isSymmetric() const;

    Vector<T> conjugateGradientSolver(linag::Vector<T> b, double tau,
     int* count = nullptr,Vector<linag::Vector<double>*>* xs = nullptr
    , linag::Vector<double>* rs = nullptr);

    double cond();

};

template<typename T>
const DenseMatrix<T> operator+(const DenseMatrix<T>& x,const
    DenseMatrix<T>& y);
template<typename T>
const DenseMatrix<T> operator-(const DenseMatrix<T>& x,const
    DenseMatrix<T>& y);
template<typename T>
const DenseMatrix<T> operator*(const DenseMatrix<T>& x,const
    DenseMatrix<T>& y);
template <typename T>
```

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


template<typename T>
const DenseMatrix<T> operator*(const DenseMatrix<T>& x,const T y);
template<typename T>
const DenseMatrix<T> operator*(const T x,const DenseMatrix<T>& y);
template<typename T>
const Vector<T> operator*(const Vector<T>& x,const DenseMatrix<T>& y)
    ;
template<typename T>
const Vector<T> operator*(const DenseMatrix<T>& x,const Vector<T>& y)
    ;


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

//template spezialication

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

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

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

```
            for (int j = 0; j < dim().cols; ++j) {
                if(i == j)
                    at(i,j) = (double)std::rand()/RAND_MAX;
                else
                    at(i,j) = 0;
            }
        }
    }

    template<>
    void linag::DenseMatrix<double>::randSPD(int notZeroPerLine) {
        assert(isSymmetric() && notZeroPerLine <= dim().cols && notZeroPerLine%2);

        randLT();
        double c = 50;
        linag::DenseMatrix<double> diagM(dim());
        diagM.randDiag();

        (*this) = (*this) + transpose() + c * diagM;

        for (int i = 0; i < dim().rows; ++i) {
            for (int j = i+std::ceil((double)notZeroPerLine/2); j < dim().cols; ++j) {
                at(i,j) = 0;
            }
            for (int j = 0; j <= i-std::ceil((double)notZeroPerLine/2); ++j) {
                at(i,j) = 0;
            }
        }
    }
}



template<typename T>
std::ostream& linag::operator<<(std::ostream& output,const linag::DenseMatrix<T>& x){
    for (int i = 0; i < x.dim().rows; ++i) {
        for (int j = 0; j < x.dim().cols; ++j) {
            output << x.at(i,j) << ",\t";
        }
        output << '\n';
    }
    return output;
}

template<typename T>
```

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

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

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

template<typename T>
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);

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

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

template<typename T>
const linag::DenseMatrix<T> linag::operator*(const T x,const linag::DenseMatrix<T>& y){
     linag::DenseMatrix<T> res(y.dim());
     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);
          }
     }
     return res;
}

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

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

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

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

    return res;
}

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){
    for (int i = idThread; i < res.dim().cols; i+=numThreads) {
        for (int j = 0; j < x.dim().rows; ++j) {
            res.at(j,i)=0;
            for (int k = 0; k < x.dim().cols; ++k) {
                res.at(j,i) += x.at(j,k) * y.at(k,i);
            }
        }
    }
}


template<typename T>
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);

```

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

    for (int i = 0; i < x.dim().rows; ++i) {
        for (int j = 0; j < x.dim().cols; ++j) {
            res.at(i,j) = x.at(i,j)-y.at(i,j);
        }
    }
    return res;
}

template<typename T>
const linag::DenseMatrix<T> linag::operator+(const linag::DenseMatrix
    <T>& x,const linag::DenseMatrix<T>& y){
    return x-(-y);
}

template<typename T>
const linag::Size linag::DenseMatrix<T>::dim() const{
    return dimension;
}

template<typename T>
linag::Vector<T> linag::DenseMatrix<T>::rowToVector(int row){
    assert(dim().row >= 0 && dim().cols < dim().rows);

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

    for (int i = 0; i < res.dim(); ++i) {
        res.at(i) = at(row,i);
    }
    return res;
}

template<typename T>
linag::Vector<T> linag::DenseMatrix<T>::colToVector(int col){
    assert(col >= 0 && col < dim().cols);

