Mathematisches Praktikum — Sommersemester 2018
Prof. Dr. Arnold Reusken — Sven Gross, Hauke Sass, Johannes Stemick

igpm Institut für Geometrie und Praktische Mathematik | RWTH AACHEN UNIVERSITY

Aufgabe 3

Bearbeitungszeit: zwei Wochen (bis Freitag, 18. Mai 2018)

Mathematischer Hintergrund: Iterative Lösungsverfahren

Elemente von C++: Klassen, Container, Überladen von Operatoren, dynamische Speicherverwaltung, Kommandozeilenargumente, Makefiles

---

Aufgabenstellung

Schreiben Sie ein Programm, das zu gegebener Matrix A und rechter Seite b mit iterativen Lösungsverfahren (Jacobi-, Gauß-Seidel- und CG-Verfahren) die Lösung des zugehörigen linearen Gleichungssystems annähert. Vervollständigen Sie zu diesem Zweck die Vektorklasse, die Ihnen zur Verfügung gestellt wird, und verfassen Sie analog eine Matrixklasse.

Lineare Iterationsverfahren

Wir betrachten folgende Aufgabe:
Für $\mathbf{A} \in \mathbb{R}^{n\times n}$ (nichtsingulär) und $\mathbf{b} \in \mathbb{R}^n$ ist das Gleichungssystem

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

zu lösen.
Wenn n groß und $\mathbf{A}$ dünnbesetzt ist, sind direkten Methoden (LR-Zerlegung, QR-Zerlegung) im allgemeinen ungeeignet [4]. Man weicht stattdessen auf gewisse iterative Verfahren aus. Ein einfacher, jedoch weitreichender Ansatz liegt darin, (1) als *Fixpunktgleichung* zu schreiben:

$$\mathbf{x} = \mathbf{x} + \mathbf{C}(\mathbf{b} - \mathbf{A}\mathbf{x}) =: \Phi(\mathbf{x}), \tag{2}$$

wobei $\mathbf{C} \in \mathbb{R}^{n\times n}$ eine geeignet zu wählende nichtsinguläre Matrix ist. Offensichtlich ist $\mathbf{x}$ Fixpunkt von (2) genau dann, wenn $\mathbf{x}$ Lösung von (1) ist. Die Lösung von (1), also der Fixpunkt von Φ, wird mit $\mathbf{x}^*$ bezeichnet. Mit einem Startwert $\mathbf{x}^0 \in \mathbb{R}^n$ führt dies auf die Fixpunktiteration

$$\begin{aligned} \mathbf{x}^{k+1} &= \Phi(\mathbf{x}^k) = \mathbf{x}^k + \mathbf{C}(\mathbf{b} - \mathbf{A}\mathbf{x}^k) \\ &= (\mathbf{I} - \mathbf{C}\mathbf{A})\mathbf{x}^k + \mathbf{C}\mathbf{b}, \quad k = 0, 1, 2, \dots . \end{aligned} \tag{3}$$

Für den Fehler $\mathbf{e}^k := \mathbf{x}^k - \mathbf{x}^*$ ergibt sich

$$\mathbf{e}^{k+1} = \mathbf{x}^{k+1} - \mathbf{x}^* = (\mathbf{I} - \mathbf{C}\mathbf{A})\mathbf{e}^k,$$

also $\mathbf{e}^k = (\mathbf{I} - \mathbf{C}\mathbf{A})^k\mathbf{e}^0$, $k = 0, 1, 2, \dots$.
Weil die Fehlerfortpflanzung *linear* ist, wird eine Methode vom Typ (3) lineares Iterationsverfahren genannt.

