Universität der Bundeswehr München

Ingenieur Mathematik

Numerische Rechneranwendungen

— Skript zur Vorlesung —

Jürgen Pannek

Lehrstuhl Mathematik und Rechneranwendungen
Fakultät für Luft- und Raumfahrttechnik
Universität der Bundeswehr München

Vorwort

Die Hauptziele der Vorlesung „Numerische Rechneranwendungen“ und der zugehörigen Übungen sind zum einen eine gründliche Einführung in `Matlab` zu geben (Bereich 1), numerische und algorithmische Fehlerquellen kennen und unterscheiden zu lernen (Bereich 2), einfach Algorithmen selbst implementieren (Bereich 3) und bereits implementierte Algorithmen in kleineren Projekten kombinieren zu können (Bereich 4). Im Rahmen der Vorlesung ist dabei geplant, folgende Themen aus den Vorlesungen „Numerische Mathematik I und II“ aufzugreifen:

Datum	Thema
12.01.2012	Einführung in `Matlab`
26.01.2012	Direkte Verfahren zur Lösung linearer Gleichungssysteme
	Ausgleichsrechnung
09.02.2012	Eigenwertberechnung
	Nichtlineare Gleichungssystemlöser
23.02.2012	Interpolationsverfahren
	Integrationsverfahren
08.03.2012	Differentialgleichungslöser

Die dabei vorgestellten Algorithmen wurden in der „Numerischen Mathematik I“ bereits behandelt bzw. werden in der „Numerischen Mathematik II“ parallel vorgestellt, so dass auf Theorie weitestgehend verzichtet wird. Die einzelnen Übungen orientieren sich dabei an den in der Vorlesung vorgestellten bzw. wiederholten Themen. Dabei ist geplant, diese Themen wie folgt den Hauptzielen der Vorlesung zuzuordnen:

Datum	Bereich	Thema
18./19.01.2012	1	Einführung in `Matlab`
		`Matlab` Hilfe
		Dokumentation
		Sauberes Programmieren und Kommentieren
25./26.01.2012	2	Zahldarstellung
		Laufzeit
		Robustheit
		Stabilität
01./02.02.2012	3	Direkte Verfahren zur Lösung linearer Gleichungssysteme
08./09.02.2012	3	Ausgleichsrechnung
15./16.02.2012	3	Eigenwertberechnung
22./23.02.2012	4	Nichtlineare Gleichungssystemlöser
29./01.03.2012	3	Interpolationsverfahren
07./08.03.2012	4	Integrationsverfahren
14./15.03.2012	4	Differentialgleichungslöser
21./22.03.2012	3	Kommentierungstool in Matlab

Inhaltsverzeichnis

Kapitel 1

Matlab Einführung

1.1 Was heißt Matlab und was ist Matlab?

MATLAB steht für **Mat**rix-**Lab**oratory und ist ein Programmsystem zur einfachen und primär interaktiven Berechnung von (wissenschaftlichen) Problemen. Wie der Name bereits verrät, liegen die Ursprünge von Matlab in der Matrizenrechnung und der linearen Algebra. Seit geraumer Zeit stehen aber auch vielfältige und sehr leistungsfähige Algorithmen für analytische Probleme und zur Erzeugung von Graphiken zur Verfügung. Der Schwerpunkt von MATLAB liegt dabei auf den Methoden des numerischen Rechnens. Zwar bietet MATLAB Erweiterungen an, sogenannte *Toolboxes*, in denen symbolische Methoden zur Verfügung stehen. Diese sind aber weniger komfortabel zu bedienen als beispielsweise in MAPLE, weswegen wir uns hier auf die Beschreibung der numerischen Routinen beschränken.

1.2 Wie startet man Matlab?

Auf den Windows-Rechnern im PC-Pool über *START* > *Alle Programme* > *MATLAB* > *R2010b* > *MATLAB R2010b*[1].

Es wird eine Programmobfläche gestartet, die gleich den Interpreter mitbringt. Hier können direkt erste Kommandos eingegeben werden:

```
>> 1*2/(3+4)-5
>> b=5
>> B=6;
>> A=[b B]
```

MATLAB unterscheidet dabei zwischen Groß- und Kleinschreibung in Variablen. Dabei ist es sinnvoll, sich von Anfang an gewisse Programmierkonventionen anzueignen, etwa Großbuchstaben für Matrizen, Kleinbuchstaben für Vektoren und Skalare und die sogenannte *CamelCase Notation*. Bei Letzterer wird die Bezeichnung einer Variablen durch eine Beschreibung ersetzt, die komplett zusammengeschrieben ist und neue Worte jeweils mit Großbuchstaben beginnen, z.B.

```
>> VektorDerNebenbedingungen=b;
```

[1] Oder entsprechende Unterverzeichnisse bei anderen Versionen

1.3 Funktionen in Matlab

Funktionen sind in MATLAB nicht im Sinne mathematischer Operatoren sondern im Sinne einer Funktionalen Programmiersprache zu verstehen. Allgemein gilt, dass Funktionen im Command Window aufgerufen und ausgeführt werden können. Hierzu ist auch die History Funktion sehr nützlich. Diese erlaubt es mit den Cursortasten nach oben/unten alte Befehle wieder aufzurufen und diese dann zu editieren und neu auszuführen. Zudem können bei der Suche einige Anfangsbuchstaben eingeben werden, um die Auswahl einzuschränken. Zwar können MATLAB Operationen im sogenannten "*Command Window*"direkt eingegeben werden, im Normalfall jedoch sollten Funktionen in Programmdateien, den sogenannten *m–Files*, gespeichert werden.

1.3.1 m–Files

Die folgende Funktion ist ein Beispiel für die Implementierung der mathematischen Funktion $x \mapsto e^x \cdot \sin(x)$ in MATLAB:

```
1 % Funktion zur Auswertung von y=exp(x)*sin(x)
2 % Beispielaufruf: y=expsin(2)
3 function y = expsin(x)
4     y = exp(x).*sin(x);
```

Programm 1.1: Funktion zur Berechnung von $e^x \cdot \sin(x)$

Hierbei legt die erste Zeile fest, dass es sich um eine Funktion mit Eingabeparameter x und Ausgabeparamter y handelt. Die zweite Zeile gibt die Rechenregel für y an, also $y = e^x \cdot \sin(x)$. Das Semikolon ";"nach der Zeile bewirkt, dass die unmittelbare Ausgabe des Ergebnisses unterdrückt wird. Eine MATLAB Funktion muss dabei nicht auf eine Zeile beschränkt sein, sondern kann aus vielen Codezeilen bestehen.

Speichert man nun dieses m–File unter dem Namen `expsin.m`, so kann die so definierte Funktion im Command Window oder von einem andere m–File aus aufgerufen werden. Hierzu sind zwei Fakten wichtig:

- Der Name des m–Files legt das Aufrufkommando fest. Im vorliegenden Fall heißt sowohl die Funktion als auch das m–File `expsin`. Diese beiden Namen sollten aus folgendem Grund übereinstimmen: Eine Funktion, die im m–File `funktion1.m` gespeichert ist die aber per Deklaration `funktion2` heißt, kann nur unter dem Namen `funktion1` aufgerufen werden. Ein Aufruf `funktion2` kann nicht ausgeführt werden (es sei denn, ein m–File mit solchem Namen existiert zusätzlich). Beachte, dass die Speicherung der Funktion `funktion1` im m–File `funktion2.m` kein Fehler ist und es entsprechend keine Fehlermeldung gibt. Eine derartige Abspeicherung ist z.B. von Nutzen, wenn Unterfunktionen implementiert werden, die nicht aufrufbar sein sollen.

- Das Verzeichnis, in dem das m–File abgelegt ist, muss MATLAB bekannt sein. Bei einem Aufruf des m–Filenamens z.B. im Command Window sucht MATLAB zunächst im aktuellen Verzeichnis nach einer solchen Datei und anschließend den sogenannten MATLAB *Pfad*. Ausführbarkeit eines m–Files kann man also am einfachsten dadruch erreichen, dass man mit

```
>> cd Ordnername
```

in das Verzeichnis wechselt, in dem sich das m–File befindet. Diese Methode hat aber den Nachteil, dass MATLAB das Verzeichnis bei Eingabe eines neuen Verzeichnisses oder nach einem Neustart wieder "vergisst". Zudem kann dadurch ein m–File in Verzeichnis `ordner1` nur unter der vollen relativen Pfadangabe ein m–File in Verzeichnis `ordner2` aufrufen. Um derartige Situationen zu vermeiden, können mit *File > Set Path* dem sogenannten *Path* bestimmte Verzeichnisse hinzugefügt werden. Wird der PFad anschließend im *Set Path* Menu mit *Save* gespeichert, so ist das angegebene Verzeichnis auch nach einem Neustart von MATLAB bekannt.
Die dabei entstehende Datei "`pathdef.m`"muss im aktuellen Verzeichnis liegen, von dem aus MATLAB gestartet wird, oder in einem der Verzeichnisse, die MATLAB vordefiniert bekannt sind. Diese Verzeichnisse sind von Installation zu Installation und bei verschiedenen Betriebssystemen unterschiedlich und können mittels des Befehls

```
>> matlabpath
```

kontrolliert werden.

1.3.2 Funktionen mehrerer Variablen

Funktionen können in MATLAB mehrere Ein- und Ausgabeparameter besitzen. Ein Beispiel hierfür bietet die folgende Funktion, die die beiden Lösungen einer quadratischen Gleichung $x^2 + px + q = 0$ berechnet:

```
1 % Funktion zur Berechnung der zwei Loesungen der quadratischen Gleichung
2 % x^2+px+q=0
3 % Beispielaufruf: [a,b]=quadgleichung(0,-1)
4 function [x1,x2] = quadgleichung(p,q)
5     x1 = -p/2 + sqrt(p^2-4*q)/2;
6     x2 = -p/2 - sqrt(p^2-4*q)/2;
```

Programm 1.2: Funktion zur Berechnung der Nullstellen einer quadratischen Funktion

Bei einem Aufruf

```
>> quadgleichung(0,-1)
```

wird dabei nur der erste Ausgabeparameter am Bildschrim ausgegeben. Um beide Parameter zu erhalten, muss die Funktion mittels

```
>> [a,b]=quadgleichung(0,-1)
```

aufgerufen werden. Das Ergebnis ist dann der Zeilenvektor `[a,b]`. Der Aufruf

```
>> [a,b]=quadgleichung(0,1)
```

zeigt übrigens, dass MATLAB auch komplexe Arithmetik beherrscht.

1.3.3 Unterfunktionen und Scripts

In einem m–File können mehrere Funktionen nacheinander definiert werden. Dabei ist aber nur die erste Funktion nach "außen" sichtbar. Alle weiteren Funktionen können nur von Funktionen innerhalb des m–Files aufgerufen werden.

Umgekehrt ist es auch möglich ein m–File ganz ohne Funktionen zu schreiben, ein sogenanntes *Script*. Dies wird dann als einfach Anweisungsfolge ohne Ein- und Ausgabeparameter interpretiert und ausgeführt:

```
>> scriptxy
>> help scriptxy
```

Das Kommando `help scriptxy` zeigt dabei (soweit vorhanden) die ersten Dokumentationszeilen des Scripts an.

1.3.4 Funktionen als Funktionsparameter

Oft ist es sinnvoll, eine Funktion an eine andere Funktion zu übergeben. Wir werden am Ende dieses Kapitels eine kleine Anwendung einer solchen Konstruktion kennenlernen. Hier woll wir nur eine Funktion schreiben, die eine übergebende Funktion f and zwei Stellen a und b auswertet. Diese lautet:

```
% Funktion zur Auswertung einer abstrakten Funktion f an den Stellen a und
% b
% Beispielaufruf: [y1,y2] = auswertung(@sin,0,1)
function [y1, y2] = auswertung(f,a,b)
    y1 = f(a);
    y2 = f(b);
```

Programm 1.3: Funktion zur Auswertung einer abstrakten Funktion f an den Stellen a und b

Wie der kommentierte Beispielaufruf zeigt, muss für den Aufruf die zu übergebende Funktion mit einem vorangestellten "@"versehen werden. Hierzu können nicht nur in MATLAB vordefinierte Funktionen wie Sinus mittels des Aufrufs

```
% Auswertung der Sinus Funktion an den Stellen 0 und 1 mit Hilfe der
% Funktion 'auswertung'
[y1, y2] = auswertung(@sin,0,1)
```

Programm 1.4: Auswertung der Sinus Funktion an den Stellen 0 und 1 mit der Funktion `auswertung`

verwendet werden. Vielmehr können auch Zeiger auf Funktionen übergeben werden, die in anderen m–Files gespeichert sind wie etwa die zuvor definierte Funktion `expsin` :

```
% Auswertung der Funktion expsin an den Stellen 0 und Pi mit Hilfe der
% Funktion 'auswertung'
[y1, y2] = auswertung(@expsin,0,pi)
```

Programm 1.5: Script zur Auswertung der Funktion `expsin` an den Stellen 0 und π mit der Funktion `auswertung`

Man beachte, dass bei Aufruf der zweiten Auswertung der zweite Ausgabewert (im Normalfall) eine kleine Zahl ungleich Null ist. Dadurch wird deutlich, dass MATLAB nicht standardmäßig symbolisch sondern mit Fließkommazahlen rechnet. Auf diese werden wir in Kapitel 2.2 genauer eingehen. Der Zahlenwert von π kann durch den vordefinierten Wert `pi` abgerufen werden.

1.4 Vektoren und Matrizen

Vektoren und Matrizen sind zwei wesentliche Datenstrukturen in MATLAB. Das folgende Script zeigt die grundlegenden Operationen für Matrizen und Vektoren:

```
% Script zur Demonstration grundlegender Operationen mit Vektoren und
% Matrizen

% Definition von Matrizen:
% Matrizen kann man einfach zeilenweise "per Hand" eingeben. Eine Matrix
% wird in eckige Klammern [] eingeschlossen. Ein Spaltenwechsel wird durch
% einen Leerbereich (Leerzeichen oder auch Tabulator) oder ein Komma
% erzeugt. Eine neue Zeile wir durch einen Zeilenabsatz oder einen
% Strichpunkt generiert:
A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]

% Oder, wenn man eine Matrix "leer" definieren moechte, z.B. um
% darin ein Rechenergebnis abzulegen
B = zeros(4,3)

% Spaltenvektoren gibt man einfach als einspaltige Matrizen ein
b = [3; 4; 5]

% und Zeilenvektoren als einzeilige Matrizen
c = [3 4 5]

% ob man Eintraege mit Leerzeichen oder Komma trennt, macht
% keinen Unterschied
d = [3, 4, 5]

% MATLAB kann Matrizen transponieren
A'

% und Matrizen mit Matrizen oder Vektoren multiplizieren
C = [6 5; 4 3; 2 1]

A*C
A*b

% Man kann auf einzelne Elemente einer Matrix oder eines Vektors
% einzeln zugreifen
A(1,1)
A(2,3)
b(3)

% und ebenso auf einzelne Zeilen
A(2,:)

% oder Spalten
A(:,3)

% Diese kann man nicht nur auslesen, sondern auch zuweisen
A(:,2) = [1; 2; 3; 4]

% Man kann (Zeilen)-Vektoren mit abgezaehlten Eintraegen auch mit einer
% Abkuerzung definieren. Hierbei gibt man den ersten Eintrag,
% die Schrittweite und den letzten Eintrag, jeweils durch Doppelpunkt
% getrennt an. Z.B.: Erzeuge den Vektor [1 2 3 4 5]
t =  [1:1:5]

% Schliesslich kann man nicht nur Matrizenrechnung direkt ausfuehren,
% sondern auf Vektoren und Matrizen auch komponentenweise rechnen
% Hierzu setzt man einen "." vor den entsprechenden mathematischen
```

```
% Operator. Um z.B. jedes Element des obigen Vektors t mit sich selbst
% zu multiplizieren, berechnet man
t.*t

% Beachte, dass t*t aus Dimensionsgruenden eine Fehlermeldung ergibt
```

Programm 1.6: Script zur Demonstration grundlegender Operationen mit Vektoren und Matrizen

Dabei ist darauf zu achten, dass es in MATLAB für Multiplikation, Division und Potenzieren je zwei verschiedene Varianten gibt, nämlich `a*b`, `a/b` und `a^b` sowie `a.*b`, `a./b` und `a.^b`. Dies macht dann einen Unterschied, wenn es sich bei `a` und/oder `b` um Vektoren oder Matrizen handelt: im ersten Fall wird stets versucht, entsprechende Matrizenoperationen durchzuführen, während die mit dem Punkt versehenen Operationen stets komponentenweise zu verstehen sind. Wenn nicht explizit Matrizenrechnung verwendet werden soll, empfiehlt es sich in MATLAB stets die mit dem Punkt versehenen Operatioren zu verwenden.

Natürlich können Vektoren auch als Rückgabewerte von Funktionen verwendet werden. Eine vektorwertige Variante der Funktion `quadgleichung` lautet:

```
% Funktion zur Berechnung der zwei Loesungen der quadratischen Gleichung
% x^2+px+q=0
% Beispielaufruf: x=quadgleichungvector(0,-1)
function x = quadgleichungvector(p,q)
    x = zeros(1,2);
    x(1) = -p/2 + sqrt(p^2-4*q)/2;
    x(2) = -p/2 - sqrt(p^2-4*q)/2;
```

Programm 1.7: Funktion zur Berechnung der Nullstellen einer quadratischen Funktion

Die `zeros` Anweisung ist bei der Funktion 1.7 nicht unbedingt nötig. Es empfiehlt sich jedoch diese mit aufzunehmen, da die explizite Deklaration eines Vektors effizientere Speicherverwaltung und damit auch schnellere Ausführung des Programms ermöglicht.

1.5 Grafik

MATLABs Standardanweisung zum Erzeugen von Grafiken in $\mathbb{R}^2$ is `plot`. Hierbei ist zu beachten, dass MATLAB Plots in der Regel Punktplots sind. Folgerichtig akzeptiert `plot` auch nur diskrete Daten in Form von Vektoren als Eingangsargumente. Diese werden dann entweder als Polygonzug oder als Punktgraph dargestellt. Einen Überblick über einfache Anwendungen von `plot` gibt das folgende Script:

```
% Script fuer drei elementare Anwendungen des plot-Befehls

% Beispiel 1: Plotten von Daten

% Wir definieren einen kleinen Datensatz mittels zweier Vektoren:
t  = [1; 2; 3; 4; 5]
m  = [0.9; 3.8; 7.9; 15; 26.7]

% Mit der "plot" Anweisung kann man nun die Daten gegeneinander
% grafisch darstellen
plot(t,m)

```

```
13 input('Druecke␣RETURN');
14
15 % Statt eines Polygonzugs kann man die Daten auch mit Punkten
16 % darstellen:
17 plot(t,m,'.')
18
19 input('Druecke␣RETURN');
20
21 % Statt dem Punkt '.' kann man auch viele andere Symbole verwenden,
22 % z.B. ein Kreuz 'x':
23 plot(t,m,'x')
24
25 input('Druecke␣RETURN');
26
27 % Beispiel 2: Plotten von Funktionen
28
29 % Funktionswerte, die mit "plot" grafisch dargestellt werden sollen,
30 % muessen in einen Vektor umgewandelt werden. Dazu definiert man zunaechst
31 % einen Vektor mit den Stuetzstellen, an denen die Funktion ausgewertet
32 % werden soll, hier das Intervall von 1 bis 5 mit einem Abstand von
33 % 0.1 zwischen je zwei Stuetzstellen
34 tt = [1:0.1:5];
35
36 % Dann weist man einem weiteren Vektor die Werte der Funktion
37 % (hier f(t)=t^2) zu. Beachte den '.' vor dem mathematischen Operator,
38 % der bewirkt, dass die Operation komponentenweise im Vektor tt
39 % ausgefuehrt wird.
40 y  = tt.^2;
41
42 % Jetzt koennen wir plotten. Der Strich '-' ist die explizite Anweisung,
43 % dass der Graph als Polygonzug dargestellt werden soll
44 plot(tt,y,'-')
45
46 input('Druecke␣RETURN');
47
48 % Beispiel 3: Gemeinsames Plotten von Daten und Funktionen
49
50 % Hierzu ist nichts weiter zun tun, als die einzelnen Argumente
51 % nacheinander in den plot Befehl zu schreiben
52 plot(t,m,'x',tt,y,'-')
```

Programm 1.8: Script für drei elementare Anwendungen des `plot` Befehls

Anhand des Scripts 1.8 sieht man, dass Funktionen mittels eines Vektors ausgewertet werden müssen, damit man sie mit `plot` darstellen kann. Damit dies auch für selbstgeschriebene Funktionen wie die Funktion `expsin`, vergleiche Funktion 1.1, müssen hier intern die mit dem Punkt versehenen Operatoren verwendet werden.

Die explizite Vektorauswertung ist etwas umständlicher und wird in den übungen (vergleiche Blatt 1, Aufgabe 7) besprochen. Die MATLAB interne Funktion `fplot` leistet im Wesentlichen dasselbe. Für mehr Informationen zu dieser Funktion wie auch zu allen anderen Funktionen, die in diesem Kapitel und folgenden Kapiteln angesprochen werden, kann und sollte die MATLAB Hilfe herangezogen werden.

Kapitel 2

Numerik und Numerische Probleme

2.1 Was ist Numerik?

Gegenstand der *numerischen Mathematik* (oder Numerik) ist die konkrete, konstruktive Berechnung einer Lösung mathematischer Probleme durch Zahlenwerte. Die Lösungsberechnung erfolgt dabei durch einen *Algorithmus*, d.h. durch eine Folge von *elementaren Anweisungen* und *Rechenoperationen*. Ein solcher Algorithmus stützt sich oft auf Ergebnisse der reinen Mathematik und reflektiert mathematische Eigenschaften des Problems. Die zu behandelnden Probleme stammen dabei oft nicht aus der Mathematik selbst, sondern aus Ingenieur–, Natur– und Wirtschaftswissenschaften.

Derartige realistische Aufgabenstellungen lassen sich im Regelfall nicht oder nur mit übermäßigem Aufwand durch analytische Methoden lösen und erfordern fast immer den Einsatz eines Rechners.

Beispiel 2.1 (Scherwinde beim Landeanflug eines Flugzeuges)
Beim Landeanflug eines Flugzeuges treten ab und zu sogenannte Scherwinde auf (vergleiche Abbildung 2.1), wurch es zu zwei bis drei Unfällen pro Jahr kommt.

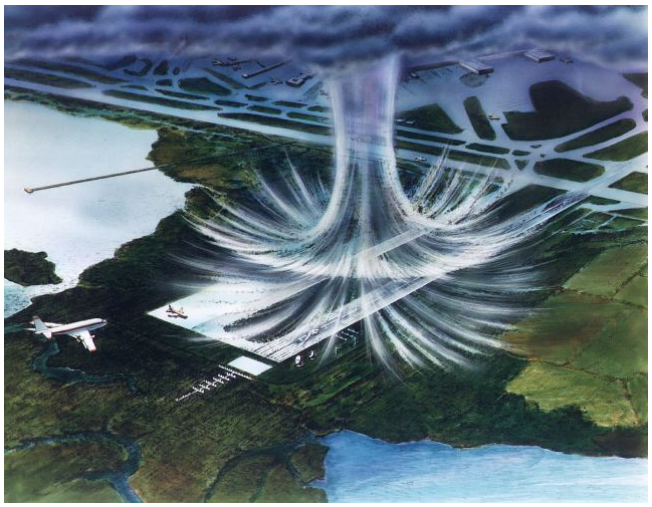


Abbildung 2.1: Scherwinde beim Landeanflug

Um einen Absturz zu vermeiden muss entschieden werden können, ob der Landeanflug abgebrochen werden sollte. Ein Kriterium hierfür ist die maximale Minimalflughöhe, die immer oberhalb einer Sicherheitsschranke $h_{\min}$ liegen sollte, vergleiche Abbildung 2.2.