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

    for (int i = 0; i < res.dim(); ++i) {
        res.at(i) = at(i,col);
    }
    return res;
}

template<typename T>
const T& linag::DenseMatrix<T>::at(int row,int col) const{
    assert(row >= 0 && col >= 0 && row < dim().rows && col < dim().
    cols);
```

```
    return data[row + col*dim().rows];
}

template<typename T>
T& linag::DenseMatrix<T>::at(int row,int col){
    assert(row >= 0 && col >= 0 && row < dim().rows && col < dim().
   cols);

    return data[row + col*dim().rows];
}

template<typename T>
const linag::DenseMatrix<T> linag::DenseMatrix<T>::transpose() const{
    linag::DenseMatrix<T> res(dim().cols,dim().cols);

    for (int i = 0; i < dim().rows; ++i) {
        for (int j = 0; j < dim().cols; ++j) {
            res.at(j,i) = at(i,j);
        }
    }
    return res;
}

template<typename T>
const linag::DenseMatrix<T> linag::DenseMatrix<T>::inverse() const{
    assert(dim().rows == dim().cols);

    linag::DenseMatrix<T> cpy(*this);
    linag::DenseMatrix<T> res(dim());

    for (int i = 0; i < dim().rows; ++i) {
        for (int j = 0; j < dim().cols; ++j) {
            if(i==j)
                res.at(i,j)=1;
            else
                res.at(i,j)=0;
        }
    }

    //gauss-jordan
    for (int k = 0; k < dim().cols; ++k) {
    //int k = 2;
        T diagValue = cpy.at(k,k);
        for (int i = 0; i < dim().cols; ++i) {
            cpy.at(k,i) /= diagValue;
            res.at(k,i) /= diagValue;
        }
        for (int i = 0; i < dim().rows; ++i) {
```

```
            if(i==k)
                continue;
            T rowMult = cpy.at(i,k);
            for (int j = 0; j < dim().cols; ++j) {
                cpy.at(i,j) -= rowMult * cpy.at(k,j);
                res.at(i,j) -= rowMult * res.at(k,j);
            }
        }
    }
    //std::cout << cpy << std::endl << res << std::endl;

    return res;
}

template<typename T>
const linag::DenseMatrix<T> linag::DenseMatrix<T>::operator-() const{
    return (T)-1* (*this);
}

template <>
Eigen::MatrixXd linag::DenseMatrix<double>::toEigen () const{
    Eigen::MatrixXd res = Eigen::MatrixXd(dim().rows,dim().cols);

    for (int i = 0; i < dim().rows; ++i) {
        for (int j = 0; j < dim().cols; ++j) {
            res(i,j)=at(i,j);
        }
    }
    return res;
}

template <typename T>
linag::DenseMatrix<T> & linag::DenseMatrix<T>::operator=(const linag
   ::DenseMatrix<T> &rhs){
    if(this != &rhs){
        if(dimension!=rhs.dim()) {
            dimension = rhs.dim();
            if(dim().rows*dim().cols > 0)
            {
                if(!data)
                {
                    data = (T*) malloc (rhs.dim().rows * rhs.dim().
   cols * sizeof(T));
                    assert(data != nullptr);
                }else {
                    data = (T *) realloc(data, rhs.dim().rows * rhs.
   dim().cols * sizeof(T));
                    assert(data != nullptr);
                }
```

```
            }
            else
                data = (T*) nullptr;
        }
        //memcpy is a "dumb" function that only copies bytes
        std::memcpy(data,rhs.data,rhs.dim().rows * rhs.dim().cols *
    sizeof(T));
    }
    return *this;
}

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

template <typename T>
linag::DenseMatrix<T>::DenseMatrix(std::initializer_list<std::
    initializer_list<T>> init){
    dimension.rows = init.size();
    dimension.cols = init.begin()->size();
    //check if all rows have same length
    for(auto row : init){
        assert(dim().cols == row.size());
    }

    if(dim().rows*dim().cols > 0)
    {
        data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
        assert(data != nullptr);
        //copy
        int i=0,j;
        for(auto row:init) {
            j=0;
            for (auto item:row) {
                at(i,j) = item;
                ++j;
            }
```