Jacobi-Verfahren Beim *Jacobi*-Verfahren wird in (3)

$$\mathbf{C} = \mathbf{D}^{-1} = (\operatorname{diag}(\mathbf{A}))^{-1}$$

gewählt. Die Iterationsvorschrift lautet in diesem Fall

$$\mathbf{x}^{k+1} = (\mathbf{I} - \mathbf{D}^{-1}\mathbf{A})\mathbf{x}^k + \mathbf{D}^{-1}\mathbf{b}. \tag{4}$$

Damit diese Methode durchführbar ist, muß $a_{i,i} \neq 0$ für alle $i = 1, 2, \ldots, n$ gelten. Zerlegt man die Matrix $\mathbf{A}$ in $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$, wobei $-\mathbf{L}$ und $-\mathbf{U}$ nur die Einträge von $\mathbf{A}$ unterhalb bzw. oberhalb der Diagonalen enthält, läßt sich (4) äquivalent folgendermaßen schreiben:

$$\mathbf{D}\mathbf{x}^{k+1} = (\mathbf{L} + \mathbf{U})\mathbf{x}^k + \mathbf{b},$$

d. h., in Komponentenschreibweise ergibt sich:

Algorithmus für das Jacobi-Verfahren:
Gegeben: Startvektor $\mathbf{x}^0 \in \mathbb{R}^n$. Für $k = 0, 1, \ldots$ berechne:

$$x_i^{k+1} = a_{i,i}^{-1} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{i,j} x_j^k \right), \quad i = 1, 2, \ldots, n.$$

Beim Jacobi-Verfahren wird die $i-$te Gleichung nach der $i-$ten Unbekannten x_i aufgelöst, wobei für die übrigen Unbekannten (x_j, $j \neq i$) Werte aus dem *vorherigen* Iterationsschritt verwendet werden. Beim Jacobi-Verfahren lassen sich also die Komponenten von $\mathbf{x}^{k+1}$ unabhängig voneinander *parallel* aus $\mathbf{x}^k$ ermitteln. Diese Methode ist leicht *parallelisierbar*.

Gauß-Seidel-Verfahren Zerlegt man die Matrix $\mathbf{A}$ wie vorher, kann man $\mathbf{C} = (\mathbf{D} - \mathbf{L})^{-1}$ wählen, woraus sich das *Gauß-Seidel*-Verfahren

$$\mathbf{x}^{k+1} = (\mathbf{I} - (\mathbf{D} - \mathbf{L})^{-1}\mathbf{A})\mathbf{x}^k + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{b}$$

ergibt. Obige Iterationsvorschrift lautet äquivalent

$$(\mathbf{D} - \mathbf{L})\mathbf{x}^{k+1} = \mathbf{U}\mathbf{x}^k + \mathbf{b}.$$

In Komponentenschreibweise erhält man

Algorithmus für das Gauß-Seidel-Verfahren:
Gegeben: Startvektor $\mathbf{x}^0 \in \mathbb{R}^n$. Für $k = 0, 1, \ldots$ berechne:

$$x_i^{k+1} = a_{i,i}^{-1} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{k+1} - \sum_{j=i+1}^{n} a_{i,j} x_j^k \right), \quad i = 1, 2, \ldots, n.$$

Bei der Berechnung der $i-$ten Komponente der neuen Annäherung $\mathbf{x}^{k+1}$ verwendet man also die *bereits vorher berechneten Komponenten der neuen Annäherung*. Da die Komponenten von $\mathbf{x}^{k+1}$ voneinander abhängen, läßt sich das Gauß-Seidel-Verfahren nicht so einfach parallelisieren.