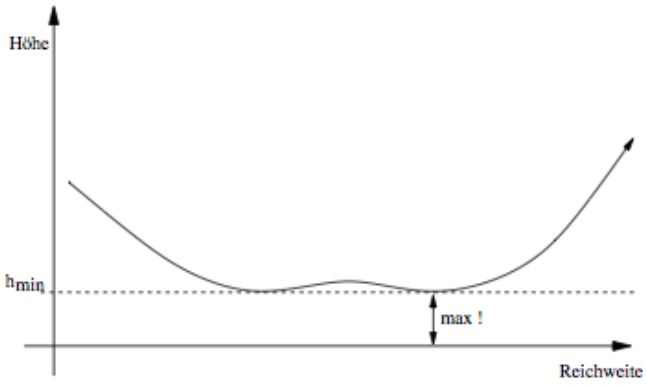


Abbildung 2.2: Maximierung der minimalen Flughöhe bei Scherwinden

Die Physik sowohl des Flugzeugs als auch der Scherwinde ist sehr gut bekannt, wodurch ein sehr realitätsnahes mathematisches Modell zur Verfügung steht. Zur Lösung des Problems sollte sowohl das Problem an sich als auch mögliche Lösungsalgorithmen zunächst ***theoretisch untersucht*** *bzw. neu entwickelt werden. In diesem Fall kann die sogenannte Variationsrechnung bzw. optimale Steuerung herangezogen werden. Zur Anwendungen kommen dabei Differentialgleichungslöser sowie lineare und nichtlineare Gleichungssystemlöser. Die hierfür bekannten Algorithmen sind jedoch für das praktische Problem* ***nicht mehr analytisch*** *auswertbar, wodurch eine* ***numerische Lösung*** *mittels eines Computers notwendig wird.*

Wenn ein Rechner zum Einsatz kommt, so ist zu beachten, dass im Allgemeinen

- die zur Auswahl stehenden Anweisungen und Rechenoperationen im Allgemeinen auf die vier Grundrechenarten sowie die sechs arithmetischen Vergleiche beschränkt sind und
- die verfügbare Menge an Zahlen durch die Nutzung von Gleitpunkt– oder Fixpunktzahlen endlicher Stellenzahl limitiert ist.

Die zweite Restriktion hat zur Folge, dass alle Zwischen– und Endergebnisse auf darstellbare Zahlen gerundet werden müssen, die wir in Abschnitt 2.2 genauer betrachten. Entsprechend treten bei der Nutzung von Computern **Rundungsfehler** auf.

Wie an Beispiel 2.1 erkennen kann, werden an die Numerik selbst vielschichtige Anforderungen gestellt:

- Entwicklung von Verfahren zur Konstruktion von (Näherungs–)Lösungen
- Auswahl geeigneter Verfahren
- Aussagen über Güte der Approximation
- Effiziente Implementierung

Die Kunst und die Tücken liegen dabei im Detail: Zum einen gibt es für manche Problemstellungen — z.B. der Lösung eines linearen Gleichungssystems — Algorithmen, die diese Probleme im Prinzip exakt lösen. Diese Art Algorithmen werden als *direkte Verfahren* bezeichnet. Da diese Verfahren aber für praktische Probleme nicht mehr per Hand durchführbar sind, treten die bereits beschriebene Rundungsfehler auf. Alternativ existieren sogenannte *iterative Verfahren* — etwa Differenzenquotienten zur Approximation von Ableitungen —, die per Konstruktion lediglich Näherungslösungen bestimmen. Derartige Algorithmen werden akzeptiert, wenn deren

sogenannter **Verfahrensfehler** abgeschätzt und durch Steigerung des Berechnungsaufwands im Prinzip beliebig klein gemacht werden kann. Bei der Nutzung von Rechnern treten jedoch wie bei den direkten Verfahren Rundungsfehler auf.

Bei praktischen Problemen ergibt sich dabei die Kette von Einzelaufgaben

Problemstellung
↓
Physikalisches Modell
↓
Mathematisches Modell
↓
Mathematische/numerische Lösung
↓
Diskussion der Ergebnisse

die im Allgemeinen mehrfach durchlaufen werden muss. Besondere Schwierigkeiten machen sogenannte **Modellfehler**, also Abweichungen des mathematischen Modells von der Realität. Ebenso müssen sogenannte **Datenfehler** berücksichtigt werden, die etwa durch falsche Messwerte entstehen.

Zusammenfassend sind folgende Fehlerquellen zu beachten:

Fehlerquelle	Erklärung anhand von Beispiel 2.1
Rundungsfehler	Der Computer kann nicht mit beliebig vielen Nachkommastellen rechnen.
Verfahrensfehler	Das numerische Verfahren berechnet zur Lösung der Differentialgleichung oder zur Lösung der linearen und nichtlinearen Gleichungssysteme liefert nur einen genäherte Lösung.
Datenfehler	Parameter des Differentialgleichungssystems oder Anfangswerte sind nur ungenau gegeben.
Modellfehler	Die Modellierung ist nicht beliebig genau.

Im Rahmen der Vorlesung werden wir mit Ausnahme der Modellfehler all diese Aspekte untersuchen.

2.2 Rundungsfehler

Wie bereits festgestellt, können bei der Anwendung numerischer Methoden auf Rechnern Rundungsfehler auftreten. Hierbei ist es wichtig zu wissen, dass diese Fehler in der Analysis der Algorithmen nicht auftauchen, da hier die aus der Mathematik bekannten Zahlenmengen $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ und $\mathbb{C}$ verwendet werden.

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{1,2}_{\in\mathbb{N}},0,-1}_{\in\mathbb{Z}},1.25,\frac{1}{10}}_{\in\mathbb{Q}},\sqrt{2},\pi}_{\in\mathbb{R}},i,1-i}_{\in\mathbb{C}}$$

Bereits die „kleinste“ hier genannte Menge $\mathbb{N}$, die in allen anderen Zahlenmengen enthalten ist, hat unendlich viele Elemente. Da bisherige Digitalrechner aber nur einen endlichen Speicher

besitzen ist bereits klar, dass es mit derzeitigen Computern nicht möglich ist alle Zahlen darzustellen. Um Rundungsfehler genauer untersuchen zu können, müssen wir uns zunächst mit der Struktur der Zahlendarstellung auf einem Computer vertraut machen. Für tiefergehende Details siehe [4].

2.2.1 Maschinenzahlen

Die gebräuchlichste Codierungsform ist die sogenannte **Gleitpunkt–Darstellung**.

Definition 2.2 (Normalisierte t–stellige Gleitpunktzahlen zur Basis B)
Die Menge der t–stelligen Gleitpunktzahlen zur Basis B ist gegeben durch

$$\mathbb{G}_{B,t} := \left\{ M \cdot B^E \mid S = 0 \vee B^{t-1} \leq |S| < B^t \text{ mit } S, E \in \mathbb{Z} \right\}.$$

Dabei wird $B \in \mathbb{N} \setminus \{1\}$ als Basis, $t \in \mathbb{Z}$ als Stellenzahl oder Mantissenlänge, $M \in \mathbb{Z}$ als Signifikand oder Mantisse und $E \in \mathbb{Z}$ als Exponent bezeichnet. Wenn $0 \neq g = S \cdot B^E \in \mathbb{G}_{B,t}$, dann sind B, T, S und E eindeutig festgelegt.

Anschaulich gesprochen bestehen die Zahlen $g \in \mathbb{G}_{B,t}$ aus folgenden Bestandteilen eines gespeicherten Vorzeichens, t Ziffern der Darstellung von M sowie der Größe E, die den Abstand zum Komma angibt.

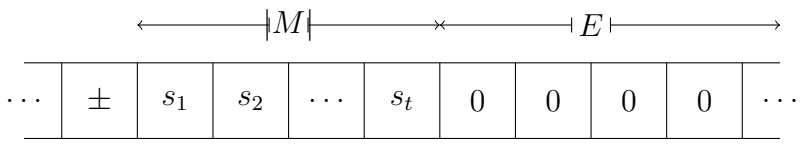


Abbildung 2.3: Zahldarstellung

Limitiert man nicht nur die Anzahl der Stellen von M (wie in Definition 2.2), sondern auch den Exponenten, dann erhält man die Menge der darstellbaren Zahlen:

Definition 2.3 (Maschinenzahlen)
Die von einer Maschine darstellbaren Zahlen sind durch die Menge

$$\mathbb{M}_{B,t,\alpha,\beta} := \{ g \in \mathbb{G}_{B,t} \mid \alpha \leq E \leq \beta \}$$

gegeben.

Bemerkung 2.4
*Bei Rechnern sind die Parameter B, t, α und β durch die gewählte Darstellung festgelegt. Die Darstellung unterscheidet sich von System zu System und muss daher berücksichtigt werden. Hierbei unterscheidet man zwischen sogenannten **Big–Endian** und **Little–Endian** Systemen. Big–Endian Systeme sind z.B. SUN, PowerPC oder IBM Mainframes, Little–Endian dagegen sind z.B. heutige PCs und Laptops (alle x86 kompatiblen Systeme). Vorsicht ist bei ARM Systemen, z.B. in Smartphones oder Tablets, geboten, da diese sowohl Little– als auch Big–Endian sein können. Big–Endian bedeutet, dass das Byte mit dem höchstwertigen Bit, d.h. die signifikanteste Stelle) zuerst gespeichert wird, also die kleineste Speicheradresse hat. Umgangssprachlich kann man das mit der Angabe Stunde:Minute:Sekunde vergleichen. Little–Endian speichert dies genau andersherum, wie es etwa bei Datumsformat Tag:Monat:Jahr der Fall ist.*

Da die Mantissenlänge in Definition 2.2 beschränkt ist, ergibt sich die sogenannte **Auflösung** zu

$$\varrho := \max\left\{\left|\frac{g-g'}{g}\right| \mid g, g' \neq 0 \text{ und } g,\, g' \text{ Nachbarn in } \mathbb{G}_{B,t}\right\} = B^{1-t}.$$

In $\mathbb{M}_{B,t,\alpha,\beta}$ erhält man zudem

kleinste positive Maschinenzahl	$x_{\min} := B^{t-1} \cdot B^{\alpha}$
größte Maschinenzahl	$x_{\max} := (B^t - 1) \cdot B^{\beta}$

In einem Digitalrechner stehen zur Speicherung von Informationen lediglich zwei Zustände zur Verfügung, die der Einfachheit halber 0 und 1 bezeichnet werden. In der Praxis können 0en und 1en z.B. magnetisch oder optisch realisiert werden. Die kleinste mögliche Speichereinheit in einem Digitalrechner wird als Bit bezeichnet und bildet die Grundlage jeder Codierung, also auch der Zahlcodierung. Daher wird als Basis B in Rechner im Allgemeinen $B = 2$ verwendet, das sogenannte Dualsystem. Andere Beispiele sind das Dezimalsystem mit $B = 10$, wie man es bei arabischen Zahlen findet, oder das Hexadezimalsystem mit Basis $B = 16$, das in seltenen Fällen auch bei Rechnern zum Einsatz kommt. Im Folgenden untersuchen wir die Norm ANSI/IEEE Std 754-1985 genauer, die für heutige Rechner gilt.

2.2.2 Norm ANSI/IEEE Std 754-1985

Die Zahlcodierung richtet sich nach der Norm ANSI/IEEE Std 754-1985 „IEEE Standard for Binary Floating–Point Arithmetic for Microprocessor Systems“, für Details vergleiche [1]. Hierbei stehen 64 Bits bzw. 8 Bytes zur Codierung einer Double–Zahl[1] zur Verfügung. Davon wird 1 Bit zur Speicherung des Vorzeichens s, 11 Bits zur Codierung des Exponenten e und 52 Bits zur Codierung der Mantisse m benutzt und der Biaswert $b := 1023$ benutzt. Damit ergibt sich die Darstellung

$$x = (-1)^s \cdot m \cdot 2^e = (-1)^s \cdot \left(1 + \frac{M}{2^{52}}\right) \cdot 2^{E-b}$$

mit der Normalisierung $1 \leq m = 1 + \frac{M}{2^{52}} < 2$. Dabei ist M die vorzeichenlose 52 Bit codierte Mantisse aus Definition 2.2 zur Basis 2. Die Mantisse enthält also nur die Zahlen hinter dem Komma. Für den Exponenten $e = E - b$ werden dagegen nur die Vorkommastellen gespeichert, wobei E in 11 Bits abgelegt ist.

Beispiel 2.5
Die Zahldarstellung

$$x = \Big(\underbrace{0}_{\textit{Vorzeichen}}\ \underbrace{10000000000}_{\textit{Exponent}}\underbrace{0100000000000000000000000000000000000000000000000000}_{\textit{Mantisse } M}\Big)_{(\textit{double})}$$

in Big–Endian ergibt

Vorzeichen:	$(-1)^0 = 1$
Mantisse:	$m = 1 + \frac{0\cdot 2^{51} + 1\cdot 2^{50} + 0\cdot 2^{49} + \ldots + 0\cdot 2^0}{2^{52}} = 1 + \frac{1}{4} = 1.25$
Exponent:	$e = (1 \cdot 2^{10} + 0 \cdot 2^9 + \ldots + 0 \cdot 2^0) - 1023 = 1$

[1] Zahl mit doppelter Genauigkeit, Überbleibsel aus FORTRAN Programmierung

und somit $x = 1 \cdot 1.25 \cdot 2^1 = 2.5$.

Der Sonderfall 0 wird im „IEEE 754“-Standard durch $0 := (-1)^s \cdot m \cdot 2^{0-1023}$ repräsentiert. Dabei wird bewußt zwischen ± 0 unterschieden. Auf weitere Sonderfälle soll hier nicht eingegangen werden.

Wie wir bereits festgestellt haben, existieren in $\mathbb{M}$ eine kleinste positive Maschinenzahl $x_{\min}$ und eine größte Maschinenzahl $x_{\max}$. Zwischen diesen beiden Zahlen sind die Maschinenzahlen aber nicht gleichmäßig verteilt, jedoch treffen folgende zwei Tatsachen zu:

- Innerhalb eines Intervalls $[B^{k-1}; B^k)$ liegen die Maschinenzahlen in gleichen Abständen

  $$a \cdot B^k \text{ mit } a = B^{-t} = .00...01 \cdot p^0,$$

  wobei a die Einheit der letzten/erste (Big/Little Endian) Mantissenstelle beim Exponenten 0 bezeichnet.

- Der relative Abstand zweier aufeinander folgender Maschinenzahlen

  $$\frac{x_{i+1} - x_i}{x_i}, \qquad x_i \neq 0$$

  variiert um höchstens einen Faktor B^{1-t}, die zuvor genannte Auflösung.

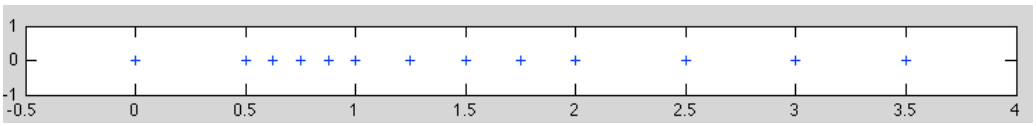


Abbildung 2.4: Zahlverteilung von $\mathbb{M}$ für $B = 2$, $t = 2$, $\alpha = -1$ und $\beta = 1$ (bzw. $m = e = b = 2$)

Es ist also möglich hiermit einen Teil der rationalen Zahlen $\mathbb{Q}$ darzustellen. Allerdings ist beispielsweise die Zahl $0.1 = \frac{1}{10}$ keine Maschinenzahl, läßt sich also nicht mit obigem Schema ausdrücken. Rundung ist also bei der Zahldarstellung und ebenso bei Rechenoperationen nicht vermeidbar.

2.2.3 Rundung

Wie wir bereits gesehen haben, sind Rundungen bei der Verwendung von Maschinenzahlen unvermeidbar. Formal definieren wir eine Rundung wie folgt:

Definition 2.6 (Rundung)
Eine korrekte Rundung ist eine Operation $\mathrm{rd} : \mathbb{R} \to \mathbb{M}$, die jeder Zahl $r \in \mathbb{R}$ die nächstliegende Maschinenzahl $x \in \mathbb{M}$ zuordnet. Es gilt also

$$|\mathrm{rd}(r) - r| \leq |x - r| \qquad \forall x \in \mathbb{M}.$$

Legt man $m_u := \min\{x \in \mathbb{M} \mid x \geq r\}$ und $m_l := \max\{x \in \mathbb{M} \mid x \leq r\}$ fest, so ist die korrekte Rundung durch die Funktion

$$\mathrm{rd}(r) : \mathbb{R} \to \mathbb{M}, r \mapsto \begin{cases} m_u & \text{falls } r \geq (g_u + g_l)/2 \\ m_l & \text{falls } r \leq (g_u + g_l)/2 \end{cases}$$

gegeben. Bei der technischen Ausführung einer Rundung muss r dabei nicht exakt bekannt sein, was z.B. bei irrationalen Zahlen von Vorteil ist. Stattdessen ist es ausreichend, die $(t+1)$te Mantisse zu kennen:

$$r = \pm.m_1 m_2 \dots m_t m_{t+1} \dots 2^e$$

$$r' = \begin{cases} .m_1 m_2 \dots m_t & \text{falls } 0 \leq m_{t+1} < 1 \\ .m_1 m_2 \dots m_t \underbrace{(m_{t+1})}_{\text{Übertrag}} & \text{falls } 1 \leq m_{t+1} \end{cases}$$

Somit ergibt sich

$$\mathrm{rd}(r) := \mathrm{sign})(r) \cdot r' \cdot B^e.$$

Bemerkung 2.7
Alternativ zur Rundung, die in Definition 2.6 angegeben wurde, lassen sich weitere Rundungen definieren, etwa

$$rd_+ : \mathbb{R} \to \mathbb{M}, r \mapsto m_u$$
$$rd_- : \mathbb{R} \to \mathbb{M}, r \mapsto m_l$$
$$rd_+ : \mathbb{R} \to \mathbb{M}, r \mapsto \begin{cases} m_u & \textit{falls } r \geq 0 \\ m_l & \textit{falls } r < 0 \end{cases}.$$

Bei diesen Rundungen spricht man von gerichteten Rundungen. Beachte, dass diese nicht korrekt sind.

Bemerkung 2.8
Eine Rundung lässt sich äquivalent zu Definition 2.6 und Bemerkung 2.7 für alle Gleitkommazahlen definieren. Der Einfachheit halber haben wir hier darauf verzichtet. Umgekehrt gilt dies nur, wenn $\pm\infty$ in $\mathbb{M}$ enthalten sind, was in ANSI/IEEE Std 754–1983 der Fall ist. Bei letzterer Norm wird zudem abwechselnd auf– und abgerundet. Um dies zu erreichen wurde festgelegt, dass $rd((g_u + g_l)/2)$ immer einen geradzahligen Signifikanden haben muss.

Für die so definierten Rundungen ergeben sich folgende Schranken:

Theorem 2.9 (Schranke für Rundungsfehler)
*Für alle $r \in \mathbb{R}$ führt die Rundung zu einem **absoluten Fehler***

$$\varepsilon_{abs} := rd(r) - r$$

*und zu einem **relativen Fehler***

$$\varepsilon_{rel} := \begin{cases} (rd(r) - r)/r & \textit{falls } r \neq 0 \\ 0 & \textit{falls } r = 0 \end{cases}.$$

Für den absoluten und relativen Fehler gilt zudem

$$rd(r) = r(1 + \varepsilon_{rel}) \qquad \textit{mit} \quad |\varepsilon_{rel}| \leq \begin{cases} \frac{B}{2} B^{-t} & \textit{für korrektes Runden} \\ B^{1-t} & \textit{für gerichtetes Runden} \end{cases}$$

$$\textit{und} \quad |\varepsilon_{abs}| \leq \frac{1}{2} B^{E-t}.$$

ε_{rel} wird auch als Maschinengenauigkeit bezeichnet.

Beweis. Klar nach Definition 2.6. □

Im Laufe einer Rechnung kann es nun passieren, dass Werte autreten, die außerhalb des darstellbaren Bereichs

$$[x_{\max}, -x_{\min}] \cup \{0\} \cup [x_{\min}, x_{\max}]$$

liegen. Hierbei unterscheidet man den sogenannten **Underflow** und **Overflow**.

- Für $r \approx 0$, $r \in]-x_{\min}, x_{\min}[$ gilt dabei stets $\mathrm{rd}(r) = 0$. Somit ist für alle $r \in]-x_{\min}, x_{\min}[\backslash\{0\}$ der relative Fehler 1, wohingegen der absolute Fehler $< x_{\min}$ ist. In diesem Fall sollte eine Warnung „underflow“ gesetzt werden. Beachte, dass der absolute Fehler zwar klein ist, dies jedoch zu massiven Problemen z.B. bei der Auswertung von $1/r$ führt.

- Ist dagegen $|r| \geq x_{\max}$, so gilt $\mathrm{rd}(r) = \mathrm{sign}(r) \cdot x_{\max}$. Hier sollte die Warnung „overflow“ gesetzt werden. Beachte, dass der relative Fehler durch 1 beschränkt ist, der absolute Fehler kann jedoch beliebig groß werden, etwa bei der Berechnung von r^2 für $r \to \infty$.

Bemerkung 2.10
Beachte, dass die Fehlerschranken aus Theorem 2.9 nur gelten, solange kein Over– oder Underflow auftreten.

2.2.4 Gleitpunkt–Arithmetik

Aus der Existenz von Over– und Underflows lässt sich bereits ableiten, dass wir für alle $x, y \in \mathbb{M}$ nicht $x \cdot y \in \mathbb{M}$ für $\cdot \in \{+, -, \times, /\}$ folgern können. Nutzt man die bereits definierte Rundung $\mathrm{rd} : \mathbb{R} \to \mathbb{M}$, so erhält man folgendes Resultat:

Theorem 2.11 (Rundungsfehlerschranken)
Für die Operationen $\odot \in \{\oplus, \ominus, \otimes, \oslash\}$ mit $a \odot b := rd(a \cdot b)$ gilt für alle $a, b \in \mathbb{G}_{B,t}$

$$\begin{aligned} a \oplus b &:= rd(a + b) = (a + b) \cdot (1 + \alpha) \\ a \ominus b &:= rd(a - b) = (a - b) \cdot (1 + \beta) \\ a \otimes b &:= rd(a \cdot b) = (a \times b) \cdot (1 + \gamma) \\ a \oslash b &:= rd(a/b) = (a/b) \cdot (1 + \delta) \end{aligned}$$

mit α, β, γ und $\delta < \varepsilon_{rel}$.

Beweis. Klar nach Definition 2.6. □

Hierbei ist zu beachten, dass α, β, γ und δ zwar von $x, y \in \mathbb{R}$ abhängen, die Schranke $\varepsilon_{\mathrm{rel}}$ jedoch a priori bekannt ist, vergleiche Theorem 2.9. Analog zu Theorem 2.9 gelten auch hier die Schranken nur, falls kein Over– oder Underflow auftritt. Auch andere Rechengesetze wie das Assoziativgesetz und das Distributivgesetz gelten im Fall der Gleitpunkt–Arithmetik nicht mehr. Kommutativität der Addition und Multiplikation hingegen kann auch hier gezeigt werden. Für weitere Details vergleiche [3].

2.2.5 Beispiele von Programm– und Rundungsfehlern

- Zu Beginn des Jahres 2010 konnten wegen eines Softwarefehlers im EMV-Sicherheitschip bei der Verarbeitung der Jahreszahl 2010 viele Bankkunden tagelang an Automaten kein Geld abheben (http://de.wikipedia.org/wiki/Programmfehler).

- 1982 stürzte ein Prototyp des F117 Kampfjets ab, da bei der Programmierung die Steuerung des Höhenruders mit der des Seitenruders vertauscht worden war (http://de.wikipedia.org/wiki/Programmfehler).

- 2002 platzte ein Parteitag der Grünen. Sie hatten 200 Delegiertenplätze zu vergeben und verteilten diese mit Hilfe von Excel auf 47 Wahlkreise. Excel zeigte gerundete Werte an; rechnete intern aber mit Kommazahlen. So wurden 202 Delegierte zur Wahlversammlung eingeladen (http://www5.in.tum.de/~huckle/bugse.html).