```
            ++i;
        }
    }
    else
        data = (T*) nullptr;
}

template <typename T>
linag::DenseMatrix<T>::~DenseMatrix(){
    if(data!= nullptr)
        free(data);
}

template <typename T>
linag::DenseMatrix<T>::DenseMatrix(linag::Size dimension):dimension(
   dimension){
    if(dim().rows*dim().cols > 0)
    {
        data = (T*) malloc(dim().rows * dim().cols * sizeof(T));
        assert(data != nullptr);
    }
    else
        data = (T*) nullptr;
}

template <typename T>
linag::DenseMatrix<T>::DenseMatrix(int rows,int cols){
    dimension.rows = rows;
    dimension.cols = cols;
    if(rows*cols > 0)
    {
        data = (T*) malloc(rows * cols * sizeof(T));
        assert(data != nullptr);
    }
    else
        data = (T*) nullptr;
}

template <typename T>
void linag::DenseMatrix<T>::zeros(){
    for (int i = 0; i < dim().rows; ++i) {
        for (int j = 0; j < dim().cols; ++j) {
            at(i,j) = (T)0;
        }
    }
}

template <typename T>
void linag::DenseMatrix<T>::id(){
```

```
    for (int i = 0; i < dim().rows; ++i) {
        for (int j = 0; j < dim().cols; ++j) {
            if(i==j)
                at(i,j) = 1;
            else
                at(i,j) = (T)0;
        }
    }
}

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){
    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

    linag::Vector<T> r1(dim().rows);
    linag::Vector<T> r2(dim().rows);
    linag::Vector<T> d(dim().rows);
    linag::Vector<T> x(dim().rows);
    linag::Vector<T> z(dim().rows);
    x.rand();
    T alpha;
    T betta;
    unsigned long t = 0;
    r1 = b - (*this)*x;
    d = r1;
    if(xs) {
        for (int i = 1; i < xs->length(); ++i) {
            xs->at(i) = nullptr;
        }
        xs->at(0) = new linag::Vector<double>(x);
    }
    if(rs) {
        for (int i = 1; i < rs->length(); ++i) {
            rs->at(i) = 0;
        }
        rs->at(0) = r1.l2norm();
    }
    do{
        z = (*this)*d;
        alpha = (r1*r1)/(d*z);
        x = x + alpha*d;
```

```
        r2 = r1 - alpha*z;
        betta = (r2*r2)/(r1*r1);
        d = r2 + betta*d;

        r1=r2;
        if(xs && t < xs->length())
            xs->at(t) = new linag::Vector<double>(x);
        if(rs && t < rs->length())
            rs->at(t) = r2.l2norm();
        ++t;
    }while (r2.l2norm()>tau);
    if(count)
        *count = t;
    return x;
}

template <typename T>
char linag::DenseMatrix<T>::isSymmetric() const{
    return dim().cols == dim().rows?1:0;
}

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

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


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

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

template <typename T>
linag::DenseMatrix<T> &linag::DenseMatrix<T>::operator=(const linag::
   SparseMatrix<T> &rhs){
            dimension = rhs.dim();
            if (rhs.dim().rows * rhs.dim().cols > 0) {
                if (!data) {
                    data = (T *) malloc(rhs.dim().rows * rhs.dim().
   cols * sizeof(T));
                    assert(data != nullptr);
                } else {
                    data = (T *) realloc(data, rhs.dim().rows * rhs.
   dim().cols * sizeof(T));
                    assert(data != nullptr);
                }
            }else
            data = (T *) nullptr;


        zeros();
        for (int i = 0; i < dim().rows; ++i) {
            for (int j = rhs.getI().at(i); j < rhs.getI().at(i + 1);
   ++j) {
                at(i, rhs.getJ().at(j)) = rhs.getV().at(j);
            }
        }
    return *this;
}

template <typename T>
double linag::DenseMatrix<T>::cond(){
    Eigen::VectorXcd eigenvalues = toEigen().eigenvalues();

    double min=std::fabs(eigenvalues(0).real()),max = std::fabs(
   eigenvalues(0).real());
    for (int i = 1; i < eigenvalues.size(); ++i) {
        if(std::fabs(eigenvalues(i).real()) > max)
```