CG-Methode

Die Methode der konjugierten Gradienten (engl.: *conjugate gradients*, daher meistens mit CG abgekürzt) ist eines der derzeit bekanntesten effizienten Verfahren zur Lösung großer dünnbesetzter Gleichungssysteme $\mathbf{A}\mathbf{x} = \mathbf{b}$ mit *symmetrisch positiv definiter* Matrix $\mathbf{A}$. Wenn $\mathbf{A} \in \mathbb{R}^{n\times n}$ s.p.d. ist, gelten die folgenden zwei grundlegenden Eigenschaften:

- Für $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ definiert $\langle \mathbf{x}, \mathbf{y} \rangle_{\mathbf{A}} := \mathbf{x}^T \mathbf{A} \mathbf{y}$ ein *Skalarprodukt* auf $\mathbb{R}^n$.
- Sei $f(\mathbf{x}) := \frac{1}{2}\mathbf{x}^T \mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x}$, $\mathbf{b}, \mathbf{x} \in \mathbb{R}^n$. Dann gilt, daß f ein eindeutiges Minimum hat und

$$\mathbf{A}\mathbf{x}^* = \mathbf{b} \iff f(\mathbf{x}^*) = \min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}). \tag{5}$$

Die Kernidee zur Minimierung von $f(\mathbf{x})$ liegt darin, ausgehend von einer bereits erhaltenen Näherungslösung, einen kleinen Schritt in eine günstige Abstiegsrichtung zu machen, also wiederholt eine Minimierung in einem eingeschränkten Raum $U \subset \mathbb{R}^n$ durchzuführen. Die Herleitung der CG-Methode kann man im Buch [4] und [5] nachlesen.

Grundidee:

- In der CG-Methode wird, ausgehend von $U_1 := \text{span}\{\mathbf{r}^0 := \mathbf{b} - \mathbf{A}\mathbf{x}^0\}$, eine Reihe von k-dimensionalen *optimalen* Teilräumen U_k, $k = 2, 3, \ldots$, des $\mathbb{R}^n$ konstruiert.
- Im $k-$ten Schritt wird die in der Energie-Norm *beste* Annäherung $\mathbf{x}^k \in U_k$ der Lösung $\mathbf{x}^*$ berechnet, d. h. $\|\mathbf{x}^k - \mathbf{x}^*\|_\mathbf{A} = \min_{\mathbf{x}\in U_k} \|\mathbf{x} - \mathbf{x}^*\|_\mathbf{A}$. Deshalb ist das CG-Methode eine Projektionsmethode. $\mathbf{x}^k$ ist die $\mathbf{A}$-orthogonale Projektion von $\mathbf{x}^*$ auf U_k.
- Um $\mathbf{x}^k$ zu berechnen, wird eine $\mathbf{A}-$orthogonale Basis des Teilraumes U_k konstruiert.

Wir verwenden die Notation $\langle \cdot, \cdot \rangle$ für das Euklidische Skalarprodukt: $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T\mathbf{y}$ für $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$.

Algorithmus für das CG-Verfahren:
Gegeben: $\mathbf{A} \in \mathbb{R}^{n\times n}$ s.p.d., $\mathbf{b} \in \mathbb{R}^n$, Startvektor $\mathbf{x}^0 \in \mathbb{R}^n$. Berechne $\mathbf{p}^0 = \mathbf{r}^0 = \mathbf{b} - \mathbf{A}\mathbf{x}^0$, $\gamma^0 := \langle \mathbf{r}^0, \mathbf{r}^0 \rangle$.
Für $k = 0, 1, 2, \ldots$, falls $\frac{\|\mathbf{r}^k\|_2}{\|\mathbf{b}\|_2} > \epsilon$ (ϵ: vorgegebene Toleranz) und $k < k_{max}$:

$$
\begin{aligned}
&\mathbf{q}^k := \mathbf{A}\mathbf{p}^k, \quad \alpha^k = \frac{\gamma^k}{\langle \mathbf{q}^k, \mathbf{p}^k \rangle}, \\
&\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^k \mathbf{p}^k, \\
&\mathbf{r}^{k+1} = \mathbf{r}^k - \alpha^k \mathbf{q}^k, \\
&\gamma^{k+1} = \langle \mathbf{r}^{k+1}, \mathbf{r}^{k+1} \rangle, \\
&\mathbf{p}^{k+1} = \mathbf{r}^{k+1} + \frac{\gamma^{k+1}}{\gamma^k} \mathbf{p}^k.
\end{aligned}
$$

Bemerkung.