- 1991 verfehlte eine Patriot-Rakete ihr Ziel um $0,34$ Sekunden; dies entspricht 1676 Metern. Die Speicherung von

$$\frac{1}{10} = (0.0001100110011001100110011001100....)_{(2)}$$

  Sekunden in einem 24 Bit Register führte jede Sekunde zu einem Fehler der internen Uhr von $0,000000095$ Sekunden. 100 Stunden nach dem Boot-Prozess (einschalten) kummulieren sich die Rundungsfehler zu:

$$0,000000095 \cdot 100 \cdot 60 \cdot 60 \cdot 10 = 0,34$$

  (http://ta.twi.tudelft.nl/nw/users/vuik/wi211/disasters.html#patriot)

- 1996 explodierte eine Ariane-5-Rakete 36,7 Sekunden nach dem Start. Ein nicht abgefangener Fehler bei der Umwandlung einer 64-Bit-Gleitkommazahl in eine 16-Bit-Ganzzahl führte zu extremer Fluglage und das Neutralisationssystem sprengte die Rakete (http://www.ima.umn.edu/~arnold/disasters/ariane.html).

2.3 Kondition und Stabilitätsanalyse

Wie wir bereits festgestellt haben, können bei der Berechnung mit Computern verschiedene Fehler auftreten. Innerhalb einer Verfahrensvorschrift wie etwa der Matrix–Vektor–Multiplikation oder innerhalb eines Algorithmus können sich derartige Fehler fortpflanzen und gegenseitig verstärken. In diesem Abschnitt beleuchten wir diese Problem genauer. Dazu betrachten wir für eine Menge $D \in \mathbb{R}^n$ und eine Abbildung $f : D \to \mathbb{R}^m$ das Problem der Berechnung

$$y = f(x), \qquad x \in D.$$

Hierbei bezeichnen wir x als Eingabedaten, y als Ausgabe– oder Resultatdaten. Anstelle einer exakten Rechnung $y = f(x)$ wird man bei der Umsetzung im Rechner eine Approximation $\tilde{f}(\tilde{x})$ erhalten, wobei

$\tilde{x}$ eine Näherung von x darstellt, die z.B. durch Rundung $\tilde{x} = \mathrm{rd}(x)$ entsteht, und

$\tilde{f}$ eine Approximation von f ist.

Die Genauigkeit der Berechnung von $y = f(x)$ wird durch die *Fehlertypen*

- Fehler in den Eingabedaten,
- Abbruchfehler und
- Rundungsfehler während der Rechnung

begrenzt, vergleiche Abbildung 2.5.

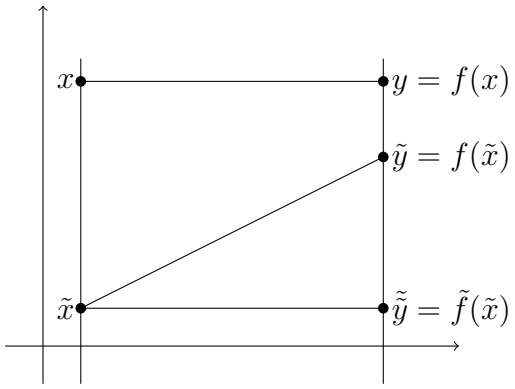


Abbildung 2.5: Fehlereinflüsse

2.3.1 Kondition

Da sich Rundungsfehler bereits bei den Eingabedaten auswirken können, können wir nicht davon ausgehen, dass wir die Funktion f an der Stelle x auswerten, sondern an einer Stelle $\tilde{x}$. Daher befassen wir uns nun damit, wie sich Fehler in x auf das Ergebnis $y = f(x)$ auswirken. Sei $\tilde{x}$ eine Näherung von x und sei

$$\begin{array}{rcl} \Delta x = \tilde{x} - x & : & \text{der } \textit{absolute} \text{ Eingabedatenfehler} \\ \frac{\tilde{x}_i - x_i}{x_i} & : & \text{der } \textit{relative} \text{ Eingabedatenfehler für } i = 1, \ldots, n \text{ und} \\ \tilde{y} = f(\tilde{x}) & : & \text{der Näherungswert für } y = f(x). \end{array}$$

Falls f eine $\mathcal{C}^1$ Funktion ist, so gilt nach der Taylor–Entwicklung erster Ordnung (oder analog nach dem Mittelwertsatz)

$$\Delta y_i = f_i(x + \Delta x) - f_i(x) \approx \frac{\partial f_i(x)}{\partial x} \Delta x = \sum_{j=1}^{n} \frac{\partial f_i(x)}{\partial x_j} \Delta x_j, \qquad i = 1, \ldots, m.$$

Der kombinierte Ausdruck $\Delta y = \tilde{y} - y$ wird auch **absoluter Ausgabedatenfehler** genannt. Für den **relative Ausgabedatenfehler** erhält man entsprechend

$$\frac{\Delta y_i}{y_i} \approx \sum_{j=1}^{n} \left(\frac{\partial f_i(x)}{\partial x_j} \frac{x_j}{f_i(x)} \right) \left(\frac{\Delta x_j}{x_j} \right), \qquad i = 1, \ldots, m.$$

Üblicherweise ist die relative Fehlerverstärkung $\left| \frac{\partial f_i(x)}{\partial x_j} \frac{x_j}{f_i(x)} \right|$ von größerer Bedeutung als die absolute Fehlerverstärkung $\left| \frac{\partial f_i(x)}{\partial x} \right|$ und wird auch als Kondition bezeichnet:

Definition 2.12 (Kondition)
Die Zahlen

$$k_{ij}(x) = \left| \frac{\partial f_i(x)}{\partial x_j} \frac{x_j}{f_i(x)} \right|$$

heißen *Verstärkungsfaktoren* bzw. (relative) *Konditionszahlen.*
Das Problem „Berechne $y = f(x)$“ heißt *gut konditioniert*, falls alle $k_{ij}(x)$ die Größenordnung 1. Andernfalls heißt das Problem *schlecht konditioniert.*

Beispiel 2.13
Betrachtet man die Multiplikation $y = f(x_1, x_2) = x_1 \cdot x_2$, so gilt $k_{11}(x) = k_{12}(x) = 1$. Die Multiplikation ist also gut konditioniert.
Für die Division $y = f(x_1, x_2) = x_1/x_2$ gilt ebens $k_{11}(x) = k_{12}(x) = 1$. Die Division ist also ebenfalls gut konditioniert.
*Im Gegensatz dazu erhält man für die Addition und Subtraktion $y = f(x_1, x_2) = x_1 \pm x_2$ die Konditionszahlen $k_{11}(x) = \left|\frac{x_1}{x_1 \pm x_2}\right|$ und $k_{12}(x) = \left|\frac{x_2}{x_1 \pm x_2}\right|$. Falls nun $x_1 \approx \mp x_2$ ist, dann ist das Problem schlecht konditioniert. Dieses Phänomen wird auch als **Auslöschung** bezeichnet.*

Die Kondition einer Abbildung kann auch mit Hilfe einer entsprechenden Norm $\|\cdot\|_p$ zusammengefasst werden. So wird beispielsweise für eine gegebene Matrixnorm $\|\cdot\|_p$ und eine invertierbare Matrix A der Wert

$$\mathrm{cond}_p(A) := \|A\|_p \|A^{-1}\|_p$$

als Kondition von A bezeichnet.

Beachte, dass bei einigen Problemen die Auslöschung vermieden werden kann, wenn das Problem geeignet umformuliert wird. Wir werden dies im Speziellen bei der LR–Zerlegung in Kapitel 3 noch einmal aufgreifen.

2.3.2 Stabilitätsanalyse

Bei der Analyse von Funktionen oder Algorithmen unterscheidet man zwei verschiedene Analysearten, die sogenannte **Vorwärts**– und **Rückwärts–Analyse**.

In der Vorwärts–Analyse versucht man, einen expliziten Ausdruck für die Ausgabedaten zu bestimmen um diesen mit dem exakten Ergebnis y zu vergleichen. Ziel ist dabei, eine Abschätzung für den relativen Ausgabedatenfehler zu bestimmen. Betrachten wir wieder das Problem $y = f(x)$, wobei f eine reellwertige Funktion ist. Im Verlauf der Funktionsauswertung bzw. des Algorithmus müssen Rundungsfehler mit einer relativen Genauigkeit $\varepsilon_{\mathrm{rel}}$ in Kauf genommen werden, d.h.

$$\sqrt{\sum_{j=1}^{n} \left| \frac{\Delta x_j}{x_j} \right|^2} \leq \varepsilon_{\mathrm{rel}}.$$

Man kann daher nicht erwarten, dass die Genauigkeit des Ergebnisses besser ist als in der folgenden Ungleichung

$$\left| \frac{\tilde{f}_i(x) - f_i(x)}{f_i(x)} \right| \approx \sqrt{\left| \sum_{j=1}^{n} \frac{\partial f_i(x)}{\partial x_j} \frac{x_j}{f_i(x)} \right|^2} \sqrt{\sum_{j=1}^{n} \left| \frac{\Delta x_j}{x_j} \right|^2} \leq C_V \max_{j \in \{1,\ldots,n\}} \|k_{ij}\|_2 \varepsilon_{\mathrm{rel}}. \tag{2.1}$$

wobei $\|\cdot\|_2$ die Euklidische Norm bezeichnet und $C_V > 0$ eine nicht zu große Konstante ist, die von Δx und x unabhängig ist. Wenn eine solche Ungleichung (2.1) gilt, so nennt man den Algorithmus vorwärts stabil.

Bei der Rückwärts–Analyse interpretiert man die berechnete Näherung als exakte Lösung eines Problems mit gestörten Eingangsdaten, als $\tilde{f}(x) = f(\tilde{x}) = f(x + \Delta x)$ und untersucht die Größe $|\Delta x|$. Gibt es mehrere Urbilder $x + \Delta x$, so nimmt man traditionell eines mit betragskleinster Störung Δx. Gilt

$$\left\| \frac{\Delta x}{x} \right\|_2 \leq C_R \varepsilon_{\text{rel}}$$

wobei wiederum $\|\cdot\|_2$ die Euklidische Norm bezeichnet und $C_R > 0$ eine nicht zu große Konstante ist, so nennt man die Funktion bzw. den Algorithmus rückwärts stabil. Für einen rückwärts stabilen Algorithmus ergibt sich

$$\begin{aligned} \left| \frac{|\tilde{f}(x) - f(x)|}{|f(x)|} \right| &= \left| \frac{|f(\tilde{x}) - f(x)|}{|f(x)|} \right| \\ &\leq \| \max_{i,j} k_{ij} \| \left\| \frac{\tilde{x} - x}{x} \right\| \leq \| \max_{i,j} k_{ij} \| C_R \varepsilon_{\text{rel}} \end{aligned}$$

Folglich ist jeder rückwärts stabile Algorithmus auch vorwärts stabil, wenn man $C_R = C_V$ setzt. Die Umkehrung gilt jedoch nicht. Weitere Details zu Kondition und Stabilitätsanalyse finden sich z.B. in [2].

Kapitel 3

Direkte Verfahren für lineare Gleichungssysteme

Der grundlegende Baustein (so gut wie) aller numerischen Algorithmen ist die Lösung linearer Gleichungssysteme. Hierfür stehen eine Reihe verschiedener Methoden zur Verfügung, von denen wir hier eine Auswahl sogenannter *direkter* Verfahren behandeln. Direkte Verfahren zeichnen sich dadurch aus, dass die Lösung „direkt “ berechnet wird, also durch elementare Zeilen– und Spaltenumformungen. Im Gegensatz dazu stehen die *iterativen* Lösern, die wir in Kapitel **??** analysieren werden. Bei diesen wird eine Folge von Teilproblemen zur Approximation der Lösung genutzt.

Ausführlich aufgeschrieben besteht das Problem, mit dem wir uns hier beschäftigen wollen, darin, Zahlen $x_1, \ldots, x_n \in \mathbb{R}$ zu bestimmen, für die das Gleichungssystem

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &= b_m \end{aligned} \tag{3.1}$$

erfüllt ist. Die Darstellung (3.1) ist aber unhandlich und deshalb werden wir das Problem eines lineare Gleichungssystems in der Matrix–Form

$$Ax = b \tag{3.2}$$

mit

$$A := \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad x := \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad \text{und} \quad b := \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \tag{3.3}$$

schreiben. Diese Schreibweise hat nicht nur den Vorteil, dass man ein Gleichungssystem viel kürzer aufschreiben kann. Hinzu kommt, dass gewisse Eigenschaften der Matrix A entscheiden, welches Verfahren zur Lösung von (3.2) überhaupt und zudem welches sinnvollerweise eingesetzt werden kann.

Notationstechnisch werden wir uns an den mathematischen Standard halten. Hierbei werden Matrizen typischerweise mit großen Buchstaben (z.B. A) und Vektoren mit kleinen Buchstaben (z.B. x oder b) bezeichnet. Die Einträge von Matrizen und Vektoren werden mit indizierten Kleinbuchstaben gekennzeichnet, vergleiche (3.3). Mit dem hochgestellten $\top$ bezeichnen wir

transponierte Matrizen und Vektoren, also z.B.

$$x^\top = \begin{pmatrix} x_1 & \cdots & x_n \end{pmatrix} \quad \text{oder} \quad A^\top = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix}$$

Ist die Anzahl m der Gleichungen in (3.3) gleich der Anzahl der Unbekannten $x_1, \ldots, x_n$, so stellt A eine quadratische Matrix der Dimension $n \times n$ dar. Für derartige Matrizen ist aus der linearen Algebra bekannt, dass genau dann eine eindeutige Lösung für das Problem (3.2) existiert, wenn die Matrix A invertierbar ist. Aus Vereinfachungsgründen werden wir in diesem Kapitel immer annehmen, dass A quadratisch und invertierbar ist.

Im Allgemeinen existieren aber folgende Fälle:

Fall	Anzahl der Lösungen
$m = n$, $\operatorname{rang}(A) = n$	Eindeutig lösbar. Da A invertierbar ist folgt $x = A^{-1}b$
$m = n$, $\operatorname{rang}(A) \neq \operatorname{rang}(A\|b)$	Keine Lösung
$m = n$, $\operatorname{rang}(A) < n$, $\operatorname{rang}(A) = \operatorname{rang}(A\|b)$	Unendlich viele Lösungen. Das Problem heißt *unterbestimmt*. Dimension des Lösungsraums ist $n - \operatorname{rang}(A)$.
$m < n$	Unendlich viele Lösungen. Das Problem heißt *unterbestimmt*. Dimension des Lösungsraums ist $n - \operatorname{rang}(A)$. Anwendung etwa in der Linearen Optimierung.
$m > n$	Im Allgemeinen keine Lösung. Das Problem heißt *überbestimmt*. Anwendung etwa in der Linearen Ausgleichsrechnung.

Tabelle 3.1: Auftretende Fälle bei Linearen Gleichungssystemen

Bevor wir einzelne Verfahren vorstellen, zeigen wir hier noch ein paar weitere Grundbegriffe der linearen Algebra in MATLAB:

```
% Script zur Demonstration von Grundlagen der linearen Algebra in Matlab

echo on;
clear all;
close all;


% Nuetzliche Befehle fuer Vektoren und Matrizen
% ---------------------------------------------

% Vektoren:
% ---------
% Definiere einen Zeilenvektor
v1 = [1 2 -3]

% Einen Spaltenvektor
v2 = [4; 5; 6]

% und noch einen Spaltenvektor
v3 = [7 8 9]'

% Das Skalarprodukt:
```

```
% Da Vektoren in MATLAB nichts anderes als einzeilige bzw. einspaltige
% Matrizen sind, wird das Skalarprodukt mit der Matrizenmultiplikation
% gebildet. Aber Achtung: die Dimension muss stimmen:
v1*v2

%v2*v2     % <- dies gibt eine Fehlermeldung

v2'*v2

% Anders herum multipliziert erhaelt man das dyadisches Produkt
v2*v1

% und die komponentenweise Multiplikation hatten wir bereits gesehen:
v2.*v3

% Normen:
% -------
% Matlab hat die Berechnung von verschiedenen Normen bereits
% vorimplementiert, so etwa die 1-, 2- und die Unendlich-Norm
norm(v1,1)
norm(v1,2)
norm(v1,inf)


% Matrizen:
% ---------
% Wie bereits gesehen kann man in Matlab eine Matrix direkt eingeben:
A = [1 2 3; 4 5 6; 7 8 9]

% Es gibt jedoch auch vordefinierte Funktionen fuer Null-, Eins- und
% Einheitsmatrizen:
A0 = zeros(3,4)
A1 = ones(5,4)
Id = eye(4)

% Die Transponierte einer Matrix erhaelt man durch den Befehl
At = A'

% und Matrizen und Vektoren lassen sich wie folgt zu groesseren Einheiten
% kombinieren:
A3 = [A1; Id]
A4 = [v2, A0]
A5 = [A1, eye(5)]

% Ebenso ist es moeglich, eine Folge von Matrizen zu erzeugen
for i=1:5
  AA(:,:,i) = i.*ones(3,3)
end

% Oftmals nuetzlich bei der Generierung von Schleifen ist die Abfrage der
% Dimension einer Matrix oder eines Vektors
size(A1)
size(A1,1)
size(A1,2)
size(v1)
size(v2)
length(v1)
length(v2)

```

```
% Ab und zu wird auch der Kopierbefehl benutzt werden muessen. Dies sollte
% aus speichertechnischen und Laufzeitgruenden aber so selten wie moeglich
% genutzt werden
Aneu = A1

% Die kopierte Matrix kann nun unabhaengig modifiziert werden
Aneu(1,1)=10
Aneu
A1

% Rang, Kern und Bild:
% --------------------
% Mathematische Begriffe wie Rang, Kern und Bild lassen sich in Matlab
% direkt eingeben und auswerten
rank(A)
null(A)
orth(A)

% und auch visualisieren, hier die Visualisierung des Kerns (1d in 3d):
kern = null(A)

% Hier ist der Kern eindimensional, daher definieren wir eine
% eindimensionale Veraenderliche xx
xx = -1:0.1:1;
% die wir nun zur Generierung eines Plots nutzen
plot3(xx.*kern(1),xx.*kern(2),xx.*kern(3));
hold on
% Nun setzen wir noch die Achsen, damit der Kern auch optisch senkrecht auf
% dem Bild steh
axis([-3 3 -3 3 -3 3]);
axis square;

% Ebenso laesst sich das Bilds visualisieren (2d in 3d):
bild = orth(A)

% Hierzu erzeugen wir zunaechst Vektoren, welche die Flaeche definieren. Da
% das Bild zweidimensional ist, benoetigen wir diesmal zwei Veraenderliche:
echo off
flaeche = [];
for xi = -5:0.5:5
  for yi = -5:0.5:5
  flaeche = [flaeche, xi.*bild(:,1) + yi.*bild(:,2)];
  end
end
echo on

% Nun erzeugen wir das Zeichengitter in der (x,y)-Ebene
[xx,yy] = meshgrid(-1:0.1:1);

% und berechnen die z-Werte der Flaeche aus den definierenden Vektoren
zz = griddata(flaeche(1,:),flaeche(2,:),flaeche(3,:),xx,yy);

% die sich abschliessend gegen die Veraenderlichen plotten lassen
mesh(xx,yy,zz);


% Determinante und Inverse
% ------------------------
% Determinante und Inverse eine Matrix lassen sich mit den folgenden
```

```
% Befehlen berechnen
det(A)
inv(A)   % <- Beispiel einer nicht invertierbaren Matrix

% Da die Matrix A nicht invertierbar ist, hier noch ein paar Beispiele von
% invertierbare Matrizen:
% Die erste Matrix ist eine 2x2-Block Drehung, die zusaetzlich in der 3.
% Koordinate eine Streckung um 2 beinhaltet
A2 = [0.5*sqrt(2), -0.5*sqrt(2), 0; 0.5*sqrt(2),0.5*sqrt(2),0; 0,0,2]
inv(A2)

% Ebenso kann man eine Matrix auch direkt eingeben und invertieren
A3 = [1 3 4; 6 5 4; 9,7,2]
inv(A3)



% Loesung linearer Gleichungssysteme
% ----------------------------------
% Zur Loesung lineare Gleichungssysteme benoetigen wir zusaetzlich zur
% Matrix, die den funktionalen Zusammenhang der gesuchten Parameter
% darstellt, eine rechte Seite von Ergebniswerten
b = [1; 2; 3]

% Nun gibt es in Matlab verschiedene Varianten, ein lineares
% Gleichungssystem zu losesen, etwa ueber die Inverse
x = inv(A3)*b

% oder die (viel effizientere) direkte Variante
x = A3\b

% Das Ergebnis laesst sich einfach gegenrechnen
A3*x - b

% Matlab kann aber auch mit nicht invertierbaren Matrizen umgehen und eine
% existierende Loesung berechnen
x = A\b

% Ein Check ergibt hier
A*x - b


% Existiert keine Loesung, so erhaelt man folgendes
b1 = [1; 2; 4]
x = A\b1

% Das Ergebnis laesst sich wieder gegenrechnen
A*x - b1
% und man erkennt, dass es keine Loesung ist.
```

Programm 3.1: Script zur Demonstration von Grundlagen der linearen Algebra in MATLAB

In den Übungen werden die hier vorgestellten Algorithmen implementiert und Stärken und Schwächen einzelner Algorithmen untersucht. Das Hauptaugenmerk liegt aber auf der Umsetzung der Methoden, die hier in sogenannten *Pseudocode* beschrieben werden. Dieser muss in Programmcode umgewandelt und anschließend gegen fehlerhafte Eingaben gesichert werden.

3.1 Gauss Elimination

Das erste direkte Verfahren, das wir hier behandeln, ist das sogenannten *Gauss'sche Eliminationsverfahren* oder auch einfach *Gauss Elimination*. Dieses Verfahren ist dadurch motiviert, dass man ein lineares Gleichungssystem der Form $Ax = b$ leicht lösen kann, wenn die Matrix A in *oberer Dreiecksform* vorliegt, d.h. wenn A die Gestalt

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix}$$

hat. In diesem Fall lässt sich die Lösung von (3.2) mittels der rekursiven Vorschrift

$$x_n = \frac{b_n}{a_{nn}}, \quad x_{n-1} = \frac{b_{n-1} - a_{n-1}x_n}{a_{n-1\,n-1}}, \ldots, \quad x_1 = \frac{b_1 - a_{12}x_2 - \ldots - a_{1n}x_n}{a_{11}}$$

oder kompakt geschrieben

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{ij}x_j}{a_{ii}} \tag{3.4}$$

(mit der Konvention $\sum_{j=n+1}^{n} a_{ij}x_j = 0$) lösen. Dieses Verfahren wird als *Rückwärtssubstitution* bezeichnet und ist algorithmisch wie folgt umsetzbar:

Algorithmus 3.1 (Rückwärtseinsetzen)
Gegeben sei eine rechte obere Dreiecks–Matrix $R \in \mathbb{R}^{n\times n}$ und ein Vektor $b \in \mathbb{R}^n$.

(0) (Durchlaufe Matrix Rückwärts)
Setze $i := n$

(1) (Löse Zeile)
Setze $x_i := (b_i - \sum_{j=i+1}^{n} R_{ij} \cdot x_j)/R_{ii}$

(2) Setze $i := i - 1$ und gehe zu (1)

Die Idee der Gauss Elimination ist nun, das Gleichungssystem $Ax = b$ in ein Gleichungssystem $\widetilde{A}x = \tilde{b}$ umzuformen, so dass $\widetilde{A}$ in obere Dreiecksform vorliegt. Um dies zu realisieren werden elementare Zeilenumformung genutzt um die Einträge unterhalb der Diagonalen zu Null zu machen. Ein Algorithmus sähe wie folgt aus:

Algorithmus 3.2 (Gauss Elimination)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n\times n}$ und ein Vektor $b \in \mathbb{R}^n$.