```
                max = std::fabs(eigenvalues(i).real());

            if(std::fabs(eigenvalues(i).real()) < min)
                min = std::fabs(eigenvalues(i).real());
        }

        return max/min;
    }



#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);
```

```
        SparseMatrix(const SparseMatrix<T>& rhs);
        SparseMatrix<T> &operator=(const SparseMatrix<T> &rhs);

        const SparseMatrix<T> operator-() const;

        const Size dim() const;

        char isSymmetric() const;

        const Vector<T>& getV() const{ return v;};
        const Vector<int>& getI() const{ return I;};
        const Vector<int>& getJ() const{ return J;};

        Vector<T> conjugateGradientSolver(linag::Vector<T> b, double
    tau, int* count = nullptr,linag::Vector<linag::Vector<double>*>*
    xs = nullptr, linag::Vector<double>* rs = nullptr);
        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);


    };


    template<typename T>
    const SparseMatrix<T> operator*(const SparseMatrix<T>& x,const T
    y);
    template<typename T>
    const SparseMatrix<T> operator*(const T x,const SparseMatrix<T>&
    y);
    template<typename T>
    const Vector<T> operator*(const Vector<T>& x,const SparseMatrix<T
    >& y);
    template<typename T>
    const Vector<T> operator*(const SparseMatrix<T>& x,const Vector<T
    >& y);
    //template <typename T>
    //void mult(linag::Vector<T> &res, const linag::SparseMatrix<T>&
    x,const linag::Vector<T>& y,int idThread,int numThreads);

}

template<typename T>
const linag::SparseMatrix<T> linag::operator*(const linag::
    SparseMatrix<T>& x,const T y){
    linag::SparseMatrix<T> res(x);
    for (int i = 0; i < res.v.length(); ++i) {
        res.v.at(i) *= y;
```

```
    }
    return res;
}

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

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

template<typename T>
const linag::Vector<T> linag::operator*(const linag::SparseMatrix<T>&
     x,const linag::Vector<T>& y){
    assert(x.dim().cols == y.length());
    linag::Vector<T> res(y.length());
    res.zeros();
    for (int i = 0; i < res.length(); ++i) {
        for (int j = x.getI().at(i); j < x.getI().at(i + 1); ++j) {
            res.at(i) += y.at(x.getJ().at(j)) * x.getV().at(j);
        }
    }
    return res;
}

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

//}

template <typename T>
linag::Vector<T> linag::SparseMatrix<T>::conjugateGradientSolver(
    linag::Vector<T> b, double tau, int* count,linag::Vector<linag::
    Vector<double>*>* xs, linag::Vector<double>* rs){
    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