- Das CG-Verfahren ist *endlich*, d. h. $\mathbf{x}^k = \mathbf{x}^*$ für ein $k \leq n$. Aufgrund von Rundungseinflüssen wird dies in der Praxis jedoch nicht gelten.
- Die CG-Methode ist ein *nichtlineares* Verfahren:

  $$\mathbf{e}^{k+1} = \varphi(\mathbf{e}^k),$$

  wobei φ eine *nichtlineare* Funktion ist.

Vektor- und Matrixklasse

Um den Algorithmus möglichst übersichtlich gestalten zu können, sollen die Vektor- und Matrixoperationen in eigenen Klassen implementiert werden. Neben der Header–Datei `vektor.h` zu der Vektorklasse werden Ihnen einige Beispielfunktionen in `vektor.cpp` zur Verfügung gestellt. Ihre Aufgabe ist es, die Funktionen der Vektorklasse zu vervollständigen und analog eine Matrixklasse `Matrix` mit den entsprechenden Operationen zu entwickeln. Dazu sollten Sie zwei Dateien `matrix.h` und `matrix.cpp` anlegen.

Anforderungen an die Matrixklasse

An Ihre Matrixklasse werden einige Anforderungen gestellt. Durch

```
Matrix A(m,n);
```

wird eine Matrix `A` mit `m` Zeilen und `n` Spalten angelegt und mit Nullen gefüllt. Um auch Felder von Matrizen anlegen zu können (dabei kann der Konstruktor nur ohne Parameter aufgerufen werden), sollten `m` und `n` den Default–Wert `1` bekommen. Mittels

```
A(i,j)
```

erfolgt der lesende und schreibende Zugriff auf das Matrixelement A_{ij}. Falls die Matrix das gewünschte Element nicht enthält, sollte mit einer Fehlermeldung abgebrochen werden. Die Dimension der Matrix sollte sich analog zur Elementfunktion `Laenge()` der Vektorklasse über die beiden Elementfunktionen `Zeilen()` und `Spalten()` auslesen lassen. Mit

```
A.ReDim(m,n);
```

wird die Matrix neu dimensioniert und wieder mit Nullen gefüllt. Der (private) Daten-Teil Ihrer Klasse könnte beispielsweise die folgende Form haben:

```
size_t Zeil, Spalt;
std::vector< std::vector<double> > mat;
```

In `Zeil` und `Spalt` können Sie die Matrixdimension speichern. Neben den üblichen Operatoren für Matrizen sollte Ihre Matrixklasse auch entsprechende Operatoren für das Matrix-Vektor-Produkt zur Verfügung stellen.

Optimierungsmöglichkeit. Warum ist das oben angegebene Format von verschachtelten `std::vector` für `mat` nicht optimal? Wie lässt sich die Matrix effizienter speichern?

Bemerkung. Die Matrizen, die wir betrachten, sind *vollbesetzt*. In der Praxis ist es aber oftmals so, dass die Matrizen sehr groß sind, aber eine bestimmte Struktur haben und *dünnbesetzt* sind. Das heißt, dass pro Zeile die Anzahl der Einträge durch eine (kleine) Konstante beschränkt ist. Damit lassen sich dünnbesetzte Matrizen auf weniger Speicherplatz speichern. Alle Iterationsverfahren, die wir hier behandeln, sind insbesondere auch für solche Matrizen geeignet, weil sie die dünnbesetzte Struktur von Matrizen erhalten. Aus Implementierungssicht kann man einfach die Datenstruktur der Matrizen tauschen, sofern nötig.