(0) (Zeilen– und Spaltenindex des zu eliminierenden Eintrags setzen)
Setze $i := n$ und $j := 1$

(1) (Subtrahiere a_{ij}/a_{jj}–mal die j–te Zeile von der i–ten Zeile)
Setze $\alpha := a_{ij}/a_{jj}$ und berechne

$$a_{ik} := a_{ik} - \alpha a_{jk} \qquad \forall k = j, \ldots, n$$
$$b_i := b_i - \alpha b_j$$

(2) (Abarbeitung von Zeilen und Spalten)
Falls $i \geq j + 2$:

Setze $i := i - 1$ und gehe zu (1)

Sonst:

Falls $j \leq n - 2$:

Setze $j := j + 1$ und $i := n$ und gehe zu (1)

Sonst:

Ende des Algorithmus

Die resultierende Matrix $\widetilde{A}$ ist nun in rechter oberer Dreiecksform und die Rückwärtssubstitution aus Algorithmus 3.1 kann auf die Matrix $\widetilde{A}$ zusammen mit dem Vektor $\tilde{b}$ angewandt werden.

3.1.1 Pivotierung

Wie sich in Schritt (1) von Algorithmus 3.2 erkennen lässt, kann es passieren, dass dieser Algorithmus zu keinem Ergebnis führt, da das Diagonalelement a_{jj} Null sein könnte. Beachte, dass dies selbst dann auftreten kann wenn das Gleichungssystem $Ax = b$ selbst lösbar ist. In Algorithmus 3.2 wurde stillschweigend angenommen, dass $a_{jj} \neq 0$ für alle $j = 1, \ldots, n$ gilt. Um diesen Fall abzufangen könnte man einen Test der Eingabedaten durchführen und eine entsprechende Fehlermeldung ausgeben. Algorithmisch bietet sich alternativ an, eine Zeile mit $a_{j\cdot} = 0$ mit einer Zeile $a_{i\cdot} > 0$ mit $i > j$ zu vertauschen, da diese Einträge während der Gauss Elimination ohnehin zu Null gerechnet werden. Hierbei muss man allerdings beachten, dass auch die Zeilen der rechte Seite b mit zu vertauschen sind. Dieses Verfahren wird *Pivotierung* genannt und lässt sich wie folgt in die Gauss Elimination aus Algorithmus 3.2 integrieren:

Algorithmus 3.3 (Gauss Elimination mit Pivotierung)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n \times n}$ und ein Vektor $b \in \mathbb{R}^n$.

(0) (Zeilen– und Spaltenindex des zu eliminierenden Eintrags setzen)
Setze $i := n$ und $j := 1$

(1a) (Pivotierung)
Falls $a_{ij} = 0$:

Gehe zu (2)

Sonst:

Falls $a_{jj} = 0$:

Vertausche a_{jk} und a_{ik} für alle $k = j, \ldots, n$
Vertausche b_j und b_i
Gehe zu (2)

(1b) (Subtrahiere a_{ij}/a_{jj}–mal die j–te Zeile von der i–ten Zeile)
Setze $\alpha := a_{ij}/a_{jj}$ und berechne

$$a_{ik} := a_{ik} - \alpha a_{jk} \qquad \forall k = j, \ldots, n$$
$$b_i := b_i - \alpha b_j$$

(2) (Abarbeitung von Zeilen und Spalten)
Falls $i \geq j + 2$:

Setze $i := i - 1$ und gehe zu (1)

Sonst:

Falls $j \leq n - 2$:

Setze $j := j + 1$ und $i := n$ und gehe zu (1)

Sonst:

Ende des Algorithmus

Hierbei wird das Element a_{ij}, das mit a_{jj} vertauscht wird, als *Pivotelement* bezeichnet. Beachte, dass wir in Schritt (1a) noch einmal unterscheiden, ob eine Vertauschung überhaupt notwendig ist. Falls der Eintrag $a_{ij} = 0$ ist, so muss dieser Eintrag nicht durch Zeilenumformungen zu Null gerechnet werden. Zudem würde es auch keinen Sinn machen, in einem solchen Fall die Zeilen $a_{i\cdot}$ und $a_{j\cdot}$ zu vertauschen, da hierdurch auf jeden Fall ein Nulleintrag in a_{jj} generiert wird.

3.1.2 Pivotsuche

Neben der Pivotierung von Algorithmus 3.3, bei der das Pivotelement sukzessive in der Spalten gewählt wird, kann man auch andere Pivotierungen nutzen, die geeignete Pivotelemente suchen. Das Kriterium zu Bestimmung eines Pivotelements wird dabei meist so gewählt, dass das Phänomen der Auslöschung vermieden wird (siehe Abschnitt 2.2). Auslöschung kann während der Gauss Elimination bei den Operationen

$$a_{ik} - \frac{a_{ij}}{a_{jj}} a_{jk} \qquad \text{sowie} \qquad b_i - \frac{a_{ij}}{a_{jj}} b_j$$

auftreten. Hier wollen wir uns auf folgende Heuristik beschränken: Man wählt das Pivotelement in der j–ten Spalte so, dass die zu subtrahierenden Ausdrücke betragsmäßig klein sind (und somit die zu berechnende Differenz betragsmäßig groß im Vergleich zu den Originalzahlen ist). Da nur die Brüche a_{jk}/a_{jj}, $k = j, \ldots, n$ und b_j/a_{jj} beeinflussbar sind, versucht man diese Brüche im Betrag durch Wahl eines Pivotelements möglichst klein zu machen. Hieraus resultiert der folgende Algorithmus:

Algorithmus 3.4 (Gauss Elimination mit Pivotsuche)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n\times n}$ und ein Vektor $b \in \mathbb{R}^n$.

(0) (Zeilen– und Spaltenindex des zu eliminierenden Eintrags setzen)
Setze $i := n$ und $j := 1$

(1a) (Pivotsuche)
Wähle den Zeilenindizes $p \in \{j, \dots, n \mid a_{pj} \neq 0\}$ aus, für den der Ausdruck

$$\max \left\{ \max_{k=j,\dots,n} \frac{|a_{pk}|}{|a_{pj}|}, \frac{|b_p|}{|a_{pj}|} \right\}$$

minimal wird.
Falls $p \neq j$:

Vertausche a_{jk} und a_{pk} für alle $k = j, \dots, n$

Vertausche b_j und b_p

Gehe zu (2)

(1b) (Subtrahiere a_{ij}/a_{jj}–mal die j–te Zeile von der i–ten Zeile)
Setze $\alpha := a_{ij}/a_{jj}$ und berechne

$$\begin{aligned} a_{ik} &:= a_{ik} - \alpha a_{jk} \qquad \forall k = j, \dots, n \\ b_i &:= b_i - \alpha b_j \end{aligned}$$

(2) (Abarbeitung von Zeilen und Spalten)
Falls $i \geq j + 2$:

Setze $i := i - 1$ und gehe zu (1)

Sonst:

Falls $j \leq n - 2$:

Setze $j := j + 1$ und $i := n$ und gehe zu (1)

Sonst:

Ende des Algorithmus

Dieser Algorithmus ließe sich noch erweitern, indem man etwa in der gesamten unteren rechten Submatrix nach geeigneten Pivotelementen sucht. Diese Art Suche bezeichnet man als *Totalpivotsuche*. Gute Implementierungen der Gauss Elimination werden immer solche Pivotsuchmethoden verwenden. Diese bieten eine Verbesserung der Robustheit aber keinen vollständigen Schutz gegen große Fehler bei schlechter Kondition, was aus prinzipiellen mathematischen Gründen nicht möglich ist.

Eine allgemeinere Strategie zur Vermeidung schlechter Kondition ist die sogenannte *Präkonditionierung*, bei der eine Matrix $P \in \mathbb{R}^{n\times n}$ gesucht wird, für die die Kondition von PA kleiner ist als die von A, so dass das dann besser konditionierte Problem $PAx = Pb$ gelöst werden kann.

In der Praxis ist die Gauss Elimination trotz ihrer Einfachheit in der Regel nicht das Mittel

der Wahl. Hintergrund hierfür ist, dass oftmals die gleiche Matrix A aber unterschiedliche rechte Seiten b — etwa bei verschiedenen Messreihen — zu betrachten sind. Da der Vektor b in Laufe des Algorithmus mittransformiert werden könnte, müsste in solch einem Fall die Gauss Elimination für jedes Paar (A, b) durchgeführt werden. Die nun folgende LR Zerlegung „sammelt" die elementaren Zeilenumformung auf, wodurch eine Transformation von b vermieden werden kann.

3.2 LR Zerlegung

Im Gegensatz zur Gauss Elimination werden bei der LR Zerlegung die elementaren Zeilenumformungen nicht auf das gesamte lineare Gleichungssystem $Ax = b$ angewandt und so sowohl die Matrix A als auch der Vektor b zu $\widetilde{A}$ und $\widetilde{b}$ verändert. Stattdessen werden die Operationen in einer Matrix L gesammelt, die nach Ende des Algorithmus in unterer Dreiecksform vorliegt.

Wenn eine derartige Zerlegung zur Verfügung steht, so erhält man das Problem

$$Ax = LRx = b,$$

das nun gelöst werden muss. Ähnlich wie bei der Gauss Elimination lässt sich auch hier die Dreiecksstruktur der Matrizen L und R ausnutzen. Hierfür betrachtet man zuerst das Teilproblem

$$Ly = b$$

zur Bestimmung der Hilfsgröße y und löst anschließend das Teilproblem

$$Rx = y$$

mittels Rückwärtssubstitution. Da L eine linke untere Dreiecksmatrizen ist, kann die Hilfsgröße y durch die sogenannte *Vorwärtssubstitution* berechnet werden, die sich analog zur Rückwärtssubstitution wie folgt gestaltet:

Algorithmus 3.5 (Vorwärtseinsetzen)
Gegeben sei eine linke untere Dreiecks–Matrix $L \in \mathbb{R}^{n\times n}$ und ein Vektor $b \in \mathbb{R}^n$.

(0) (Durchlaufe Matrix vorwärts)
Setze $i := 1$

(1) (Löse Zeile)
Setze $x_i := (b_i - \sum_{j=1}^{i-1} L_{ij} \cdot x_j)/L_{ii}$

(2) Setze $i := i + 1$ und gehe zu (1)

Da wir bei der LR Zerlegung nur die elementaren Zeilentransformationen der Gauss Elimination in einer Matrix L speichern wollen, erinnert der zugehörige Algorithmus sehr stark an die Gauss Elimination aus Algorithmus 3.2. Einzig der Schritt (1) ist zu modifizieren, um die angesprochene Speicherung durchzuführen.

Algorithmus 3.6 (LR Zerlegung)
Gegeben sei eine Matrix $A \in \mathbb{R}^{n\times n}$, ein Vektor $b \in \mathbb{R}^n$ und eine Matrix $L \in \mathbb{R}^{n\times n}$.

(0) (Zeilen– und Spaltenindex des zu eliminierenden Eintrags setzen)
Setze $i := n$, $j := 1$ und $L = \mathrm{Id}$.

(1) (Subtrahiere a_{ij}/a_{jj}–mal die j–te Zeile von der i–ten Zeile)
Setze $l_{ij} := a_{ij}/a_{jj}$ und berechne

$$a_{ik} := a_{ik} - l_{ij}a_{jk} \qquad \forall k = j, \ldots, n$$

(2) (Abarbeitung von Zeilen und Spalten)
Falls $i \geq j + 2$:

Setze $i := i - 1$ und gehe zu (1)

Sonst:

Falls $j \leq n - 2$:

Setze $j := j + 1$ und $i := n$ und gehe zu (1)

Sonst:

Ende des Algorithmus

Da wir die Matrix L zu Beginn von Algorithmus 3.6 als Einheitsmatrix definieren und im folgenden nur Einträge l_{ij} mit $i > j$ neu setzen, ist L in unterer Dreiecksform. Die Matrix R entspricht der modifizierten Matrix A am Ende von Algorithmus 3.6.

Die für die Gauss Elimination vorgestellte Pivotierung bzw. Pivotsuche zur Vermeidung von Fehler und verbessern der Kondition des Algorithmus kann auch im Fall der LR Zerlegung angewandt werden. Da hierzu aber Einträge oberhalb der Diagonalen von L ungleich Null gesetzt werden müssen um entsprechende Tauschungen vorzunehmen, die Matrix L also nicht mehr in unterer Dreiecksform wäre, käme die Lösung mittels Vorwärts– und Rückwärtssubstitution nicht weiter in Frage. Daher werden derartige Permutationen in einer Permutationsmatrix P gesammelt und das Problem (wie bei der Präkonditionierung) als $PAx = PLRx = b$ geschrieben.

Eine weitere Strategie zur Behandlung schlecht konditionierter Gleichungssysteme, die wir nun genauer untersuchen wollen, ist die QR Zerlegung einer Matrix.

3.3 QR Zerlegung

Die LR Zerlegung, die explizit oder implizit Grundlage der bisher betrachteten Lösungsverfahren war, hat unter Konditionsgesichtspunkten einen wesentlichen Nachteil: Es kann nämlich passieren, dass die einzelnen Matrizen L und R der Zerlegung deutlich größere Kondition haben als die zerlegte Matrix A, also

$$\mathrm{cond}_p(A) \ll \mathrm{cond}_p(L) \cdot \mathrm{cond}_p(R).$$

Beispiel 3.7
Betrachte die Matrix

$$A = \begin{pmatrix} 0.001 & 0.001 \\ 1 & 2 \end{pmatrix}$$

mit LR Zerlegung

$$L = \begin{pmatrix} 1 & 0 \\ 1000 & 1 \end{pmatrix}, \qquad R = \begin{pmatrix} 0.001 & 0.001 \\ 0 & 1 \end{pmatrix}.$$

Hier gilt $cond_2(A) \approx 5000$*, wohingegen* $cond_2(L) \approx 1000000$ *und* $cond_2(R) \approx 1000$ *ist. Die* 2*–Kondition von* L *ist also etwa* 200 *mal so groß wie die von* A*.*

Selbst wenn wir eventuelle Fehler in der Berechnung von R und L vernachlässigen oder z.B. durch geschickte Pivotsuche vermindern können, so kann die schlechte Konditionierung von R und L durch die Verstärkung der bei der Rückwärts– und Vorwärtssubstitution auftretenden Rundungsfehler zu großen Fehlern Δx in der Lösung führen, besonders wenn die Matrix A selbst bereits schlecht konditioniert ist. Bei der LR Zerlegung kann es also passieren, dass die Kondition der Teilprobleme, die im Algorithmus auftreten, deutlich schlechter ist als die des Ausgangsproblems.

Bemerkung 3.8
Die Kondition des Ausgangsproblems hängt nur von der Problemstellung ab, die der Teilprobleme aber von dem verwendeten Algorithmus, weswegen diese auch als numerische Kondition bezeichnet werden.

Um die numerische Kondition zu verringern, wollen wir einen Algorithmus so konstruieren, dass die Kondition der Teilprobleme nicht größer ist als die Kondition des Originalproblems. Hierzu verwenden wir sogenannte *orthogonale* Matrizen, d.h. Matrizen Q für die $QQ^\top = 1$ gilt. Der Vorteil dieser Matrizen ist, dass sie in der $\|\cdot\|_2$–Norm die Kondition 1 haben, also nicht fehlerverstärkend wirken. Anschaulich gesprochen sind orthogonale Transformationen nichts anderes als *Drehungen* oder *Spiegelungen*. Hierdurch lässt sich auch die Kondition 1 leicht erkennen, da Drehungen und Spiegelungen längen– und winkeltreu sind. Zudem lassen sich Probleme der Form $Qy = b$ sehr leicht lösen, da entsprechend der Eigenschaft $QQ^\top = 1$ folgt, dass $Q^{-1} = Q^\top$. Somit ergibt sich die Lösung durch Transponieren, also $y = Q^\top b$.

Unser Ziel ist es nun, einen Algorithmus zu konstruieren, der eine gegebene Matrix A in eine orthogonale Matrix Q und eine rechte obere Dreiecksmatrix R zerlegt. Falls uns dies gelingt, so kann die Lösung von $Ax = QRx = b$ durch die zwei Teilprobleme

$$\begin{aligned} y &= Q^\top b \\ Rx &= y \end{aligned}$$

berechnet werden, wobei für letzteres wieder die Rückwärtssubstitution aus Algorithmus 3.1 verwendet werden kann.

Die Idee der QR Zerlegung liegt darin, die Zeilen der Matrix A als Vektoren aufzufassen und durch orthogonale Transformationen, sprich lineare Transformationen, die sich durch orthogonale Matritzen darstellen lassen, auf die gewünschte Form zu bringen. Hier wird die gewünschte Form darin bestehen, dass für die j–te Spalte von A die Einträge a_{kj}, $k > j$ zu Nulleinträgen

gemacht werden.
Zur Realisierung eines solchen Algorithmus bieten sich zwei Varianten an, eine basierend auf Spiegelungen, die zur sogenannten QR Zerlegung mit *Householder Spiegelungen* führt, und eine Variante, die Drehung nutzt und QR mit *Givens–Rotationen* genannt wird. Hier beschränken wir uns auf den Fall der Householder Spiegelungen.
Geometrisch gesehen suchen wir nun also eine Folge von Spiegelungen, die die Spalten der Matrix A nacheinander in die gewünschte Form bringen. Betrachten wir die j–te Spalte $a_{\cdot j} \in \mathbb{R}^n$, so ist eine Matrix $H^{(j)}$ gesucht, die $a_{\cdot j}$ auf einen Vektor der Form $\tilde{a}_{\cdot j} = (\underbrace{*, *, \ldots, *}_{j \text{ Stellen}}, 0)^\top$ bringt.
Der Vektor soll also in die Ebene $E_j = \operatorname{span}(e_1, \ldots, e_j)$ gespiegelt werden. Um diese Spiegelung zu konstruieren betrachten wir allgemeine Spiegelmatrizen der Form

$$H = H(v) = \mathrm{Id} - \frac{2vv^\top}{v^\top v},$$

wobei $v \in \mathbb{R}^n$ ein beliebiger Vektor ist. Eine Multiplikation mit H entspricht dabei geometrisch eine Spiegelung an der Ebene mit Normalenvektor $v/\|v\|$. Nun liegt es daran, den Vektor v richtig zu wählen. Durch etwas Mathematik ergibt sich hier die Wahl $v = a_{\cdot j}$, $j = 1, \ldots, n-1$. Dies führt zum folgenden Algorithmus:

Algorithmus 3.9 (QR Zerlegung)
Gegeben sei eine Matrix $A \in \mathbb{R}^{m \times n}$.

(0) Setze $Q := \mathrm{Id} \in \mathbb{R}^{m \times m}$.

(1) Für j von 1 bis $n-1$

(a) Setze $c := \operatorname{sgn}(a_{jj}) \sqrt{\sum_{i=j}^n a_{ij}^2}$
Falls $c = 0$:
Setze $j := j + 1$.
Sonst:
(i) Setze $a_{jj} := c + a_{jj}$ (die Einträge von v_i, $i \geq j$ stehen jetzt in a_{ij}, $i \geq j$).
(ii) Setze $d := 1/(ca_{jj})$
(iii) (Berechnung der Matrix Q)
Für k von 1 bis m
Setze $e := d\left(\sum_{i=j}^m a_{ij} q_{ik}\right)$
Für i von j bis m: Setze $q_{ik} := q_{ik} - e a_{ij}$
(iv) (Berechnung von R)
Für k von $j+1$ bis n (Berechnung der Spalten $j+1$ bis n, die Spalte j wir noch zur Speicherung von v_j benötigt)
Setze $e := d\left(\sum_{i=j}^m a_{ij} a_{ik}\right)$
Für i von j bis m: Setze $a_{ik} := a_{ik} - e a_{ij}$
Setze $a_{jj} := -c$. (Berechnung der Spalte j, hier ändert sich nur das Diagonalelement)

Im Gegensatz zu der explizit berechneten Matrix Q kann R am Ende von Algorithmus 3.9 aus A abgelesen werden, indem nur die obere rechte Dreiecksmatrix verwendet wird. Bei der

Lösung des linearen Gleichungssystems sollte die Invertierbarkeit der Matrix R geprüft werden, bevor der Algorithmus zur Rückwärtssubstitution 3.1 aufgerufen wird. Die Invertierbarkeit kann dabei an der Diagonalen von R abgelesen werden: Ist eine Diagonalelement Null, so ist die Matrix nicht invertierbar. Alternativ kann ein solcher Test bereits im obigen Algorithmus 3.9 umgesetzt werden, indem geprüft wird, ob die Werte c ungleich Null sind. In diesem Algorithmus entspricht c der Spiegelrichtung, die hier der Robustheit so gewählt ist, dass in Richtung des größeren Teilwinkels gespiegelt wird. Die Spiegelungen selbst werden nicht als Matrix–Matrix Multiplikation ausgeführt, sondern durch die Hilfsvariablen

$$d = \frac{2}{v^\top v}, \quad e = dv^\top w \quad \text{und} \quad H^{(j)}w = w - ev.$$

Bemerkung 3.10
Die QR Zerlegung funktioniert immer, auch wenn die Matrix A nicht invertierbar ist. Die resultierende Matrix Q ist in jedem Fall invertierbar, die Matrix R ist genau dann invertierbar wenn A invertierbar ist. Zudem, falls $A \in \mathbb{R}^{n\times n}$ eine invertierbare Matrix ist, dann gilt für die Matrizen Q und R der QR Zerlegung

$$cond_2(Q) = 1 \quad und \quad cond_2(R) = cond_2(A).$$

Bemerkung 3.11
Beachte, dass $cond_p(Q) = 1$ nur für $p = 2$ gelten muss. Da für zwei Vektornormen aber die Abschätzung $\|x\|_p \leq C_{pq}\|x\|_q$ gilt, ist zumindest eine extreme Verschlechterung gegenüber anderer induzierter Matrixnormen ausgeschlossen.

Nachdem wir Löser für lineare Gleichungssysteme eingeführt haben, behandeln wir als Nächstes eines der Hauptanwendungsgebiete solcher Methoden, die sogenannte lineare Ausgleichsrechnung.

Kapitel 4

Lineare Ausgleichsrechnung

Dieses Anwendungsbeispiel ist für fast alle experimentellen Wissenschaften wichtig, seien es Naturwissenschaften wie Physik und Chemie oder Wirtschafts– und Sozialwissenschaften.

Das Grundproblem der linearen Ausgleichsrechnung rührt daher, dass ein Experiment mit verschiedenen Eingabewerten $t_1, t_2, \ldots, t_m$ durchgeführt wird und man entsprechende Messwerte $z_1, z_2, \ldots, z_m$ erhält. Auf Grund von theoretischen Überlegungen, z.B. auf Grund eines physikalischen Gesetzes oder einer selbst erdachten Theorie), kennt man eine Funktion $f(t)$, für die $f(t_i) = z_i$ gelten sollte. Diese Funktion wiederum hängt von weiteren unbekannten Parametern $x_1, x_2, \ldots, x_n$ ab, die es zu bestimmen gilt. Hier schreiben wir $f(t, x)$ für $x = (x_1, \ldots, x_n)^\top$ um die Abhängigkeit der zu untersuchenden Funktion von den zu bestimmenden Parametern zu betonen. Beispiele einer derartigen Funktion sind etwa

$$f(t, x) = x_1 + x_2 t \quad \text{oder} \quad f(t, x) = x_1 + x_2 t + x_3 t^2.$$

Wie man hieran bereits erkennen kann, muss die Funktion $f(t, x)$ keine lineare Funktion sein, sie muss aber linear von den Parametern x_i abhängen.

Wenn wir annehmen, dass die Funktion f das Experiment wirklich exakt beschreibt und keine Messfehler vorliegen, so könnten wir die Parameter x_i durch Lösen des (im Allgemeinen nichtlinearen) Gleichungssystems

$$\begin{aligned} f(t_1, x) &= z_1 \\ &\vdots \\ f(t_m, x) &= z_m \end{aligned} \tag{4.1}$$

bestimmen. In vielen praktischen Fällen ist dieses Gleichungssystem linear, so z.B. in den beiden obigen Beispielen, in denen sich (4.1) zu $\widetilde{A}x = z$ mit

$$\widetilde{A} = \begin{pmatrix} 1 & t_1 \\ \vdots & \vdots \\ 1 & t_m \end{pmatrix} \quad \text{bzw.} \quad \widetilde{A} = \begin{pmatrix} 1 & t_1 & t_1^2 \\ \vdots & \vdots & \vdots \\ 1 & t_m & t_m^2 \end{pmatrix} \quad \text{und} \quad z = \begin{pmatrix} z_1 \\ \vdots \\ z_m \end{pmatrix}$$

ergibt. Diese linearen Gleichungssysteme haben m Gleichungen, eine für jedes Wertepaar (t_i, z_i), und n Unbekannte, nämlich die unbekannten Parameter x_i. Bei diesen Probleme ist m üblicherweise sehr viel größer als n, das System ist also überbestimmt.