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

```
    linag::Vector<T> r2(dim().rows);
    linag::Vector<T> d(dim().rows);
    linag::Vector<T> x(dim().rows);
    linag::Vector<T> z(dim().rows);
    x.rand();
    T alpha;
    T betta;
    unsigned long t = 0;
    r1 = b - (*this)*x;
    d = r1;
    if(count)
        *count = 0;
    if(xs) {
        for (int i = 1; i < xs->length(); ++i) {
            xs->at(i) = nullptr;
        }
        xs->at(0) = new linag::Vector<double>(x);
    }
    if(rs) {
        for (int i = 1; i < rs->length(); ++i) {
            rs->at(i) = 0;
        }
        rs->at(0) = r1.l2norm();
    }
    do{
        z = (*this)*d;
        alpha = (r1*r1)/(d*z);
        x = x + alpha*d;
        r2 = r1 - alpha*z;
        betta = (r2*r2)/(r1*r1);
        d = r2 + betta*d;

        r1=r2;
        if(count)
            ++*count;
        if(xs && t < xs->length())
            xs->at(t) = new linag::Vector<double>(x);
        if(rs && t < rs->length())
            rs->at(t) = r2.l2norm();
        ++t;
    }while (r2.l2norm()>tau);

    return x;
}


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<
    linag::Vector<double>*>* xs, linag::Vector<double>* rs){
    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

    linag::Vector<T> r1(dim().rows);
    linag::Vector<T> r2(dim().rows);
    linag::Vector<T> d(dim().rows);
    linag::Vector<T> x(dim().rows);
    linag::Vector<T> z(dim().rows);
    linag::Vector<T> z1(dim().rows);
    linag::Vector<T> z2(dim().rows);
    x.rand();
    T alpha;
    T betta;
    unsigned long t = 0;
    r1 = b - (*this)*x;
    z1 = Pinv * r1;
    d = z1;
    if(count)
        *count = 0;
    if(xs) {
        for (int i = 1; i < xs->length(); ++i) {
            xs->at(i) = nullptr;
        }
        xs->at(0) = new linag::Vector<double>(x);
    }
    if(rs) {
        for (int i = 1; i < rs->length(); ++i) {
            rs->at(i) = 0;
        }
        rs->at(0) = r1.l2norm();
    }
    do{
        z = (*this)*d;
        alpha = (r1*z1)/(d*z);
        x = x + alpha*d;
        r2 = r1 - alpha*z;
        z2 = Pinv*r2;
        betta = (z2*r2)/(z1*r1);
        d = z2 + betta*d;

```

```
        r1=r2;
        z1=z2;
        if(count)
            ++*count;
        if(xs && t < xs->length())
            xs->at(t) = new linag::Vector<double>(x);
        if(rs && t < rs->length())
            rs->at(t) = r2.l2norm();
        ++t;
    }while (r2.l2norm()>tau);

    return x;
}


template <typename T>
char linag::SparseMatrix<T>::isSymmetric() const{
    return dim().cols == dim().rows?1:0;
}

template <typename T>
const linag::SparseMatrix<T> linag::SparseMatrix<T>::operator-()
    const{
    linag::SparseMatrix<T> res(*this);
    res.v = res.v *(-1);
    return res;
}

template<typename T>
const linag::Size linag::SparseMatrix<T>::dim() const{
    return dimension;
}

template <typename T>
linag::SparseMatrix<T>::SparseMatrix(const linag::SparseMatrix<T>&
    rhs){
    dimension = rhs.dim();
    v = rhs.v;
    I = rhs.I;
    J = rhs.J;
}

template <typename T>
linag::SparseMatrix<T> &linag::SparseMatrix<T>::operator=(const linag
    ::SparseMatrix<T> &rhs){
    if(this != &rhs) {
        dimension = rhs.dim();
        v = rhs.v;
        I = rhs.I;
```

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

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
    vc=0;
    Ic=-1;
    int Jc=0;
    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){
                if(Ic != i){
                    I.at(++Ic) = vc;
                }
                v.at(vc++) = rhs.at(i,j);
                J.at(Jc++) = j;
            }
        }
    }
    I.at(++Ic) = vc;
}

template <typename T>
linag::SparseMatrix<T> & linag::SparseMatrix<T>::operator=(const
   linag::DenseMatrix<T>& rhs){
    //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;
            }
        }
    }
    //rows/cols
    dimension.rows=rhs.dim().rows;
    dimension.cols=rhs.dim().cols;
    //set array size
    I = linag::Vector<int>(Ic);
    J = linag::Vector<int>(vc);
    v = linag::Vector<T>(vc);

    //convert dense matrix to sparse matrix
    vc=0;
    Ic=0;
    int Jc=0;
    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){
                if(!Ic || Ic != i){
                    I.at(Ic++) = vc;
                }
                v.at(vc++) = rhs.at(i,j);
                J.at(Jc++) = j;
            }
        }
    }
}


#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

Abbildungsverzeichnis