Sicherheit der Vektor- und Matrixklassen

Ein sehr häufig auftretender Fehler besteht darin, dass sich der Programmierer bei den Zeilen- und Spaltenindizes irrt – besonders oft ist der angegebene Index um Eins zu groß oder zu klein. Besonders schwer zu finden ist ein solcher Fehler, wenn der Zugriff dann über die Grenzen eines Arrays hinaus erfolgt und fremde Speicherinhalte falsch interpretiert oder sogar überschrieben werden. Wird dabei der dem Programm zugeteilte Speicherplatz überschritten, so bricht das Betriebssystem die Anwendung mit einer Schutzverletzung[1] ab. Wird der zugewiesene Speicher nicht überschritten, so kann es sein, dass das Programm weiterläuft, aber fehlerhafte Ergebnisse produziert.

Deshalb sollten Sie solchen Fehlern vorbeugen. Alle Zugriffe auf die Elemente des Arrays, das der Vektor- und Matrixklasse zur Datenspeicherung zugrunde liegt, sollen ausschließlich in den Zugriffsoperatoren geschehen. Alle anderen Funktionen müssen die Zugriffsoperatoren benutzen, wenn sie Einträge lesen oder schreiben.

Schreiben Sie in den Zugriffsoperatoren eine Überprüfung, ob die Indizes im gültigen Bereich liegen und brechen Sie das Programm ab, falls dieser überschritten ist[2]. Um keine großen Geschwindigkeitseinbußen zu erleiden, sollten Sie diese Überprüfung per Präprozessoranweisung `#ifdef` abschalten können, wenn Sie sicher sind, dass ihr Programm fehlerfrei läuft.

[1] Fehlermeldung: `segmentation fault`

[2] engl. *range checking*

Test der Vektor- und Matrixklassen

Die Datei `test.cpp` soll Ihnen bei der Fehlersuche in den Klassen helfen. Bevor Sie also zur Programmierung der iterativen Lösungsverfahren schreiten, sollten Ihre Klassen sämtliche darin enthaltenen Tests bestehen. Dazu müssen Sie lediglich aus `test.cpp` und Ihren beiden Dateien `vektor.cpp` und `matrix.cpp` ein ausführbares Programm erzeugen und starten.

Schnittstellen

Durch die Header–Datei `unit.h` werden Ihnen einige Variablen und Funktionen zur Verfügung gestellt. Wie üblich, müssen Sie zu Beginn die Funktion

```
void Start ( int Bsp, Matrix &A, Vektor &x0, Vektor &b, double &tol, int
    &maxiter);
```

aufrufen, wobei `Bsp` wieder im Bereich von `1` bis `AnzahlBeispiele` liegen darf. Zur Matrix `A` soll dann mit iterativen Lösungsverfahren die Lösung der linearen Gleichungssysteme $\mathbf{Ax} = \mathbf{b}$ angenähert werden. Als Startvektor verwenden Sie bitte den Vektor `x0`. Wenn $\frac{\|\mathbf{r}^k\|_2}{\|\mathbf{b}\|_2} <$ `tol` oder $k \geq$ `maxiter` erfüllt sind, können Sie die Iteration abbrechen. Ihr Resultat übergeben Sie zusammen mit der Anzahl der benötigten Iterationen an die Funktion

```
void Ergebnis ( const Vektor &x, int Iterationen, int Methode );
```

wo es bewertet und ausgegeben wird (Methode `0`: Jacobi; Methode `1`: GS; Methode `2`: CG).
Bitte speichern Sie das relative Residuum $\frac{\|\mathbf{r}^k\|_2}{\|\mathbf{b}\|_2}$ in einem Vektor und schreiben Sie diesen in eine Datei (mit `ofstream`), z. B. `jacobi.txt`. Mit Hilfe von gnuplot ([2], [3]), können wir diese Größe wie folgt zeichnen:

```
> plot "jacobi.txt" w lp
```

Um eine logarithmische Skalierung der Achsen zu aktivieren, gibt es bei gnuplot die Befehle `set logscale x` bzw. `set logscale y`.