Da Messwerte eines Versuchs praktisch immer mit Fehlern behaftet sind, ist es sicherlich zu optimistisch anzunehmen, dass das Gleichungssystem $\widetilde{A}x = z$ lösbar ist. Wie wir bereits in Tabelle 3.1 festgestellt haben, besitzen überbestimmte Systeme nur in Ausnahmefällen eine

Lösung. Statt also den vermutlich vergeblichen Versuch zu machen, eine exakte Lösung x dieses Systems zu finden, wollen wir eine möglichst gute Näherungslösung finden. Das bedeutet, wenn $\widetilde{A}x = z$ nicht lösbar ist, so suchen wir ein x, so dass $\widetilde{A}x$ „möglichst nahe“ bei z liegt. Um dies durchführen zu können, müssen wir zunächst den Begriff „möglichst nahe“ wählen und mathematisch formulieren. Eine gute Wahl sollte ein sinnvoller Abstand sein und gleichzeitig eine einfache Lösung des resultierenden Problems zulassen. Hier bietet sich das sogenannten *Ausgleichsproblem*, auch *Methode der kleinsten Quadrate* genannt, an.

$$\text{Finde } x = (x_1, \ldots, x_n)^\top, \text{ so dass } \varphi(x) := \|z - \widetilde{A}x\|_2^2 \text{ minimal wird.} \tag{4.2}$$

Hierbei bezeichnet $\|y\|_2$ wie üblich die Euklidische Norm eine Vektors $y \in \mathbb{R}^n$, also $\|y\|_2 = \sqrt{\sum_{i=1}^n y_i^2}$.

4.1 Lösung mittels Normalengleichung

Eine Lösung dieses Problems ist aus der Analysis bekannt und führt auf die sogenannte *Normalengleichung*. Diese Gleichung erhält man, wenn man die Funktion φ zu minimieren versucht. Da φ stetig differenzierbar ist, kann ein Extremum durch Berechnung der Nullstelle der ersten Ableitung $\nabla\varphi$ bestimmt werden. Beachte, dass der Gradient der Funktion $g(x) := \|f(x)\|_2^2$ für beliebiges $f : \mathbb{R}^n \to \mathbb{R}^n$ durch $\nabla g(x) = 2\nabla f(x)^\top f(x)$ gegeben ist. Wir erhalten also

$$\nabla\varphi(x) = 2\widetilde{A}^\top\widetilde{A}x - 2\widetilde{A}^\top z.$$

Falls nun $\widetilde{A}$ vollen Spaltenrang besitzt, also $\operatorname{rang}(A) = n$, so ist die zweite Ableitung $\nabla^2\varphi(x) = 2\widetilde{A}^\top\widetilde{A}$ positiv definit und folglich jede Nullstelle $\nabla\varphi(x) = 0$ ein Minimum von φ. Es gilt also, dass ein Vektor x die Funktion φ genau dann minimiert, wenn die Normalengleichung

$$\widetilde{A}^\top\widetilde{A}x = \widetilde{A}^\top z \tag{4.3}$$

erfüllt ist. Das Ausgleichsproblem kann also wie folgt gelöst werden:

$$\begin{aligned} &\text{Löse} \quad Ax = b \\ &\text{mit} \quad A = \widetilde{A}^\top\widetilde{A} \text{ und } b = \widetilde{A}^\top z \end{aligned} \tag{4.4}$$

Das zunächst also recht kompliziert scheinende Minimierungsproblem (4.2) kann also auf die Lösung eines linearen Gleichungssystems (4.4) zurückgeführt werden. In MATLAB lässt sich dieses Problem nun mit den Mitteln der linearen Algebra sowie der Methoden aus Kapitel 3 oder **??** lösen:

```
% Script zur Demonstration der linearen Ausgleichsrechnung mit Hilfe der
% Normalengleichung
close all;
clear all;

% Aufstellen der Daten
% --------------------
AA = [1 2
      1 4
      1 6
      1 8 ]

```

```
13 z =  [ 111
14         185
15         317
16         387 ]
17
18 % Grafische Darstellung der Daten
19 % -------------------------------
20 plot(AA(:,2), z, 'ro');      % r = rote Farbe, o = Punktdarstellung
21 axis([0 10 0 500]);          % Skalierung der Achsen
22 hold on;                     % Hold Befehl, um weiter plotten zu koennen
23
24 % Warten auf Mausklick
25 ginput(1);
26
27 % Aufstellen der Normalengleichungen
28 % ---------------------------------
29 A = AA'*AA
30 b = AA'*z
31
32 % Loesen des Gleichungssystems
33 % ----------------------------
34 x=A\b
35
36 % Grafische Darstellung des Ergebnisses
37 % -------------------------------------
38 tt = 0:0.1:10;                  % Aufloesung der x-Achse
39 plot(tt,x(1)+x(2).*tt,'b-');    % b = blaue Farbe, - = Liniendarstellung
40
41 % Warten auf Mausklick
42 ginput(1);
43
44 % Direkte Loesung mit Matlab
45 % --------------------------
46 x1=AA\z
47
48 % Warten auf Usereingabe (d.h. zumindest auf RETURN)
49 pause;
50
51
52
53 % Sonderfall: alle Punkte liegen auf einer Parabel:
54 % -------------------------------------------------
55 t = [-2; -1; 0; 1; 2]
56 AA = [t.^0, t.^1, t.^2]
57 z = [17; 6; 1; 2; 9]
58
59 % Grafische Darstellung der Daten
60 % -------------------------------
61 figure;                  % neues Plotfenster oeffnen
62 plot(AA(:,2), z, 'ro'); % r = rote Farbe, o = Punktdarstellung
63 axis([-3 3 0 20]);       % Skalierung der Achsen
64 hold on;                 % Hold Befehl, um weiter plotten zu koennen
65
66 % Warten auf Mausklick
67 ginput(1);
68
69 % Aufstellen der Normalengleichung
70 % --------------------------------
71 A = AA'*AA
```

```
72 b = AA'*z
73
74 % Loesen des Gleichungssystems
75 % ----------------------------
76 x=A\b
77
78 % Grafische Darstellung des Ergebnisses
79 % --------------------------------------
80 tt = -3:0.1:3;                          % Aufloesung der x-Achse
81 plot(tt,x(1)+x(2).*tt+x(3).*tt.^2,'b-');% b = blaue Farbe, - = Liniendarstellung
```

Programm 4.1: Script zur Demonstration der linearen Ausgleichsrechnung mit Hilfe der Normalengleichung

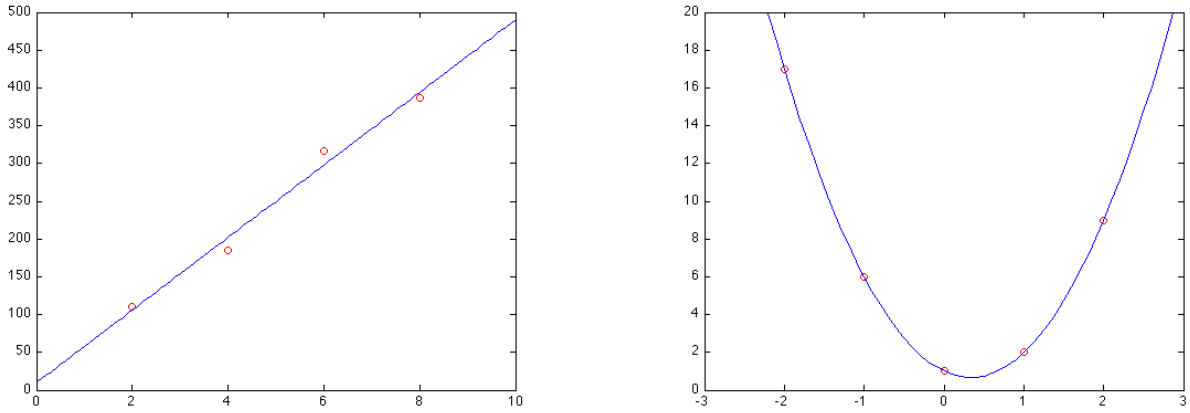


Abbildung 4.1: Ergebnisplots von Programm 4.1

Die Verwendung der Normalengleichung zur Lösung des linearen Ausgleichsproblems ist aber nicht immer ratsam. Hintergrund hierfür ist die Tatsache, dass die Matrix $\widetilde{A}^\top \widetilde{A}$ bedingt durch ihre Struktur sehr große Kondition haben kann. Die Verwendung der Normalengleichung ist also nicht robust.

4.2 Lösung mittels QR Zerlegung

Eine Alternative zur Lösung des linearen Ausgleichsproblems mit der Normalengleichung ist die Lösung mittels der QR Zerlegung aus Abschnitt 3.3. Wie wir bereits gesehen haben, ist die QR Zerlegung ein robustes Verfahren, so dass der Nachteil der Normalengleichung hier vermieden werden kann. Die Frage ist jedoch, wie kann man die QR Zerlegung genau zur Lösung des linearen Ausgleichsproblems nutzen?

Wie Algorithmus 3.9 zeigt, kann die QR Zerlegung auch auf nicht quadratische Matrizen $\widetilde{A}$ mit n Spalten und $m > n$ Zeilen angewandt werden. Als Resultat erhält man eine Zerlegung $\widetilde{A} = QR$ mit

$$R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}$$

wobei $R_1 \in \mathbb{R}^{n \times n}$ eine quadratische obere rechte Dreiecksmatrix ist. Beachte, dass die Normalengleichung genau dann lösbar ist, wenn $\widetilde{A}$ vollen Spaltenrang n besitzt, was wir annehmen. In diesem Fall ist die Matrix R_1 invertierbar.

Nun wählt man x so, dass der Vektor

$$s = Q^\top r = Q^\top z - Q^\top \widetilde{A} z$$

minimale $\|\cdot\|_2$–Norm besitzt, dann hat auch r minimale $\|\cdot\|_2$–Norm. Dies folgt, da wegen der Orthogonalität von $Q^\top$ die Gleichung $\|r\|_2 = \|s\|_2$ gilt. Wir zerlegen s in $s^{(1)} = (s_1, \ldots, s_n)^\top$ und $s^{(2)} = (s_{n+1}, \ldots, s_m)^\top$. Wegen der Form von $R = Q^\top \widetilde{A}$ ist der Vektor $s^{(2)}$ unabhängig von x und wegen

$$\|s\|_2^2 = \sum_{i=1}^{m} s_i^2 = \sum_{i=1}^{n} s_i^2 + \sum_{i=n+1}^{m} s_i^2 = \|s^{(1)}\|_2^2 + \|s^{(2)}\|_2^2$$

wird diese Norm genau dann minimal, wenn die Norm $\|s^{(1)}\|_2^2$ minimal wird.

Wir suchen also ein $x \in \mathbb{R}^n$, so dass

$$\|s^{(1)}\|_2 = \|y^{(1)} - R_1 x\|_2$$

minimal wird, wobei $y^{(1)}$ die ersten n Komponenten des Vektors $y = Q^\top z$ bezeichnet. Da R_1 eine invertierbare obere Dreiecksmatrix ist, kann man durch Rückwärtssubstitution eine Lösung x des Gleichungssystems $R_1 x = y$ berechnen, für die

$$\|s^{(1)}\|_2 = \|y^{(1)} - R_1 x\|_2 = 0$$

gilt, womit offenbar ein Minimum erreicht wird. Zusammenfassend kann man also das Ausgleichsproblem wie folgt mit der QR Zerlegung direkt lösen:

Algorithmus 4.1 (Lösung des Ausgleichsproblems mit der QR Zerlegung)
Gegeben sei eine Matrix $\widetilde{A} \in \mathbb{R}^{m \times n}$ mit $m > n$ und maximalem Spaltenrang n sowie ein Vektor $z \in \mathbb{R}^m$.

(1) Berechne die QR Zerlegung von $\widetilde{A}$ gemäß Algorithmus 3.9 und erhalte eine Matrix $R = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}$ wobei $R_1 \in \mathbb{R}^{n \times n}$ obere rechte Dreiecksform hat.

(2) Berechne $y = Q^\top z \in \mathbb{R}^m$ und setze $y^{(1)} = (y_1, \ldots, y_n)^\top \in \mathbb{R}^n$.

(2) Löse das Gleichungssystem $R_1 x = y^{(1)}$ durch Rückwärtssubstitution.

Geometrisch passiert bei diesem Algorithmus das Folgende: Das Bild von $\widetilde{A}$ wird durch die orthogonale Transformation $Q^\top$ längentreu auf den Unterraum $\mathrm{span}(e_1, \ldots, e_n)$ abgebildet. Der Vektor der rechten Seite wir ebenso zu $y = Q^\top z$ transformiert und die ersten n Einträge $y^{(1)}$ herausgenommen, wodurch auch $y^{(1)}$ im Unterraum $\mathrm{span}(e1, \ldots, e_n)$ liegt. In diesem Unterraum können wir das entstehende Gleichungssystem $R_1 x = y^{(1)}$ durch Rückwärtssubstitution lösen.

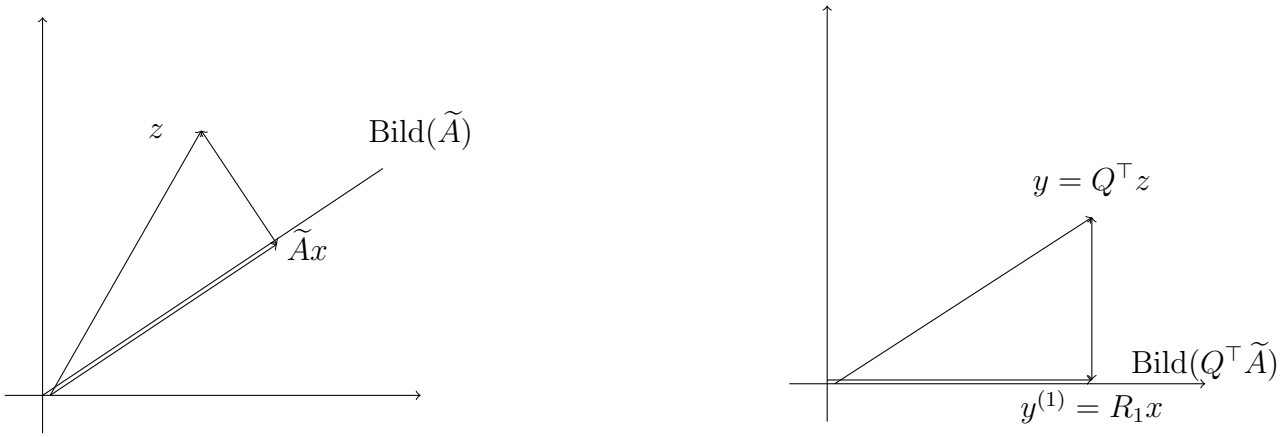


Abbildung 4.2: Veranschaulichung von Algorithmus 4.1

Kapitel 5

Eigenwertberechnung

Eigenwerte von Matrizen spielen in diversen Anwendungen, etwa der Lösung linearer Differentialgleichungen oder der Stabilität von Regelkreisen eine wichtige Rolle. Gesucht sind dabei diejenigen $\lambda \in \mathbb{C}$, für die die sogenannte *Eigenwertgleichung*

$$Av = \lambda v \tag{5.1}$$

für eine Matrix $A \in \mathbb{R}^{n\times n}$ und einen Vektor $v \in \mathbb{C}^n$ erfüllt ist. Dabei wird der Vektor $v \in C^n$ *Eigenvektor* genannt. Die zugehörigen Eigenwerte sind dabei die Nullstellen des charakteristischen Polynoms

$$p(\lambda) = \det(A - \lambda \mathrm{Id}).$$

Da die Funktion p für $n \geq 2$ nichtlinear ist, ist auch das Problem (5.1) *nichtlinear*.

Zur Eigenwertberechnung kann man also nichtlineare Gleichungssystemlöser wie etwa das Newton Verfahren, vergleiche Abschnitt 6.2, verwenden. Im Sinne von Stabilität und Konvergenzeigenschaften gibt es jedoch bessere Algorithmen, die die spezielle Struktur des Eigenwertproblems (5.1) ausnutzen. Solche Algorithmen zur Berechnung von Eigenwerten und zugehörigen Eigenvektoren werden wir im Folgenden behandeln.

Bevor wir jedoch zu numerischen Verfahren zur Berechnung von Eigenwerten kommen, wollen wir kurz die vielleicht naheliegendste Methode untersuchen: die Berechnung der Eigenwerte über die Nullstellen des charakteristischen Polynoms. Diese Methode ist numerisch äußerst schlecht konditioniert (unabhängig von der Kondition der Eigenwertberechnung selbst) und bereits kleinste Rundungsfehler können sehr große Fehler im Ergebnis nach sich ziehen.

Beispiel 5.1
Als Beispiel betrachten wir das Polynom

$$P(\lambda) = (\lambda - 1)(\lambda - 2)\cdots(\lambda - 20)$$

mit Nullstellen $\lambda_i = i$ für $i = 1, \ldots, 20$. Wenn diese Polynom als charakteristisches Polynom einer Matrix berechnet wird (z.B. ist es gerade das charakteristische Polynom $\Xi_A(\lambda)$ der Matrix $A = diag(1, 2, \ldots, 20)$), liegt es üblicherweise nicht in der obigen „Nullstellen“–Darstellung sondern in anderer Form vor, z.B. ausmultipliziert. Wenn man das obige $P(\lambda)$ ausmultipliziert, so ergeben sich Koeffizienten zwischen 1 für λ^{20} und $20! \approx 10^{20}$ für den konstanten Term. Stört man nun den Koeffizienten vor λ^{19}, der in der Größenordnung von 10^3 liegt mit dem sehr kleinen Wert $\varepsilon = 2^{-23} \approx 10^{-7}$, so erhält man für das gestörte Problem

$$\widetilde{P}(\lambda) = P(\lambda) - \varepsilon\lambda^{19}.$$

die folgenden Nullstellen

$$\begin{aligned}
\lambda_1 &= 1.000\,000\,000 \\
\lambda_2 &= 2.000\,000\,000 \\
\lambda_3 &= 3.000\,000\,000 \\
\lambda_4 &= 4.000\,000\,000 \\
\lambda_5 &= 4.999\,999\,928 \\
\lambda_6 &= 6.000\,006\,944 \\
\lambda_7 &= 6.999\,697\,234 \\
\lambda_8 &= 8.007\,267\,603 \\
\lambda_9 &= 8.917\,250\,249 \\
\lambda_{10/11} &= 10.095\,266\,145 \pm 0.643\,500\,904i \\
\lambda_{12/13} &= 11.793\,633\,881 \pm 1.652\,329\,728i \\
\lambda_{14/15} &= 13.992\,358\,137 \pm 2.518\,830\,070i \\
\lambda_{16/17} &= 16.730\,737\,466 \pm 2.812\,624\,894i \\
\lambda_{18/19} &= 19.502\,439\,400 \pm 1.940\,330\,347i \\
\lambda_{20} &= 20.846\,908\,101
\end{aligned}$$

Wie man an Beispiel 5.1 gut erkennen kann, ist das Problem der Eigenwertberechnung sehr schlecht konditioniert. Im vorliegenden Fall hat die kleine Störung nur nur beträchtliche Fehler bewirkt, sondern 10 Nullstellen sind durch die Störung komplex geworden.

5.1 Vektor– und Inverse Vektor–Iteration

Die einfachste Möglichkeit der Berechnung von Eigenwerten ist die Vektoriteration, die entweder als *direkte Iteration* (auch bekannt als *Von–Mises Iteration* oder *power iteration*) oder als *inverse Iteration* (auch *inverse power iteration*) durchführen lässt.

Hierzu betrachten wir reelle quadratische symmetrische Matrizen $A \in \mathbb{R}^{n\times n}$. Die Idee der direkten Iteration beruht darauf, für einen beliebigen Startwert $x^{(0)} \in \mathbb{R}^n$ die Iteration

$$x^{(i+1)} = Ax^{(i)} \tag{5.2}$$

durchzuführen. Für dieses einfache Verfahren lässt sich Folgendes zeigen:

Theorem 5.2 (Rayleigh Quotient)
Sei $A \in \mathbb{R}^{n\times n}$ eine reelle symmetrische Matrix und $\lambda_1 = \lambda_1(A)$ ein einfacher Eigenwert, für den die Ungleichung

$$|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$$

für alle anderen Eigenwerte $\lambda_j = \lambda_j(A)$ gilt. Sei weiterhin $x^{(0)} \in \mathbb{R}^n$ ein Vektor, für den $\langle x^{(0)}, v_1\rangle \neq 0$ für den zu λ_1 gehörigen normierten Eigenvektor v_1 gilt. Dann konvergiert die Folge

$$y^{(i)} := x^{(i)}/\|x^{(i)}\|$$

für $x^{(i)}$ aus (5.2) gegen $\pm v_1$, also gegen einen normierten Eigenvektor zum Eigenwert λ_1. Insbesondere konvergiert damit der sogenannte Rayleigh'sche Quotient

$$\lambda^{(i)} := \frac{{x^{(i)}}^\top A x^{(i)}}{{x^{(i)}}^\top x^{(i)}} = {y^{(i)}}^\top A y^{(i)} \tag{5.3}$$

gegen den Eigenwert λ_1.

Beachte, dass die Eigenwerte quadratischer symmetrischer reller Matrizen reellwertig sind.

5.1.1 Vektor–Iteration

Die Vektoriteration lässt sich also sehr einfach umsetzen, nämlich durch die wiederholte Auswertung des Rayleigh'schen Quotienten. Nun wissen wir zwar dank Theorem 5.2, dass die entstehende Folge konvergiert (wenn eine quadratische symmetrische Matrix A vorliegt). Wir wissen aber nicht, ob wir die Berechnung nach endlich vielen Schritten beenden können und dann den betragsgrößten Eigenwert λ_1 vorliegen haben. Im Gegenteil, es kann sein, dass wir diesen Wert noch nicht einmal theoretisch — also ohne Rundungsfehler — nach endlich vielen Schritten erhalten. Daher fügen wir dem folgenden Algorithmus einen **akzeptierbaren Verfahrensfehler** bei, indem wir die Iteration abbrechen, wenn sich die Norm der Differenz zweier aufeinander folgender Ergebnisse nur noch um einen Toleranzwert tol unterscheidet.

Bemerkung 5.3
Im Gegensatz zu den bisher besprochenen Algorithmen liegen hier also nicht nur Rundungsfehler, sondern auch Verfahrensfehler vor.

Eine Umsetzung der Vektor–Iteration gestaltet sich wie folgt:

Algorithmus 5.4
Gegeben sei eine reelle symmetrische Matrix $A \in \mathbb{R}^{n\times n}$, ein Vektor $y \in \mathbb{R}^n$ mit $\|y\|_2 = 1$ und eine Fehlerschranke tol $\in \mathbb{R}$.

(0) Setze $x := 0 \in \mathbb{R}^n$.

(1) Solange $\|y - x\|_2 \geq$ tol

  (a) Setze $x := y$

  (b) Setze $y := Ay$

  (c) Setze $y := y/\|y\|_2$

(2) Setze $\lambda_1 := y^\top A y$

Beachte, dass wir hier das Abbruchkriterium auf Basis der Werte von y gewählt haben. Dies ließe sich äquivalent auch über die Werte λ aus (5.3) bewerkstelligen, benötigt aber pro Iteration einmal die Auswertung von (5.3).

Dieses Verfahren hat mehrere Nachteile: Erstens erhalten wir nur den betragsmäßig größten Eigenwert $|\lambda_1|$ und den zugehörigen Eigenvektor, zweitens hängt die Konvergenzgeschwindigkeit

davon ab, wie schnell die Terme $|\lambda_j/\lambda_1|^i$, also insbesondere $|\lambda_2/\lambda_1|^i$ gegen Null konvergieren. Falls also $|\lambda_1| \approx |\lambda_2|$ und damit $|\lambda_2/\lambda_1| \approx 1$ gilt, ist nur eine sehr langsame Konvergenz zu erwarten.

