Aufgabe 3

Bearbeitungszeit: zwei Wochen (bis Freitag, 27. Mai 2022)

Mathematischer Hintergrund: Berechnung von Eigenwerten/-vektoren mit der Potenzmethode

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

---

Aufgabenstellung

Schreiben Sie ein Programm, das zu einer gegebenen Matrix A den betragsgrößten Eigenwert samt Eigenvektor mit der Potenzmethode bestimmt. Vervollständigen Sie zu diesem Zweck die Vektorklasse, die Ihnen zur Verfügung gestellt wird, und verfassen Sie analog eine Matrixklasse.

Potenzmethode

Gegeben sei eine Matrix $A \in \mathbb{R}^{n\times n}$. Der betragsgrößte Eigenwert λ^* sei eindeutig bestimmt (er ist damit insbesondere reell) und habe die algebraische Vielfachheit 1. Der zugehörige Eigenvektor sei x^*, und es sei k ein beliebig gewählter Index, so dass $x^*_k \neq 0$ gilt. Da der Eigenvektor nur bis auf Vielfache bestimmt ist, kann man o. B. d. A. $x^*_k = 1$ annehmen. Man startet die Potenzmethode mit einem beliebigen Vektor $x^0 \in \mathbb{R}^n \setminus \{0\}$ und iteriert wie folgt:

$$\begin{aligned} \tilde{x}^i &\leftarrow Ax^{i-1}, \\ x^i &\leftarrow \tilde{x}^i/\tilde{x}^i_k. \end{aligned}$$

Die Vektoren x^i konvergieren nun für fast alle Startwerte x^0 (Beweisidee siehe unten) gegen den Eigenvektor x^*, und die Einträge $\tilde{x}^i_k$ konvergieren gegen λ^*. Weil man zu Beginn jedoch nicht weiß, für welche Werte von k die Komponente x^*_k nicht verschwindet, und um zu vermeiden, dass $\tilde{x}^i_k$ zu klein oder gar 0 wird und das Verfahren damit instabil wird oder abbricht, modifiziert man den Algorithmus noch leicht: Man startet mit $k = 1$ und wählt k jeweils neu, wenn $c \cdot |\tilde{x}^i_k| \leq \|\tilde{x}^i\|_\infty$ gilt, und zwar so, dass $|\tilde{x}^i_k| = \|\tilde{x}^i\|_\infty$ ist. Hierbei ist $c > 1$ eine (nicht zu große) Konstante, z. B. $c = 2$. Nach hinreichend vielen Iterationen ändert sich k schließlich nicht mehr, und man erhält Konvergenz wie oben.

Beweisidee

$\mathbb{R}^n$ lässt sich (eindeutig) in die direkte Summe $\mathbb{R}^n = \langle x^* \rangle \oplus V$ zerlegen, wobei V unter A invariant ist, d. h. $Av \in V$ für alle $v \in V$. Damit kann man x^0 eindeutig darstellen als $x^0 = \alpha x^* + v$ mit $\alpha \in \mathbb{R}$ und $v \in V$. Dann gilt $A^i x^0 = (\lambda^*)^i \alpha x^* + A^i v$ bzw.

$$\frac{A^i x^0}{(\lambda^*)^i} = \alpha x^* + \frac{A^i v}{(\lambda^*)^i}.$$

Weil die Eigenwerte von $A|_V$ vom Betrag kleiner als $|\lambda^*|$ sind, konvergiert $\frac{A^i v}{(\lambda^*)^i}$ für $i \to \infty$ gegen 0, der gesamte Ausdruck folglich gegen αx^*. Wenn $\alpha \neq 0$ ist (daher funktioniert der Algorithmus auch nur für fast alle x^0), sind die x^i lediglich beschränkte Vielfache dieses Ausdrucks. Damit konvergieren sie (weil die k–te Komponente auf 1 normiert ist) gegen ein Vielfaches von αx^*, was wieder ein Eigenvektor von A zum Eigenwert λ^* ist. Die Konvergenz der $\tilde{x}^i_k$ gegen λ^* ergibt sich dann sofort. [2, 3]

Vektor- und Matrixklasse

Um den Algorithmus möglichst übersichtlich gestalten zu können, sollen die Vektor- und Matrixoperationen in eigenen Klassen versteckt werden. Neben der Header–Datei `vector.h` zu der Vektorklasse werden Ihnen einige Beispielfunktionen in `vector.cpp` zur Verfügung gestellt. Zusätzlich wird Ihnen die Header-Datei `matrix.h` zur Matrixklasse zur Verfügung gestellt. Ihre Aufgabe ist es, die Funktionen der Vektorklasse zu vervollständigen und analog die Implementierung der Matrixklasse `Matrix` mit den entsprechenden Operationen zu entwickeln. Dazu sollten Sie die Datei `matrix.cpp` anlegen. Einen Ausdruck der Header–Datei `vector.h` finden Sie am Ende des Aufgabenblatts.