Kommandozeilenparameter

Die Nummer des zu rechnenden Beispiels soll diesmal nicht zur Laufzeit des Programms abgefragt, sondern als Kommandozeilenparameter übergeben werden. Hat das ausführbare Programm etwa den Namen `loesung`, so soll z. B. mit dem Aufruf "`loesung 2`" das zweite Beispiel gerechnet werden. Verwenden Sie dazu den Mechanismus von C++ zur Übergabe von Kommandozeilenparametern: Deklarieren Sie das Hauptprogramm als Funktion

```
int main ( int argc, char *argv[] );
```

Die Zahl `argc` gibt Ihnen dann die Zahl der Argumente an, wobei der Programmname als erstes Argument zählt. Im obigen Beispiel hat `argc` daher den Wert `2`. Die Variable `argv` ist ein Feld von Zeigern, so dass `argv[i]` auf die Zeichenkette[3] im C-Format (d. h. nullterminiert) zeigt, die dem `i`-ten Kommandozeilenparameter entspricht. Also zeigt `argv[0]` auf den String `"loesung"` und `argv[1]` auf den String `"2"`.

Zur Umwandlung von Zeichenketten in Integerwerte können Sie sogenannte *String-Streams* verwenden. Dazu müssen Sie die Standard-Header-Datei `sstream`[4] in Ihr Programm einbinden. Ist die Zeichenkette `s` beispielsweise definiert als `char s[10]`, dann können Sie mit der Deklaration

```
istringstream IStr(s);
```

[3] engl. *string*
[4] Vorsicht! Die Schreibweise `strstream` ist veraltet und sollte nicht mehr benutzt werden.

einen Input-String-Stream `IStr` definieren, dessen Inhalt aus der Zeichenkette `s` besteht. Aus diesem können Sie dann Objekte verschiedenen Typs wie aus dem Eingabe-Stream `cin` auslesen. Achten Sie darauf, dass Ihr Programm auch auf fehlerhafte Eingaben sinnvoll reagiert.

Makefile

Das `Makefile` zur Erzeugung des Testprogramms für die Vektor- und Matrixklasse könnte zum Beispiel wie folgt beginnen (<*Tab*> steht für das Tab(ulator)-Zeichen):

```
CXXFLAGS = -O2 -Wall

test: vektor.o matrix.o test.o
   <Tab> $(CXX) $(CXXFLAGS) -o test vektor.o matrix.o test.o

test.o: test.cpp vektor.h matrix.h
   <Tab> $(CXX) $(CXXFLAGS) -c test.cpp

vektor.o: vektor.cpp vektor.h
   <Tab> $(CXX) $(CXXFLAGS) -c vektor.cpp

...
```

Zur Erzeugung von `test` werden also die Dateien `vektor.o`, `matrix.o` und `test.o` benötigt, und der Befehl dazu lautet

```
g++ -O2 -Wall -o test vektor.o matrix.o test.o
```

Erstellen Sie ein vollständiges `Makefile`, damit es Ihre Programme erzeugt. Schreiben Sie insbesondere auch ein Ziel `clean`, das alle kompilierten Dateien löscht. Markieren Sie dieses Ziel als `.PHONY`. [1]

Literatur

[1] *Dokumentation zu GNU Make.* `http://www.gnu.org/software/make/`.

[2] *Dokumentation zu gnuplot.* `http://www.gnuplot.info/documentation.html`.

[3] *Grundkurs Gnuplot.* `http://www.mathematik.hu-berlin.de/~lamour/WR_I_WR_II_Num_I/gnuplotkurs.html`.

[4] W. Dahmen and A. Reusken. *Numerische Mathematik für Ingenieure und Naturwissenschaftler.* Springer Verlag, Heidelberg, 2. edition, 2008.

[5] A. Meister. *Numerik linearer Gleichungssysteme.* Vieweg, Wiesbaden, 2005.