5.1.2 Inverse Vektor–Iteration

Die *Inverse Vektor–Iteration* vermeidet sowohl den Nachteil der Einschränkung auf den betragsmaßig größten Eigenwert als auch der langsamen Konvergenz. Sei wiederrum $A \in \mathbb{R}^{n\times n}$ eine reelle symmetrische Matrix. Wir setzen voraus, dass wir einen Schätzwert $\tilde{\lambda} \in \mathbb{R}$ für einen Eigenwert $\lambda_j = \lambda_j(A)$ kennen, für den die Ungleichung

$$|\tilde{\lambda} - \lambda_j| < |\tilde{\lambda} - \lambda_k| \qquad \text{für alle } k = 1, \ldots, n,\ k \neq j$$

mit $\lambda_k = \lambda_k(A)$ gilt. Dann betrachten wir die Matrix $\widetilde{A} = (A - \tilde{\lambda}\mathrm{Id})^{-1}$. Diese besitzt zwar die gleichen Eigenvektoren wie die Matrix A, jedoch sind die zugehörigen Eigenwerte gegeben durch $1/(\lambda_k - \tilde{\lambda})$ für $k = 1, \ldots, n$. Somit ist nun $1/(\lambda_j - \tilde{\lambda})$ der betragsmäßig größte Eigenwert.

Die Inverse Vektor–Iteration ist nun gegeben durch

$$x^{(i+1)} = (A - \tilde{\lambda}\mathrm{Id})^{-1} x^{(i)}. \tag{5.4}$$

Aus Theorem 5.2 angewandt auf die Matrix $(A-\tilde{\lambda}\mathrm{Id})^{-1}$ statt A folgt, dass diese Iteration gegen einen normierten Eigenvektor v_j von $(A-\tilde{\lambda}\mathrm{Id})^{-1}$ zum Eigenwert $1/(\lambda_j - \tilde{\lambda})$ konvergiert. Wegen

$$\begin{aligned} (A - \tilde{\lambda}\mathrm{Id})^{-1} v_j &= 1/(\lambda_j - \tilde{\lambda}) v_j \\ \iff (\lambda_j - \tilde{\lambda}) v_j &= (A - \tilde{\lambda}\mathrm{Id}) v_j \\ \iff \lambda_j v_j &= A v_j \end{aligned}$$

ist dies gerade ein Eigenvektor von A zum Eigenwert λ_j. Die Konvergenzgeschwindigkeit ist bestimmt durch den Term

$$\left(\max_{\substack{k=1,\ldots,n \\ k\neq j}} \frac{|\lambda_j - \tilde{\lambda}|}{|\lambda_k - \tilde{\lambda}|} \right)^i,$$

d.h. je besser der Schätzwert, desto schneller die Konvergenz.

Die tatsächliche Implementierung der Iteration (5.4) ist hier etwas komplizierter als bei der direkten Vektor–Iteration (5.2). Während dort in jedem Schritt eine Matrix–Vektor Multiplikation mit Aufwand $O(n^2)$ durchgeführt werden muss, geht hier die Inverse $(A-\tilde{\lambda}\mathrm{Id})^{-1}$ ein. In der Praxis berechnet man nicht die Inverse, was wiederum ein schlecht konditioniertes Problem ist, sondern löst das lineare Gleichungssystem

$$(A - \tilde{\lambda}\mathrm{Id}) x^{(i+1)} = x^{(i)}, \tag{5.5}$$

wobei bei der Verwendung eines direkten Verfahrens die Matrix $(A - \tilde{\lambda}\mathrm{Id})$ nur einmal am Anfang der Iteration faktorisiert werden muss. Hierzu bietet sich die Choleski–Zerlegung an, da $(A - \tilde{\lambda}\mathrm{Id})$ symmetrisch ist. In allen weiteren Schritten ist lediglich einmal die Vorwärts– und Rückwärtssubstitution durchzuführen. Der Aufwand der Zerlegung ($O(n^3)$) kommt also hier zum Aufwand des Verfahrens hinzu, die einzelnen Iterationsschritte haben bei diesem Vorgehen allerdings keinen höheren Rechenaufwand als bei der direkten Iteration, da die Vorwärts–/Rückwärtssubstitution wie die matrix–Vektor Multiplikation den Aufwand $O(n^2)$ besitzen.

Dies führt zu folgenden Algorithmus:

Algorithmus 5.5
Gegeben sei eine reelle symmetrische Matrix $A \in \mathbb{R}^{n\times n}$, ein Vektor $y \in \mathbb{R}^n$ mit $\|y\|_2 = 1$, eine Fehlerschranke tol $\in \mathbb{R}^+$ und eine Schätzung des Eigenwertes $\tilde{\lambda}$.

(0) Setze $x := 0 \in \mathbb{R}^n$.

(1) Solange $\|y - x\|_2 \geq$ tol

  (a) Setze $x := y$

  (b) Löse das lineare Gleichungssystem $(A - \tilde{\lambda}\mathrm{Id})y = y$

  (c) Setze $y := y/\|y\|_2$

(2) Setze $\lambda := y^\top Ay$

Für sehr gute Schätzwerte $\tilde{\lambda} \approx \lambda_j$ wird die Matrix $(A - \tilde{\lambda}\mathrm{Id})$ „fast“ singulär (für $\tilde{\lambda} = \lambda_j$ wäre sie singulär), weswegen die Kondition von $(A - \tilde{\lambda}\mathrm{Id})$ sehr groß wird. Wegen der besonderen Struktur des Algorithmus führt dies hier aber nicht auf numerische Probleme, da zwar die Lösung y des Gleichungsystems (5.5) mit großen Fehlern behaftet sein kann, sich diese Fehler aber in der hier eigentlich wichtigen *normierten Lösung* $y/\|y\|$ nicht auswirken. Hierzu betrachten wir folgendes Beispiel:

Beispiel 5.6
Gegeben sei die Matrix

$$A = \begin{pmatrix} -1 & 3 \\ -2 & 4 \end{pmatrix}$$

mit den Eigenwerten $\lambda_1(A) = 2$ *und* $\lambda_2(A) = 1$. *Wir wählen* $\tilde{lambda} = 1 - \varepsilon$ *für* $0 < \varepsilon \ll 1$. *Dann ist die Matrix*

$$(A - \tilde{\lambda}Id) = \begin{pmatrix} -2+\varepsilon & 3 \\ -2 & 3+\varepsilon \end{pmatrix} \quad mit \quad (A - \tilde{\lambda}Id)^{-1} = \frac{1}{\varepsilon(\varepsilon+1)} \underbrace{\begin{pmatrix} 3+\varepsilon & -3 \\ 2 & -2+\varepsilon \end{pmatrix}}_{=:B}$$

fast singulär und man sieht leicht, dass für die Kondition z.B. in der Zeilensummennorm die Abschätzung $cond_\infty(A - \tilde{\lambda}Id) > 1/\varepsilon$ *gilt. Trotzdem ist die Berechnung von* $y := y/\|y\|_2$ *mittels* (5.5) *nicht stark anfällig für Rundungsfehler, denn es gilt*

$$y := \frac{(A - \tilde{\lambda}Id)^{-1}y}{\|(A - \tilde{\lambda}Id)^{-1}y\|_2} = \frac{By}{\|By\|_2}$$

d.h. der ungünstige Faktor $1/(\varepsilon(\varepsilon+1))$ *kürzt sich heraus. Zudem bewirkt die Normierung eine weitere Erhöhung der Robustheit gegenüber Fehlern: Betrachten wir beispielsweise* $y = (1,0)^\top$ *und* $\varepsilon = 1/1000$, *so erhält man*

$$By = \begin{pmatrix} 3.001 \\ 2 \end{pmatrix} \quad und \quad y = \frac{By}{\|By\|_2} = \begin{pmatrix} 0.8321356035 \\ 0.5545722116 \end{pmatrix}$$

während man für die gestörte Lösung mit $\tilde{y} = (1.1, 0.1)^\top$

$$B\tilde{y} = \begin{pmatrix} 3.6011 \\ 2.4001 \end{pmatrix} \quad und \quad \tilde{y} = \frac{B\tilde{y}}{\|B\tilde{y}\|_2} = \begin{pmatrix} 0.8321178327 \\ 0.5545988754 \end{pmatrix}$$

erhält. Die Störung in der ersten Nachkommastelle der rechten Seite wirkt sich in $B\tilde{y}$ also in der ersten, in dem für den Algorithmus wichtigen normierten Vektor $\tilde{y} = B\tilde{y}/\|B\tilde{y}\|_2$ aber erst in der fünften Nachkommastelle aus.

Wie man bereits erkennen kann, ist die Vektor– bzw. Inverse Vektor–Iteration gut geeignet, alle Eigenwerte einer symmetrischen reellen Matrix A zu berechnen. Nachteil hierbei ist allerdings, dass entsprechende Schätzwerte bekannt sein oder beschafft werden müssen.

5.2 QR Faktorisierung

Im Gegensatz zur Vektor– bzw. Inversen Vektor–Iteration benötigt die QR Faktorisierung keine Schätzwerte der Eigenwerte der Matrix A. Zudem werden mit der QR Faktorisierung alle Eigenwerte in einer einzigen Rechnung bestimmt. Wie bereits bei der Lösung von linearen Gleichungssystemen wird auch hier die Faktorisierung mittels orthogonaler Matrizen eine wichtige Rolle spielen.

Die Grundidee des Verfahrens, dass wir hier nur für symmetrische reelle Matrizen untersuchen, ist, dass für derartige Matrizen eine Orthonormalbasis aus Eigenvektoren $v_1, \ldots, v_n$ besteht, so dass für die orthogonale Matrix $Q = (v_1, \ldots, v_n) \in \mathbb{R}^{n\times n}$ die Gleichung

$$Q^\top AQ = \Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$$

gilt, wobei λ_i wieder die Eigenwerte der Matrix A sind. Wenn wir also eine solche orthogonale Matrix Q finden können, mit der sich A auf Diagonalgestalt konjugieren lässt, so können wir die Eigenwerte direkt aus der Diagonalen der resultierenden Matrix Λ und die Eigenvektoren aus den Spalten der Matrix Q ablesen. Leider ist eine direkte Transformation auf Diagonalgestalt nicht möglich. Insbesondere lässt sich die QR Zerlegung nicht hierfür verwenden, da man damit zwar eine Matrix R in oberer Dreiecksform erhält, die resultierenden Nulleinträge aber durch die Multiplikation mit Q von rechts wieder zunichte gemacht werden.

Die benötigte Transformation lässt sich aber iterativ bestimmen. Dabei empfiehlt es sich, die Matrix A in einem vorbereitenden Schritt zunächst auf eine möglichst einfache Form zu bringen, um die Anzahl der Rechenoperationen pro Iteration klein zu halten. Hierzu wählt man in diesem Fall die *Tridiagonalgestalt*, d.h. das sowohl die Diagonale also auch eine Diagonale oberhalb wie unterhalb der Diagonalen nicht mit Nulleinträgen besetzt sind, der Rest hingegen aus Nullen besteht. Diese Form lässt sich wieder mit Hilfe von *Householder Transformationen* konstruieren. Da diese nicht ganz genauso berechnet werden, wie dies in Algorithmus 3.9 der Fall war, werden wir sie hier nicht mit Q sondern mit P bezeichnen. Die änderungen an Algorithmus 3.9 sind dabei minimal: Man lässt den Iterationsindex j in Schritt (1) lediglich von 2 bis $n-1$ laufen und erniedrigt alle *Spaltenindizes* in den folgenden Berechnungen um 1, d.h. a_{jj} wird durch a_{jj-1} und alle a_{ij} durch a_{ij-1} ersetzt. Falls gewünscht kann zudem die Berechnung von R durch $P^{(j)}A^{(j)}{P^{(j)}}^\top$ ersetzt werden.

Die iterative Berechnung von Q kann nun durch folgenden Algorithmus geschehen:

Algorithmus 5.7
Gegeben sei eine symmetrische reelle Matrix $A \in \mathbb{R}^{n\times n}$.

(0) Setze $A^{(1)} := P^\top AP$ (mit P aus modifizierter QR Zerlegung 3.9) und $i := 1$

(1) Berechne eine QR Zerlegung $A^{(i)} = Q^{(i)}R^{(i)}$

(2) Setze $A^{(i+1)} := R^{(i)}Q^{(i)}$, $i := i+1$ und gehe zu (1)

An dieser Stelle noch einige Bemerkungen zu der QR Faktorisierung:

Bemerkung 5.8
(1) Tatsächlich ist es nicht notwendig, dass $A^{(1)}$ in Tridiagonalform vorliegt, es reduziert aber den benötigten Rechenaufwand erheblich da alle folgenden Matrizen $A^{(i)}$ wiederum in Tridiagonalform sind und leicht QR zerlegbar sind.
(2) Wie bereits erwähnt können die Eigenwerte am Ende der Iteration aus der Diagonalen von $A^{(i)}$ abgelesen werden. Der Algorithmus funktioniert dabei auch für mehrfache Eigenwerte, es können aber Blöcke übrigbleiben, wenn Eigenwerte λ_i und λ_j mit $\lambda_i = -\lambda_j$ existieren.
(3) Als Abbruchkriterium empfiehlt sich die Größe der Nichtdiagonalelemente von $A^{(i)}$.

Der Algorithmus konvergiert wie die Vektor–Iteration langsam, wenn betragsmäßig nahe beieinanderliegende Eigenwerte existieren. Abhilfe können hierbei sogenannte *Shiftstrategien* bringen, auf die wir hier aber nicht eingehen werden.

Kapitel 6

Lösung nichtlinearer Gleichungssysteme

Wie wir bereits in Kapitel 5 gesehen haben, gibt es neben den linearen Gleichungssystemen auch nichtlineare Probleme, die zu lösen sind. Typischerweise werden die Lösungen nichtlinearer Gleichungen über die Nullstelllen einer Funktion $f : \mathbb{R}^n \to \mathbb{R}^n$ definiert, für die dann ein $x^\star \in \mathbb{R}^n$ mit

$$f(x^\star) = 0 \tag{6.1}$$

gesucht wird.

Ein einfaches Beispiel einer solchen nichtlinearen Gleichung ist die Berechnung der Quadratwurzel:

Beispiel 6.1
Berechne $x^\star \in \mathbb{R}$ mit $f(x) =$ für $f(x) = x^2 - 2$. Die eindeutige reelle Lösung ist $\sqrt{2}$, ein numerisches Verfahren zur Berechnung von Nullstellen kann also insbesondere zur Berechnung von Quadratwurzeln herangezogen werden.

Beachte, dass $\sqrt{2}$ keine Maschinenzahl ist, vgl. Abschnitt 2.2. Wir müssen hier, ähnlich wie in der iterativen Berechnung von Eigenwerten in Abschnitt 5.1, also oft damit leben, dass nicht das exakte Ergebnis berechnet werden kann und einen Verfahrensfehler zulassen.

6.1 Fixpunkt Iteration

Die wohl einfachste Methode zur Lösung eines nichtlineare Gleichungssystems in Nullstellenform (6.1) ist die sogenannte *Fixpunkt Iteration*. Die Idee dieser Methode beruht darauf, die Lösung eines nichtlinearen Gleichungssystems als Fixpunktgleichung umzuformulieren, also die Abbildung

$$g(x) = f(x) + x \tag{6.2}$$

zu betrachten. Für die Abbildungen $f, g : \mathbb{R}^n \to \mathbb{R}^n$ gilt, dass $f(x^\star) = 0$ äquivalent ist zu $g(x^\star) = x^\star$. Ein Punkt $x \in \mathbb{R}^n$ mit $g(x) = x$ heißt *Fixpunkt* von g. Statt einer Nullstelle der Funktion f können wir also alternativ einen Fixpunkt der Funktion g suchen.

Die Hilfsmittel hierfür stellt der Banach'sche Fixpunktsatz bereit:

Theorem 6.2 (Banach'scher Fixpunktsatz)
Sei A eine Teilmenge eines vollständigen normierten Raumes mit Norm $\|\cdot\|$ und sei $\Phi : A \to A$ eine Kontraktion, d.h. es existiert eine Konstante $k \in (0,1)$, so dass die Ungleichung

$$\|\Phi(x) - \Phi(y)\| \leq k\|x - y\|$$

für alle $x, y \in A$ gilt. Dann existiert ein eindeutiger Fixpunkt $x^\star \in A$, gegen den alle Folgen der Form $x^{(i+1)} = \Phi(x^{(i)}$ mit beliebigem $x^{(0)} \in A$ konvergieren. Darüberhinaus gelten die a priori und a posteriori Abschätzungen

$$\|x^{(i)} - x^\star\| \leq \frac{k^i}{1-k}\|x^{(1)} - x^{(0)}\| \quad \textit{und} \quad \|x^{(i)} - x^\star\| \leq \frac{k}{1-k}\|x^{(i)} - x^{(i-1)}\|.$$

Theorem 6.2 garantiert dann die Genauigkeit

$$\|x^{(i+1)} - x^\star\| \leq \frac{k}{1-k}\text{tol}.$$

Hierbei muss man sich merken, dass nicht jede Abbildung g die Voraussetzungen von Theorem 6.2 erfüllt. Falls g jedoch differenzierbar ist, ist die Kontraktionseigenschaft gegeben, falls

$$\sup_{x \in A} \|\nabla g(x)\|_\infty =: k < 1$$

gilt, wobei $\nabla g(x)$ hier die Jacobi–Matrix, also die matrixwertige Ableitung der Funktion g an der Stelle x bezeichnet und $\|\cdot\|_\infty$ die von der ∞–Norm induzierte Zeilensummennorm für Matrizen ist.

Falls die zuvor genannte Funktion g also die Bedingungen des Banach'schen Fixpunktsatzes auf einer Umgebung A von $x^\star$ erfüllt, so konvergiert folgendes Verfahren für alle $x^{(0)} \in A$:

Algorithmus 6.3 (Fixpunkt Iteration)
Gegeben sei eine Funktion $g : \mathbb{R}^n \to \mathbb{R}^n$, ein Anfangswert $x^{(0)} \in \mathbb{R}^n$, eine Fehlerschranke $\text{tol} \in \mathbb{R}^+$ und eine maximale Iterationszahl $\text{maxiter} \in \mathbb{N}$.

(0) Setze $i := 0$ und $x^{(-1)} := x^{(0)} + 2\text{tol}$

(1) Solange $\|x^{(i)} - x^{(i-1)}\| \geq \text{tol}$ und $i \leq \text{maxiter}$

  (a) Setze $x^{(i+1)} := g(x^{(i)})$

  (b) Setze $i := i + 1$

Zur Veranschaulichung betrachten wir folgendes Beispiel:

Beispiel 6.4
Gegeben sei der zweigelenkige Roboterarm aus Abbildung 6.1. Die Aufgabe besteht dabei darin, eine Winkelkonfiguration $x = (\alpha, \beta)$ zu finden, so dass der Greifer des Arms in die Position (p_1, p_2) gebracht wird. Das zu lösende Fixpunktproblem $f(x) = x$ ist hierbei gegeben durch

$$\begin{aligned} f(\alpha,\beta) &= \begin{pmatrix} l_1\cos(\alpha) + l_2\cos(\beta - (\pi - \alpha)) - p_1 + \alpha \\ l_1\sin(\alpha) + l_2\sin(\beta - (\pi - \alpha)) - p_2 + \beta \end{pmatrix} \\ &= \begin{pmatrix} l_1\cos(\alpha) - l_2\cos(\alpha + \beta) - p_1 + \alpha \\ l_1\sin(\alpha) - l_2\sin(\alpha + \beta) - p_2 + \beta \end{pmatrix}. \end{aligned}$$

Hier verwenden wir die Werte $l_1 = 2$, $l_2 = 1$ *mit Zielkonfiguration* $(p_1, p_2) = (1, 2)$, *der aufrechten Position als Startwert sowie einer geforderten Genauigkeit von* 10^{-10} *zum Start der Fixpunkt Iteration, vgl. Programme 6.1, 6.2 und 6.3. Nach* 55 *Iterationsschritten erhalten wir hiermit die Lösung* $(\alpha, \beta) = (0.6435, 4.7124)$.

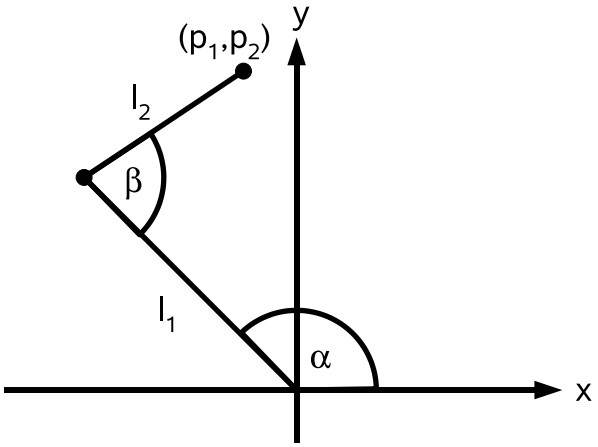


Abbildung 6.1: Konfiguration eines zweigelenkigen Roboterarms

```
% Loese mit Fixpunkt Iteration
% ----------------------------

% Setze Startwerte und Genauigkeiten
x=zeros(2,1);
x(1)=pi/2.;
x(2)=pi;
maxiter=100;
tol=1.e-10;

% Fuehre Fixpunktiteration aus
[x,opt]=fixpunktiteration('roboterfixpunktfunktion',x,tol,maxiter);

% Gebe Ergebnis aus
x
```

Programm 6.1: Script zur Demonstration der Fixpunktiteration anhand des Roboterbeispiels 6.4

```
% Musterloesung zu Numerische Rechneranwendungen im Wintertrimester 2012
%
% Funktion zur Auswertung des Roboter Fixpunktproblems
%
% Input:
% x     Auswertpunkt
%
% Output:
% f     Funktionswert
%
% Autor: Juergen Pannek
```

```
12
13 function [f] = roboterfixpunktfunktion(x)
14
15 n = size(x, 1);
16 f = zeros(n, 1);
17
18 f(1) = 2. * cos(x(1)) - cos(x(1) + x(2)) - 1. + x(1);
19 f(2) = 2. * sin(x(1)) - sin(x(1) + x(2)) - 2. + x(2);
```

Programm 6.2: Fixpunkt Funktion des Roboterbeispiels 6.4

```
1  % Musterloesung zu Numerische Rechneranwendungen im Wintertrimester 2012
2  %
3  % Fixpunktiteration im Sinne des Banachschen Fixpunktsatzes
4  %
5  % Input:
6  % func    Fixpunktfunktion
7  % x       Startpunkt
8  % tol     Fehlertolernz
9  % maxiter Maximale Iterationszahl
10 %
11 % Output:
12 % x       letzter berechneter Punkt
13 % opt     =1 ==> Optimalitaet errreicht
14 %
15 % Autor: Juergen Pannek
16
17 function [x,opt]=fixpunktiteration(func,x,tol,maxiter)
18
19 xalt=x;
20
21 opt=0;
22
23 for iter=1:maxiter
24     [x]=feval(func,x);
25     normx=norm(x-xalt);
26     fprintf(1,'\n iter=%3d    ||x-xalt||=%e',iter,normx);
27     if (normx < tol)
28         break;
29     end
30     xalt=x;
31 end
32
33 if (iter==maxiter)
34     fprintf(1,'\nWarning: Reached maximal number of iterations');
35 end
```

Programm 6.3: Funktion zur Aufführung der Fixpunktiteration

An dieser Stelle noch einige Bemerkungen zu der Fixpunkt Iteration:

Bemerkung 6.5
(1) Die Fixpunkt Iteration ist eines der wenigen Verfahren für nichtlineare Gleichungssysteme das ohne zusätzliche Informationen wie Ableitungen etc. auskommt.

(2) Falls $\nabla g(x^\star) \neq 1$ ist und g statt einer Kontraktion eine Expansion ist, kann die Iterationsvorschrift $x^{(i+1)} = g^{-1}(x^{(i)})$ verwendet werden.
(3) Das Verfahren konvergiert lediglich linear.