Anforderungen an die Matrixklasse

An Ihre Matrixklasse werden einige Anforderungen gestellt. Beachten Sie dazu auch die `matrix.h` Datei. Durch

```
mapra::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 `GetLength()` der Vektorklasse über die beiden Elementfunktionen `GetRows()` und `GetCols()` auslesen lassen. Mit

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

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

```
std::size_t rows_;
std::size_t cols_;
std::vector<double> elems_;
```

Der Konstruktor könnte dann ein entsprechend großes (eindimensionales) Feld reservieren. Ihre Funktionen müssen dafür sorgen, dass die Matrixelemente darin korrekt abgelegt werden. In `rows_` und `cols_` 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.

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 zugrude liegt, sollen ausschließlich in den Zugriffsoperatoren geschehen. Alle anderen Funktionen müssen die Zugriffsoperatoren benutzen, wenn sie Einträge lesen oder schreiben.

[1] Fehlermeldung: `segmentation fault`

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.

Test der Vektor- und Matrixklassen

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

Weiterhin sollen Sie wieder eigene Unit Tests über die MaPraTest-Klasse hinzufügen und damit die Funktionalität Ihrer Klasse prüfen.

Schnittstellen

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

```
std::tuple<mapra::Matrix, mapra::Vector, double> mapra::GetExample(int
    ex_id);
```

aufrufen, wobei `ex_id` im Bereich von `1` bis `num_examples` liegen darf. Sie können dafür die in C++17 eingeführten *Structured binding declarations* benutzen, diese erlauben einen Aufruf wie folgt:

```
auto [A, x, eps] = mapra::GetExample(ex_id);
```

Danach ist `A` ein Objekt vom Typ `mapra::Matrix`, `x` ein Objekt vom Typ `mapra::Vector` und `eps` ein Objekt vom Typ `double`. Das `std::tuple` wird so also automatisch entpackt. Zur Matrix `A` soll dann mit Hilfe der Potenzmethode der betragsgrößte Eigenwert samt Eigenvektor bestimmt werden. Als Startvektor verwenden Sie bitte den Vektor `x0`. Wenn $\|x^i - x^{i-1}\|_\infty \leq$ `eps` und $|\tilde{x}_k^i - \tilde{x}_k^{i-1}| \leq$ `eps` erfüllt sind, können Sie die Iteration abbrechen. Ihr Resultat übergeben Sie zusammen mit der Anzahl der benötigten Iterationen an die Funktion

```
void mapra::CheckSolution(const Vector& eigVec, double eigVal, std::
    int64_t iterations);
```

wo es bewertet und ausgegeben wird.

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 3` das dritte 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[]);
```

[2] engl. *range checking*

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 zeigt, die dem `i`-ten Kommandozeilenparameter entspricht. Also zeigt `argv[0]` auf den String `"loesung"` und `argv[1]` auf den String `"3"`.

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

```
std::istringstream isstream(s);
```

einen Input-String-Stream `isstream` definieren, dessen Inhalt aus der Zeichenkette `s` besteht. Aus diesem können Sie dann Objekte verschiedenen Typs wie aus dem Eingabe-Stream `std::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):

```
CXX      = g++
CXXFLAGS = -std=c++17 -Wall -Wextra -Wpedantic -O2 -fno-inline -fPIC -fPIE

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

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

...
```

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

```
g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 -fno-inline -fPIC -fPIE -o
   test vector.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]

Programmabnahme

Lösen Sie obige Aufgabe, sodass alle Tests der Praktikumsumgebung durchlaufen, und laden Sie anschließend Ihre Quellcode (als Gruppe) im Lernraum hoch. Danach lassen Sie Ihren Code testieren. Damit Sie Ihr Testat bestehen, sollten mindestens folgende Punkte erfüllt sein:

- Ihr Programm kompiliert und unsere vorgegebenen Tests (Test der Vektor-/Matrixklasse mithilfe von `test.o`, Durchlaufen aller Beispiele) werden bestanden.
- Ihr Code ist lesbar und wartbar (siehe dazu Aufgabe 1).
- Sie haben eigene Unit Tests geschrieben, mit denen die Funktionalität Ihres Codes überprüft wird.
- Sie haben ein funktionierendes Makefile erstellt, welches sowohl Ihre Lösung als auch Ihre Unit Tests kompiliert.

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

Literatur

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

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

[3] Golub, G. und C. van Loan: *Matrix Computations.* Johns Hopkins University Press, Baltimore, 2003.

Header–Datei `vector.h` zur Vektorklasse

```
#ifndef VECTOR_H_
#define VECTOR_H_

#include <cstddef>
#include <iostream>
#include <vector>

namespace mapra {

class Vector {
 public:
  explicit Vector(std::size_t len = 1);

  double& operator()(std::size_t);
  double operator()(std::size_t) const;

  Vector& operator+=(const Vector&);
  Vector& operator-=(const Vector&);
  Vector& operator*=(double);
  Vector& operator/=(double);

  Vector& Redim(std::size_t);
  std::size_t GetLength() const;
  double Norm2() const;
  double NormMax() const;

  static void VecError(const char str[]);

  friend Vector operator+(const Vector&, const Vector&);
  friend Vector operator-(const Vector&, const Vector&);
  friend Vector operator-(const Vector&);

  friend double operator*(const Vector&, const Vector&);
  friend Vector operator*(double, const Vector&);
  friend Vector operator*(const Vector&, double);
  friend Vector operator/(const Vector&, double);

  friend bool operator==(const Vector&, const Vector&);
  friend bool operator!=(const Vector&, const Vector&);

  friend std::istream& operator>>(std::istream&, Vector&);
  friend std::ostream& operator<<(std::ostream&, const Vector&);

 private:
  std::vector<double> elems_;
};

}  // namespace mapra

#endif  // VECTOR_H_
```