Aufgrund des letzten Punktes, der langsamen Konvergenz, verwendet man im Allgemeinen ein Verfahren, dass schneller konvergiert, das sogenannte Newton Verfahren.

6.2 Newton Verfahren

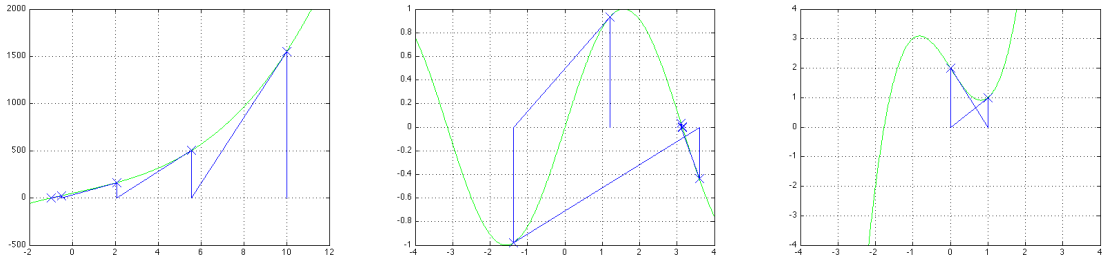
Ein weiteres Verfahren zur Lösung nichtlinearer Gleichungssysteme ist das sogenannten *Newton Verfahren*. Im Vergleich zur Fixpunkt Iteration konvergiert dieses deutlich schneller, nämlich quadratisch. Der Preis, den man hierfür zahlen muss, ist die Notwendigkeit von Ableitungen.

Die Idee des Newton Verfahrens ist wie folgt: Berechne eine Tangente $g(x)$ von f im Punkt $x^{(i)}$, d.h. die Gerade

$$g(x) = f(x^{(i)}) + \nabla f(x^{(i)}) \cdot (x - x^{(i)})$$

und wähle $x^{(i+1)}$ als Nullstelle von g, also

$$\begin{aligned} & f(x^{(i)}) + \nabla f(x^{(i)}) \cdot (x - x^{(i)}) \\ \iff \quad & \nabla f(x^{(i)}) x^{(i+1)} = \nabla f(x^{(i)}) x^{(i)} - f(x^{(i)}) \\ \iff \quad & x^{(i+1)} = x^{(i)} - \nabla f(x^{(i)})^{-1} f(x^{(i)}). \end{aligned}$$



(a) $f(x) = x^3 + 50x + 50$ (b) $f(x) = \sin(x)$ (c) $f(x) = x^3 - 2x + 2$

Abbildung 6.2: Newton Schritte für verschiedene Funktionen

Beachte, dass im letzten Schritt die Inverse verwendet werden muss, da man nicht durch eine Matrix teilen kann. In einer praktischen Umsetzung berechnet man an dieser Stelle allerdings nicht die Inverse der Jacobi–Matrix $\nabla f(x^{(i)})^{-1}$, sondern löst das lineare Gleichungssystem

$$\nabla f(x^{(i)}) \Delta x^{(i)} = -f(x^{(i)})$$

und erhält anschließend den nächsten Iterationswert durch $x^{(i+1)} = x^{(i)} + \Delta x$.

Formal lässt sich der Algorithmus im $\mathbb{R}$ also wie folgt beschreiben:

Algorithmus 6.6 (Newton Verfahren)
Gegeben sei eine Funktion $f : \mathbb{R}^n \to \mathbb{R}^n$, die Ableitung $\nabla f : \mathbb{R}^n \to \mathbb{R}^{n \times n}$, ein Anfangswert $x^{(0)} \in \mathbb{R}^n$, eine Fehlerschranke tol $\in \mathbb{R}^+$ und eine maximale Iterationszahl maxiter $\in \mathbb{N}$.

(0) Setze $i := 0$ und $x^{(-1)} := x^{(0)} + 2\text{tol}$

(1) Solange $\|x^{(i)} - x^{(i-1)}\| \geq \text{tol}$ und $i \leq \text{maxiter}$

(a) Löse das lineare Gleichungssystem $\nabla f(x^{(i)})\Delta x^{(i)} = -f(x^{(i)})$

(b) Setze $x^{(i+1)} = x^{(i)} + \Delta x$.

(c) Setze $i := i + 1$

An dieser Stelle einige Bemerkungen zum Newton Verfahren:

Bemerkung 6.7
(1) Falls die Funktion f an der Nullstelle $x^\star$ zweimal stetig partiell differenzierbar ist und die Jacobi–Matrix $\nabla f(x^\star)$ invertierbar ist, so konvergiert das Verfahren lokal quadratisch.
(2) Ein wesentlicher Nachteil des Newton Verfahrens ist, dass die Ableitung der Funktion f benötigt wird. Diese kann man zwar durch eine geeignete numerische Näherung ersetzen und damit die expliziter Berechnung von $\nabla f(x^{(i)})$ vermeiden. Hierzu verwendet man die sogenannten Vorwärts-, Rückwärts- oder Zentralen Differenzen

$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x)}{h}$$
$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x - he_i) - f(x)}{h}$$
$$\frac{\partial f}{\partial x_i}(x) \approx \frac{f(x + he_i) - f(x - he_i)}{2h}$$

Die numerische Approximation von Ableitungen ist aber im Allgemeinen sehr anfällig gegenüber Rundungsfehlern, weswegen die numerische Stabilität dieser Verfahren (möglichst unter Hinzunahme weiterer Informationen über f) genau geprüft werden sollte.

Kapitel 7

Numerische Interpolation

Die Interpolation befasst sich mit dem Problem der Angabe von approximierten Zwischenwerten. Dies bedeutet etwa für eine gegebene Funktion, dass man diese nicht an einer bestimmten Stelle auswerten möchte, etwa weil die Auswertung der Funktion zu teuer wäre. Stattdessen bestimmt man eine interpolierende Funktion, die die Funktion approximiert und leicht auswertbar ist. Dies ist die sogenannte *Funktionsinterpolation*. Ein gängiges einfaches Beispiel der Funktionsinterpolation ist die in der Stochastik und Statistik oft benötigte Gauss–Verteilungsfunktion

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{x} e^{-y^2/2} dy,$$

für die keine geschlossene Formel existiert.
Ähnlich wie bei der Funktionsinterpolation verhält es sich auch bei der sogenannten *Dateninterpolation*. Hier hat man keinen funktionalen Zusammenhang, sondern lediglich Messwerte gegeben und möchte an „Nichtmesspunkten“ approximierte Messwerte bestimmen. Da die Messpunkte bei der Funktionsinterpolation vorgegeben werden können, ist die Funktionsinterpolation ein Spezialfall der Dateninterpolation.

Das Problem der Interpolation ist also das Folgende: Gegeben sei eine Menge von Paaren (x_i, f_i) für $i = 0, \ldots, n$, die etwa durch Auswertung der zu interpolierenden Funktion oder als Messdaten aus Experimenten gewonnen wurden. Nun wird eine einfach auszuwertende Funktion F gesucht, für die die Gleichung

$$F(x_i) = f_i \qquad \text{für } i = 0, \ldots, n \tag{7.1}$$

gilt.

In diesem Kapitel werden wir Verfahren entwickeln, die das Dateninterpolationsproblem und somit auch den Spezialfall dess Funktionsinterpolationsproblems lösen können. Die Wichtigkeit des Spezialfalls liegt in deisem Zusammenhang darin, dass man bei der Interpolation einer Funktion f in natürlicher Weise einen *Interpolationsfehler* über den Abstand zwischen f und F definieren kann, und so ein Maß für die Güte des Verfahrens erhält. Daher werden wir Verfahren betrachten, die speziell auf die Funktioninterpolation zugeschnitten sind. Da hierbei die Wahl der sogenannten *Stützstellen* x_i aus dem Verfahren hervorgeht, also insbesondere nicht frei wählbar ist, sind diese Methoden nicht für die Dateninterpolation anwendbar. Ebenso ist bei der Dateninterpolation kein Gütemaß angebbar, da es keine Funktion f gibt bezüglich der man den Fehler messen könnte.

7.1 Polynominterpolation

Eine einfache und (oft) effektive Methode zur Interpolation ist die Wahl von F als Polynom, also als Funktion der Form

$$P(x) = a_0 + a_1 x^1 + a_2 x^2 + \ldots + a_m x^m. \tag{7.2}$$

Hierbei werden die Werte a_i, $i = 0, \ldots, m$ als *Koeffizienten* des Polynoms bezeichnet. Die höchste auftretenden Potenz (hier m falls $a_m \neq 0$) heißt *Grad* des Polynoms. Um zu betonen, dass wir hier ausschließlich Polynome verwenden, schreiben wir in diesem Abschnitt „P“ statt „F“ für die Interpolationsfunktion. Den Raum der Polynome vom Grad $\leq m$ bezeichnen wir mit $\mathcal{P}_m$. Dieser Funktionenraum ist ein $m+1$ dimensionaler Vektorraum über $\mathbb{R}$ bzw. $\mathbb{C}$ mit Basis $\mathcal{B} = \{1, x, \ldots, x^m\}$, da Addition von Polynomen und Multiplikation mit Skalaren wieder ein Polynom desselben Grads ergeben. Andere Basen dieses Vektorraums sind dabei ebenso wählbar.

Das Problem der Polynominterpolation liegt nun darin, ein Polynom P zu bestimmen, das (7.1) erfüllt. Zunächst einmal müssen wir uns dazu überlegen, welchen Grad das gesuchte Polynom haben soll. Dazu kann folgendes Theorem verwendet werden:

Theorem 7.1
Sei $n \in \mathbb{N}$ und seien Daten (x_i, f_i) für $i = 0, \ldots, n$ gegeben, so dass $x_i \neq x_j$ für alle $i \neq j$ gilt, also die Stützstellen paarweise verschieden sind. Dann gibt es genau ein Polynom $P \in \mathcal{P}_n$ von Grad $\leq n$, das die Bedingung

$$P(x_i) = f_i \qquad \text{für } i = 0, \ldots, n \tag{7.3}$$

erfüllt.

Entsprechend „passt“ zu $n+1$ gegebenen Datenpunkten also ein Polynom vom Grad n. Das Problem $P(x_i) = f_i$ für $i = 0, \ldots, n$ lässt sich äquivalent durch das lineare Gleichungssystem

$$\begin{pmatrix} 1 & x_0 & \cdots & x_0^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_0 \\ \vdots \\ f_n \end{pmatrix}$$

ausdrücken. Nun könnte man die Algorithmen aus Kapitel 3 dazu verwenden, dieses Problem zu lösen. Hier werden wir jedoch andere, effizientere Herangehensweisen untersuchen, wie das Problem der Bestimmung geeigneter Parameter $a_0, \ldots a_n$ gelöst werden kann.

Das Haupthilfsmittel, das wir hierzu verwenden werden, sind die sogenannten *Lagrange Polynome*, die im Prinzip nur eine geschickte Art der Darstellung sind. Für die gegebenen Stützstellen $x_0, \ldots, x_n$ definieren wir die Lagrange Polynome für $i = 0, \ldots, n$ als

$$L_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}. \tag{7.4}$$

Wie man leicht nachrechnen kann, sind Lagrange Polynome der Form (7.4) Polynome vom Grad n und erfüllen

$$L_i(x_k) = \begin{cases} 1 & \text{für } i = k \\ 0 & \text{für } i \neq k \end{cases}.$$

Somit folgt die Aussage:

Theorem 7.2
Seien Daten (x_i, f_i) für $i = 0, \ldots, n$ mit $x_i \neq x_j$ für $i \neq j$ gegeben. Dann ist das eindeutige Interpolationspolynom $P(x)$, das (7.3) erfüllt, gegeben durch

$$P(x) = \sum_{i=0}^{n} f_i L_i(x). \tag{7.5}$$

Die Lagrange Polynome sind orthogonal (sogar orthonormal) bezüglich des Skalarproduktes $\langle P.Q\rangle := \sum_{i=0}^{n} P(x_i)Q(x_i)$ auf dem Raum der Polynome $\mathcal{P}_n$. Somit bilden sie eine Orthonormalbasis von $\mathcal{P}_n$ bezüglich dieses Skalarproduktes, da sich jedes Polynom vom Grad $\leq n$ mittels

$$P = \sum_{i=0}^{n} P(x_i)L_i = \sum_{i=0}^{n} \langle P, L_i\rangle L_i$$

als Summe der L_i schreiben lässt. Im Zuge der Funktionsinterpolation werden wir sehen, dass Orthogonalität eine nützliche Eigenschaft ist.

Beispiel 7.3
Betrachte die Daten $(3, 68)$, $(2, 16)$, $(5, 352)$. Die zugehörigen Lagrange Polynome sind gegeben durch

$$\begin{aligned} L_0(x) &= \frac{x-2}{3-2}\frac{x-5}{3-5} = -\frac{1}{2}(x-2)(x-5) \\ L_1(x) &= \frac{x-3}{2-3}\frac{x-5}{2-5} = \frac{1}{3}(x-3)(x-5) \\ L_2(x) &= \frac{x-2}{5-2}\frac{x-3}{5-3} = \frac{1}{6}(x-2)(x-3) \end{aligned}$$

Somit erhalten wir

$$P(x) = -68 \cdot \frac{1}{2}(x-2)(x-5) + 16 \cdot \frac{1}{3}(x-3)(x-5) + 352 \cdot \frac{1}{6}(x-2)(x-3)$$

und es gilt $P(3) = 68$, $P(2) = 16$ sowie $P(5) = 352$.

Zählt man die notwendigen Operation ab, so sieht man, dass die direkte Auswertung der Polynoms P in dieser Form den Aufwand $\mathcal{O}(n^2)$ besitzt, also deutlich effizienter als die Lösung eines lineare Gleichungssystems ist ($\mathcal{O}(n^3)$). Für die effiziente direkte Auswertung sollte man die Nenner der Lagrange Polynome vorab berechnen und speichern, damit diese nicht bei jeder Auswertung von P erneut berechnet werden müssen.

Noch wesentlich effizienter lassen sich die Lagrange Polynome auswerten, wenn deren Darstellung geschickt umformuliert wird. Dazu schreiben wir den Zähler von (7.4) als

$$\frac{\ell(x)}{x - x_i} \qquad \text{mit} \quad \ell(x) := \prod_{j=0}^{n} x - x_j.$$

Den Nenner schreiben wir mittels der sogenannten *baryzentrischen Koordinaten*

$$w_i := \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{1}{x_i - x_j}.$$

Dann gilt $L_i(x) = \ell(x)\frac{w_i}{x-x_i}$ und damit

$$P(x) = \sum_{i=0}^{n} L_i(x) f_i = \sum_{i=0}^{n} \ell(x) \frac{w_i}{x - x_i} f_i = l(x) \sum_{i=0}^{n} \frac{w_i}{x - x_i} f_i.$$

Beispiel 7.4
Betrachte wiederum Beispiel 7.3. Das zugehörige ℓ ist gegeben durch

$$\ell(x) = (x-2)(x-3)(x-5)$$

und die w_i ergeben sich als

$$\begin{aligned} w_0 &= \frac{1}{3-2}\frac{1}{3-5} = -\frac{1}{2} \\ w_0 &= \frac{1}{2-3}\frac{1}{2-5} = \frac{1}{3} \\ w_0 &= \frac{1}{5-2}\frac{1}{5-3} = \frac{1}{6}. \end{aligned}$$

Damit erhalten wir

$$\begin{aligned} P(x) &= \ell(x)\left(\frac{-\frac{1}{2}}{x-3}68 + \frac{\frac{1}{3}}{x-2}16 + \frac{\frac{1}{6}}{x-5}352\right) \\ &= -68 \cdot \frac{1}{2}(x-2)(x-5) + 16 \cdot \frac{1}{3}(x-3)(x-5) + 352 \cdot \frac{1}{6}(x-2)(x-3) \end{aligned}$$

also wie zu erwarten das gleiche Polynom wie in Beispiel 7.3.

Um dieses Verfahren effizient zu implementieren, teilen wir die Berechnung in zwei Algorithmen auf, die Berechnung der baryzentrischen Koordinaten und die Auswertung des Interpolationspolynoms:

Algorithmus 7.5 (Berechnung der baryzentrischen Koordinaten)
Gegeben seien Stützstellen $x_0, \ldots, x_n$.

Für i von 0 bis n

(1) Setze $w_i := 1$

(2) Für j von 0 bis n

(a) Falls $j \neq i$ setze $w_i := w_i/(x_i - x_j)$

Algorithmus 7.6 (Auswertung des Interpolationspolynoms)
Gegeben seien Stützstellen $x_0, \ldots, x_n$, Stützwerte $f_0, \ldots, f_n$, baryzentrische Koordinaten $w_0, \ldots, w_n$ sowie die Auswertungsstelle x.

(0) Setze $l := 1$ und $s := 0$

(1) Für i von 0 bis n

  (a) Setze $y := x - x_i$

  (b) Falls $y = 0$ ist, setze $P := f_i$ und beende Algorithmus

  (c) Setze $l := l \cdot y$

  (d) Setze $s := s + w_i \cdot f_i / y$

(2) Setze $P := l \cdot s$

Zählt man die Anzahl der Operationen für Algorithmus 7.5, so sieht man, dass die Berechnung der baryzentrischen Koordinaten $\mathcal{O}(n^2)$ Operationen benötigt. Dies entspricht der Ordnung des Aufwandes der direkten Auswertung von P. Der Trick liegt aber nun darin, die w_i nur einmal vorab zu berechnen und die gespeicherten Werte in der Auswertung von P zu verwenden.
Durch Abzählen der Operationen für Algorithmus 7.6 erhält man deswegen $\mathcal{O}(n)$ Operationen. Sind also die w_i einmal berechnet, so ist die Auswertung für ein gegebenes x deutlich weniger aufwendig als die direkte Auswertung von P. Dies ist z.B. bei der graphischen Darstellung des Polynoms ein wichtiger Vorteil, da das Polynom dabei für viele verschiedene x ausgewertet werden muss.

7.2 Funktionsinterpolation

Im Spezialfall der Funktionsinterpolation kann der Fehler des interpolierenden Polynoms P, d.h. der Abstand von P zu der zu interpolierenden Funktion f, abgeschätzt werden. Hierbei bezeichnen wir mit $[a, b]$ ein Interpolationsintervall mit der Eigenschaft, dass alle Stützstellen $x_i \in [a, b]$ erfüllen und verwenden als Abstandsbegriff die Maximumsnorm

$$\|f\|_\infty := \max_{x \in [a,b]} |f(x)|.$$

Dann ergibt sich folgende Abschätzung des Fehlers:

Theorem 7.7
Sei f eine $(n+1)$ mal stetig differenzierbare Funktion und P das Interpolationspolynom zu den paarweise verschiedenen Stützstellen $x_0, \ldots, x_n$. Dann gilt:

(i) Für alle $x \in [a, b]$ gibt es ein $\xi \in [a, b]$, so dass die Gleichung

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!(x - x_0)(x - x_1) \cdots (x - x_n)}$$

gilt.

(ii) Für alle $x \in [a,b]$ gilt die Abschätzung

$$|f(x) - P(x)| \leq \|f^{(n+1)}\|_\infty \left| \frac{(x-x_0)\cdots(x-x_n)}{(n+1)!} \right|.$$

(iii) Es gilt die Abschätzung

$$\|f - P\|_\infty \leq \|f^{(n+1)}\|_\infty \frac{(b-a)^{n+1}}{(n+1)!}.$$

Wir illustrieren diese Abschätzungen an den folgenden zwei Beispielen:

Beispiel 7.8
Betrachte die Funktion $f(x) = \sin(x)$ auf dem Intervall $[0, 2\pi]$. Die Ableitungen von f sind

$$f^{(1)}(x) = \cos(x),\ f^{(2)]}(x) = -\sin(x),\ f^{(3)}(x) = -\cos(x),\ f^{(4)}(x) = \sin(x)\ldots.$$

Für alle diese Funktionen gilt $|f^{(k)}(x)| \leq 1$ für alle $x \in \mathbb{R}$. Mit äquidistanten Stützstellen $x_i = 2\pi i/n$ ergibt sich damit die Abschätzung

$$|f(x) - P(x)| \leq \max_{y\in[a,b]} |f^{(n+1)}(y)| \frac{(b-a)^{n+1}}{(n+1)!} \leq \frac{(2\pi)^{n+1}}{(n+1)!}.$$

Dieser Term konvergiert für wachsende n sehr schnell gegen 0, weswegen man schon für kleine n eine sehr gute Übereinstimmung der Funktionen erwarten kann.

Beispiel 7.9
Betrachte die sogenannte Runge Funktion $f(x) = 1/(1+x^2)$ auf dem Intervall $[-5, 5]$. Die exakten Ableitungen führen zu ziemlich komplizierten Termen, man kann aber nachrechnen, dass für gerade n die Gleichung

$$\max_{y\in[a,b]} |f^{(n)}(y)| = |f^{(n)}(0)| = n!$$

gitl, für ungerade n zudem zumindest approximativ

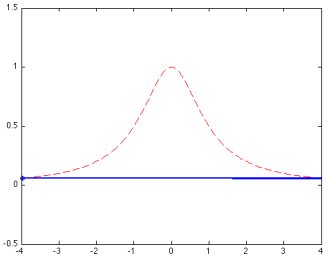
$$\max_{y\in[a,b]} |f^{(n)}(y)| \approx n!$$

Damit ergibt sich

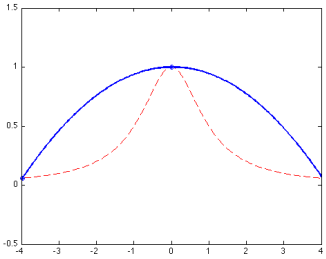
$$|f(x) - P(x)| \leq \max_{y\in[a,b]} |f^{(n+1)}(y)| \frac{(b-a)^{n+1}}{(n+1)!} \approx (n+1)! \frac{(b-a)^{n+1}}{(n+1)!} = 10^{n+1}.$$

Dieser Term wächst für große n gegen unendlich, weswegen die Abschätzung hier keine brauchbare Fehlerschranke liefert. Und tatsächlich zeigen sich bei dieser Funktion für äquidistante Stützstellen bei numerischen Tests große Problem: Insbesondere lässt sich für wachsende n keine Konvergenz erzielen, stattdessen stellt man für große n starke Oszillationen des interpolierenden Polynoms fest.

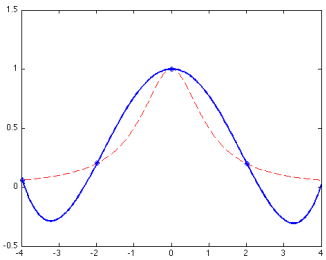
Das Fehlerverhalten der Interpolation der Runge Funktion ist auch in den folgenden Graphiken erkennbar. Beachte, dass die Oszillationen zuerst an den Rändern des Interpolationsbereichs auftreten.



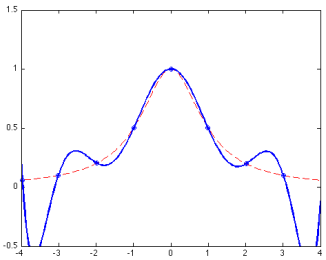
(a) $n = 2$



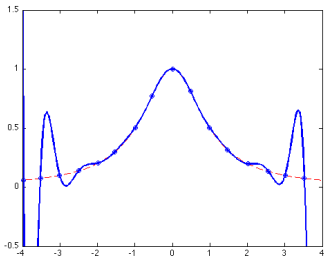
(b) $n = 3$



(c) $n = 5$



(d) $n = 9$



(e) $n = 17$

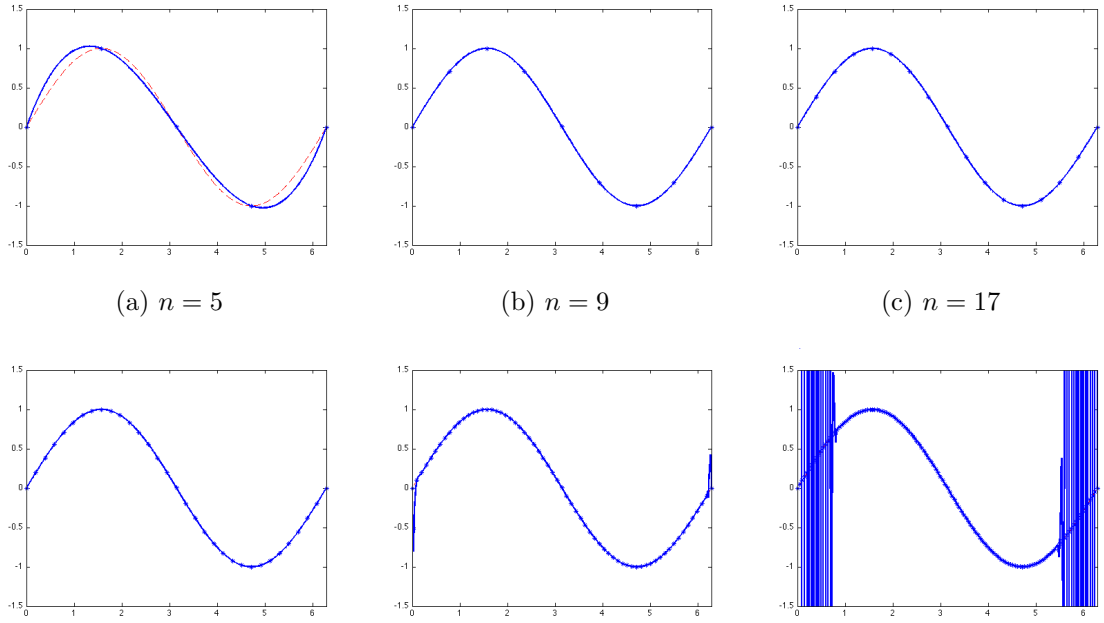
Abbildung 7.1: Interpolation der Runge Funktion mit verschiedenen Anzahlen äquidistanter Stützstellen

Wenn wir uns numerische Resultate der Interpolation der Sinus Funktion aus Beispiel 7.8 anschauen, dann würden wir nach der Fehlerabschätzung in Beispiel 7.8 erwarten, dass die Approximation für wachsendes n immer besser wird. Bis zu einem gewissen Grad stimmt dies auch, wie wir in Abbildung 7.2 sehen, kommt es aber auch hier zu Oszillationen. Diese sind nun aber nicht mehr beispielbedingt wie dies bei der Runge Funktion aus Beispiel 7.9 bzw. Abbildung 7.1 der Fall war, sondern kommen durch die sehr schlecht Kondition des Interpolationsproblems zustande.

Bemerkung 7.10
Auf eine genaue Analyse der Kondition des Interpolationsproblems verzichten wir hier. Diese ist für paarweise verschiedene Stützstellen $x_0, \ldots, x_n$ und zugehörige Lagrange Polynome L_i gegeben durch

$$\varepsilon_{abs} = \| \sum_{i=0}^{n} |L_i| \|_\infty.$$

Im Fall der Funktionsinterpolation hat man in der Fehlerabschätzungen aus Theorem 7.7 aber noch die Freiheit, die Stützstellen $x_0, \ldots, x_n$ zu wählen. Dies kann mittels sogenannter *orthogonaler Polynome* geschehen, wobei wir hier nur den Spezialfall der *Tschebyscheff Polynome* erwähnen wollen. Die Stützstellen der Tschebyscheff Polynome sind durch $x_i = \cos((2i+1)\pi/(2n+2))$ gegeben und zeichnen sich dadurch aus, dass das Produkt $(x - x_0) \cdots (x - x_n)$ in der Fehlerabschätzung (ii) aus Theorem 7.7 minimiert wird. Vergleicht man für äquidistante



(a) $n = 5$ (b) $n = 9$ (c) $n = 17$

(d) $n = 33$ (e) $n = 65$ (f) $n = 129$

Abbildung 7.2: Interpolation der Sinus Funktion mit verschiedenen Anzahlen äquidistanter Stützstellen

und Tschebyscheff Polynome die absolute Kondition des Interpolationproblems in Abhängigkeit von der Anzahl der Stützstellen, so erhält man folgende Werte:

n	ε_{abs} für äquidistante Stützstellen	ε_{abs} für Tschebyscheff Stützstellen
5	3.11	2.10
10	29.89	2.49
15	512.05	2.73
20	10986.53	2.90
60	$2.97 \cdot 10^{15}$	3.58
100	$1.76 \cdot 10^{27}$	3.90

Tabelle 7.1: Kondition ε_{abs} für verschiedene Stützstellen

7.3 Splineinterpolation

Wir haben gesehen, dass die Polynominterpolation aus Konditionsgründen problematisch ist, wenn wir viele Stützstellen gegeben haben und diese nicht — wie die Tschebyscheff Stützstellen — optimal gewählt sind. Dies kann insbesondere bei der Dateninterpolation (7.1) auftreten, wenn die Stützstellen fest vorgegeben sind und nicht frei gewählt werden können. Wir behandeln daher in diesem Abschnitt eine alternative Interpolationstechnik, die auch bei einer großen Anzahl von Stützstellen problemlos funktioniert. Wir betrachten dazu paarweise verschiedene Stützstellen und nehmen an, dass diese aufsteigend angeordnet sind, also $x_0 < x_1 < \ldots < x_n$ gilt.

Die Grundidee der *Splineinterpolation* liegt darin, die interpolierende Funktion nicht global, sondern nur auf jedem Teilintervall $[x_i, x_{i+1}]$ als Polynom zu wählen. Diese Teilpolynome sollten dabei an den Intervallgrenzen nicht beliebig, sondern möglichst glatt zusammenlaufen. Eine solche Funktion, die aus glatt zusammengefügten stückweisen Polynomen besteht, nennt man *Spline*.
Die verschiedenen Klassen von Splines unterscheiden sich nun dadurch, dass sie auf Polynomen verschiedenen Grades aufbauen. Wir werden hier exemplarisch die sogenannten *kubischen Splines* behandeln, die — wie der Name bereits andeutet — auf Polynomen dritten Grades beruhen. Splines, die auf anderen Polynomen aufbauen, werden aber ganz ähnlich behandelt.

Kubische Splines werden in Anwendungen wie z.B. der Computergrafik bevorzugt verwendet, und wir wollen als nächstes den Grund dafür erläutern. Ein Kriterium zur Wahl der Ordnung eines Splines — speziell bei grafischen Anwendungen, aber auch bei „klassischen“ Interpolationsproblemen — ist, dass die Krümmung der interpolierenden Kurve möglichst klein sein soll. Die Krümmung einer Kurve $y(x)$ in einem Punkt x ist gerade gegeben durch die zweite Ableitung $y''(x)$. Die Gesamtkrümmung für alle $x \in [x_0, x_n]$ kann nun auf verschiedene Arten gemessen werden, hier verwenden wir die L_2 Norm $\|\cdot\|_2$ für quadratisch integrierbare Funktionen, die für eine Funktion $g : [x_0, x_n] \to \mathbb{R}$ durch

$$\|g\|_2 := \left(\int_{x_0}^{x_n} g^2(x) dx \right)^{1/2}$$

gegeben ist. Die Krümmung einer zweimal stetig differenzierbaren Funktion $y : [x_0, x_n] \to \mathbb{R}$ über dem gesamten Intervall kann also mittels $\|y''\|_2$ gemessen werden. Ein Spline zeichnet sich dabei dadurch aus, dass seine Krümmung in der L_2 Norm minimal ist, d.h. $\|S''\|_2 \leq \|y''\|_2$.

Diese Eigenschaft erklärt auch den Namen Spline: Ein „Spline“ ist im Englischen eine dünne Holzlatte. Wenn man diese so verbiegt, dass sie vorgegebenen Punkten folgt (diese also „interpoliert“), so ist auch bei dieser Latte die Krümmung, die hier näherungsweise die notwendige „Biegeenergie“ beschreibt, minimal — zumindest für kleine Auslenkungen der Latte.

Um einen derartigen Spline zu berechnen müssen wir zunächst die obige umgangssprachliche Definition in mathematische Ausdrücke fassen.

Definition 7.11 (Kubischer Spline)
Seien $x_0 < x_1 < \ldots < x_n$ Stützstellen. Eine stetige Funktion $S : [x_0, x_n] \to \mathbb{R}$ heißt kubischer Spline, falls die folgenden zwei Bedingungen erfüllt sind:

(i) Auf jedem Intervall $I_k = [x_{k-1}, x_k]$ mit $k = 1, \ldots, n$ ist S gegeben durch ein Polynom S_k dritten Grades, d.h. für $x \in I_k$ gilt

$$S(x) = S_k(x) = a_k + b_k(x - x_{k-1}) + c_k(x - x_{k-1})^2 + d_k(x - x_{k-1})^3. \tag{7.6}$$

(ii) Die Ableitungen der Polynome S_k an den Stützstellen erfüllen die Bedingungen

$$S_k'(x_k) = S_{k+1}'(x_k) \qquad \text{und} \qquad S_k''(x_k) = S_{k+1}''(x_k)$$

für alle $k = 1, \ldots, n-1$.

Bedingung (i) besagt, dass S aus Polynomen zusammengesetzt ist, während Bedingung (ii) das „glatte Zusammenstoßen“ präzisiert: Die Bedingungen an die Ableitungen garantieren, dass die Splines an den „Nahtstellen“ keine „Knicke“ haben und ihre Krümmungungen stetig ineinander übergehen.

Ein solcher kubischer Spline aus Definition 7.11 löst dann das Interpolationsproblem, falls zusätzlich die Bedingung (7.1) erfüllt ist, also $S(x_i) = f_i$ für alle $i = 0, \dots, n$ gilt.

Wir müssen uns zunächst Gedanken darüber machen, ob das Problem der Splineinterpolation so wohldefiniert ist, d.h. ob ein interpolierender Spline immer existiert und ob er eindeutig ist. Zur Erinnerung: Nach den Ableitungsregeln für Polynome gilt

$$S_k'(x) = b_k + 2c_k(x - x_{k-1}) + 3d_k(x - x_{k-1})^2 \quad \text{und}$$
$$S_k''(x) = 2c_k + 6d_k(x - x_{k-1}).$$

Zur Bestimmung von S müssen wir $4n$ Werte a_i, b_i, c_i und d_i für $i = 1, \dots, n$ berechnen. Aus Definition 7.11(ii) erhalten wir hierbei $2n - 2$ lineare Gleichungen und aus (7.1) erhalten wir weitere $2n$ lineare Gleichungen. Insgesamt ergeben sich so $4n - 2$ lineare Gleichungen für $4n$ Unbekannte. Dieses lineare Gleichungssystem ist damit lösbar, allerdings gibt es unendlich viele Lösungen. Der Grund für die fehlenden Gleichungen liegt in den Randpunkten x_0 und x_n, in denen wir keine Glattheitsbedingungen fordern müssen.

Um eine eindeutige Lösung zu erhalten, müssen wir zwei weitere Gleichungen festlegen, welche sich üblicherweise aus Randbedingungen in den Punkten x_0 und x_n ergeben. Wir wollen hierbei die sogenannten *natürlichen Randbedingungen* betrachten. Hier wird gefordert, dass die zweiten Ableitungen am Rand gleich Null sein sollen, also $S''(x_0) = S''(x_n) = 0$. Dies führt auf zwei zusätzlichen Gleichungen, womit das Gleichungssystem dann eindeutig lösbar ist (tatsächlich muss man hier natürlich noch nachrechnen, dass das entstehende System eine invertierbare Matrix besitzt, was wir hier nicht explizit durchführen werden).

Bemerkung 7.12
Natürlich kann man viele andere Randbedingungen festlegen, die dann auf andere zusätzliche Gleichungen führen. Oftmals werden hierzu periodische Randbedingungen, d.h. $S'(x_0) = S'(x_n)$ und $S''(x_0) = S''(x_n)$, oder hermite'sche Randbedinungen, d.h. $S'(x_0) = f'(x_0)$ und $S'(x_n) = f'(x_n)$ (nur sinnvoll bei Funktionsinterpolation verwendet.

Um die Koeffizienten a_k, b_k, c_k und d_k für $k = 1, \dots, n$ tatsächlich numerisch zu berechnen, empfiehlt es sich zunächst die Werte

$$f_k'' := S_k''(x_k) \quad \text{und} \quad h_k := x_k - x_{k-1}$$

für $k = 0, \dots, n$ bzw. $k = 1, \dots, n$ zu definieren. Löst man dann die vier Gleichungen

$$S_k(x_{k-1}) = f_{k-1}, \quad S_k(x_k) = f_k, \quad S_k''(x_{k-1}) = f_{k-1}'', \quad S_k''(x_{k-1}) = f_k'' \tag{7.7}$$

unter Ausnutzung der Ableitungsregeln für Polynome nach a_k, b_k, c_k und d_k auf, so erhält man

$$a_k = f_{k-1} \tag{7.8}$$
$$b_k = \frac{f_k - f_{k-1}}{h_k} - \frac{h_k}{6}(f_k'' + 2f_{k-1}'') \tag{7.9}$$
$$c_k = \frac{f_{k-1}''}{2} \tag{7.10}$$
$$d_k = \frac{f_k'' - f_{k-1}''}{6h_k}. \tag{7.11}$$

Da die Werte h_k und f_k direkt aus den Daten verfügbar sind, müssen lediglich die Werte f'' berechnet werden. Da aus den natürlichen Randbedingungen sofort $f_0'' = f_n'' = 0$ folgt, brauchen nur die Werte $f_1'', \dots, f_{n-1}''$ berechnet werden.

Beachte, dass wir in (7.7) bereits die Bedingungen an S_k und S_k'' in den Stützstellen verwendet haben. Aus den noch nicht benutzten Gleichungen für die ersten Ableitungen erhält man nun die Gleichungen für die Komponenten f_k'': Aus $S_k'(x_k) = S_{k+1}'(x_k)$ erhält man

$$b_k + 2c_k(x_k - x_{k-1}) + 3d_k(x_k - x_{k-1})^2 = b_{k+1}$$

für $k = 1, \ldots, n-1$. Indem man hier die Werte f_k'' und h_k gemäß den obigen Definitionen erhält man

$$h_k f_k'' + 2(h_k + h_{k+1}) f_k'' + h_{k+1} f_{k+1}'' = 6\frac{f_{k+1} - f_k}{h_{k+1}} - 6\frac{f_k - f_{k-1}}{h_k} := \delta_k$$

für $k = 1, \ldots, n-1$. Dies liefert genau $n-1$ Gleichungen für die $n-1$ Unbekannten $f_1'', \ldots, f_{n-1}''$. In Matrixform geschrieben erhalten wir so das Gleichungssystem

$$\begin{pmatrix} 2(h_1+h_2) & h_2 & 0 & \cdots & \cdots & 0 \\ h_2 & 2(h_2+h_3) & h_3 & \ddots & \ddots & \vdots \\ 0 & h_3 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & h_{n-1} & 2(h_{n+1}+h_n) \end{pmatrix} \begin{pmatrix} f_1'' \\ f_2'' \\ \vdots \\ \vdots \\ f_{n-1}'' \end{pmatrix} = \begin{pmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \vdots \\ \delta_{n-1} \end{pmatrix}. \tag{7.12}$$

Zur Berechnung des Interpolationssplines löst man also zunächst dieses Gleichungssystem und berechnet dann gemäß der obigen Formel die Koeffizienten a_k, b_k, c_k, d_k aus den f_k''.

Algorithmus 7.13 (Spline Interpolation mit natürlichen Randbedingungen)
Gegeben seien Stützstellen $x_0, \ldots, x_n$ und Stützwerte $f_0, \ldots, f_n$.

(1) Für k von 1 bis n

Setze $h_k := x_k - x_{k-1}$

(2) Für k von 1 bis $n-1$

Setze $\delta_k := 6\frac{f_{k+1}-f_k}{h_{k+1}} - 6\frac{f_k - f_{k-1}}{h_k}$

(3) Löse das lineare Gleichungssystem (7.12) und setze $f_0'' = f_n'' = 0$

(4) Für k von 1 bis n

Setze Koeffizienten gemäß (7.8) – (7.11)

Nachdem die Koeffizienten a_k, b_k, c_k und d_k für $k = 1, \ldots n$ bestimmt sind, kann man den kubischen Spline mit natürlichen Randbedingungen mit Hilfe der Formel (7.6) einfach auswerten.

7.4 Trigonometrische Interpolation und Fourier Transformation

N.N.

Kapitel 8

Numerische Integration

Die Integration von Funktionen ist eine elementare mathematische Operation, die in vielen Formeln benötigt wird. Im Gegensatz zur Ableitung, die für praktisch alle mathematischen Funktionen explizit analytisch berechnet werden kann, gibt es viele Funktionen, deren Integrale man nicht explizit angeben kann. Verfahren zur numerischen Integration (man spricht auch von Quadratur) spielen daher eine wichtige Rolle, sowohl als eigenständige Algorithmen als auch als Basis für andere Anwendungen wie z.B. der numerischen Lösung von Differentialgleichungen. Das Problem lässt sich hierbei ganz einfach beschreiben: Für eine Funktion $f : \mathbb{R} \to \mathbb{R}$ soll das Integral

$$\int_a^b f(x)dx \tag{8.1}$$

auf einem Intervall $[a, b]$ berechnet werden.

Hier beschränken wir uns auf zwei einfache Algorithmen, die nichtsdestotrotz in vielen Anwendungen gute Ergebnisse erzielen, nämlich die Newton–Cotes und die zusammengesetzten Newton–Cotes Formeln, die auch als iterierte oder aufsummierte Newton–Cotes Formeln bezeichnet werden.

8.1 Newton–Cotes Formeln

Die Grundidee der numerischen Integration liegt darin, das Integral (8.1) durch eine Summe

$$\int_a^b f(x)dx \approx (b-a)\sum_{i=0}^{n} \alpha_i f(x_i) \tag{8.2}$$

zu approximieren. Hierbei heißen die x_i die Stützstellen und die α_i die Gewichte der Integrationsformel. Die Stützstellen x_i können hierbei beliebig vorgegeben werden, folglich benötigen wir eine Formel, mit der wir zu den x_i sinnvolle Gewichte α_i berechnen können.

Die Idee der Newton–Cotes Formeln liegt nun darin, die Funktion f zunächst durch ein Interpolationspolynom P vom Grad n (zu den Daten $(x_i, f(x_i))$, $i = 0, \ldots, n$) zu approximieren und dann das Integral über dieses Polynom zu berechnen. Wir führen diese Konstruktion nun durch:

Da wir einen expliziten Ausdruck in den $f(x_i)$ erhalten wollen, bietet sich die Darstellung von P mittels der Lagrange Polynome an, also

$$P(x) = \sum_{i=0}^{n} f(x_i)L_i(x)$$

mit

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j},$$

vgl. Abschnitt 7.1. Das Integral über P ergibt sich dann zu

$$\int_a^b P(x)dx = \int_a^b \sum_{i=0}^{n} f(x_i)L_i(x)dx = \sum_{i=0}^{n} f(x_i) \int_a^b L_i(x)dx$$

Um die Gewichte α_i in (8.2) zu berechnen, setzen wir

$$(b-a)\sum_{i=0}^{n} \alpha_i f(x_i) = \sum_{i=0}^{n} f(x_i) \int_a^b L_i(x)dx.$$

Auflösen nach α_i liefert dann

$$\alpha_i = \frac{1}{b-a} \int_a^b L_i(x)dx. \tag{8.3}$$

Diese α_i können dann explizit berechnet werden, denn die Integrale über die Lagrange Polynome L_i sind explizit lösbar. Hierbei hängen die Gewichte α_i von der Wahl der Stützstellen x_i ab, nicht aber von den Funktionswerten $f(x_i)$. Für äquidistante Stützstellen $x_i = a + \frac{i(b-a)}{n}$ sind die Gewichte aus (8.3) in Tabelle 8.1 für $n = 1, \ldots, 7$ angegeben.

n	α_0	α_1	α_2	α_3	α_4	α_5	α_6	α_7
1	$\frac{1}{2}$	$\frac{1}{2}$						
2	$\frac{1}{6}$	$\frac{4}{6}$	$\frac{1}{6}$					
3	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{8}$				
4	$\frac{7}{90}$	$\frac{32}{90}$	$\frac{12}{90}$	$\frac{32}{90}$	$\frac{7}{90}$			
5	$\frac{19}{288}$	$\frac{75}{288}$	$\frac{50}{288}$	$\frac{50}{288}$	$\frac{75}{288}$	$\frac{19}{288}$		
6	$\frac{41}{840}$	$\frac{216}{840}$	$\frac{27}{840}$	$\frac{272}{840}$	$\frac{27}{840}$	$\frac{216}{840}$	$\frac{41}{840}$	
7	$\frac{751}{17280}$	$\frac{3577}{17280}$	$\frac{1323}{17280}$	$\frac{2989}{17280}$	$\frac{2989}{17280}$	$\frac{1323}{17280}$	$\frac{3577}{17280}$	$\frac{751}{17280}$

Tabelle 8.1: Gewichte der Newton–Cotes Formeln aus (8.3) für äquidistante Stützstellen x_i

Beachte, dass sich die Gewichte immer zu 1 aufsummieren und symmetrisch in i sind, d.h. es gilt $\alpha_i = \alpha_{n-i}$. Außerdem sind die Gewichte unabhängig von den Intervallgrenzen a und b.

Algorithmus 8.1
Gegeben sei eine Funktion $f : \mathbb{R} \to \mathbb{R}$, ein Integrationintervall $[a, b]$ sowie ein $n \in \mathbb{N}$ mit zugehörigen Gewichten α_i, $i = 0, \ldots, n$.

(1) Setze $h := (b-a)/n$ und $F := 0$

(2) Für i von 0 bis n

Setze $F := F + \alpha_i f(a + ih)$

(3) Setze $F := (b-a) \cdot F$

Aus der Abschätzung des Interpolationsfehlers kann man eine Abschätzung für den Integrationsfehler

$$F_n[f] := \int_a^b f(x)dx - (b-a)\sum_{i=0}^{n} \alpha_i f(x_i)$$

ableiten. Hierbei müssen die Stützstellen x_i nicht unbedingt äquidistant liegen. Für äquidistante Stützstellen ergibt sich die Abschätzung

$$|F_n[f]| \leq c_n h^{n+2} \max_{y\in[a,b]} |f^{(n+1)}(y)| \tag{8.4}$$

wobei $f^{(n+1)}$ wieder die $(n+1)$te Ableitung der Funktion f, $h = (b-a)/n$ den Abstand zwischen den Stützstellen und c_n Konstanten mit

$$c_n = \frac{1}{(n+1)!}\int_0^n \prod_{i=0}^{n} |z - z_i|dz \qquad \text{mit} \quad z = n\frac{x-a}{b-a},\ z_i = n\frac{x_i - a}{b-a}$$

bezeichnet.

Bemerkung 8.2
Für gerades n, (n+2)-mal differenzierbares f und symmetrisch verteilte Stützstellen x_i (z.B. äquidistante Stützstellen) lässt sich die bessere Abschätzung $|F_n[f]| \leq d_n h^{n+3} \max_{y\in[a,b]} |f^{(n+2)}(y)|$ beweisen, wobei die d_n ähnlich wie c_n berechnet werden.

Die Konstanten c_n und d_n können für gegebenes n explizit berechnet werden. In Tabelle 8.2 sind die darauf basierenden Fehlerabschätzungen für $n = 1, \ldots, 7$ und äquidistante Stützstellen approximativ angegeben, wobei $M_n := \max_{y\in[a,b]} |f^{(n)}(y)|$ ist.

n	1	2	3	4	5	6	7
	$\frac{(b-a)^3 M_2}{12}$	$\frac{(b-a)^5 M_4}{2880}$	$\frac{(b-a)^5 M_4}{6480}$	$\frac{5.2(b-a)^7 M_6}{10^7}$	$\frac{2.9(b-a)^7 M_6}{10^7}$	$\frac{6.4(b-a)^9 M_8}{10^{10}}$	$\frac{3.9(b-a)^9 M_8}{10^{10}}$

Tabelle 8.2: Fehlerabschätzungen der Newton–Cotes Formeln für äquidistante Stützstellen

Beachte, dass die Formeln mit ungeradem $n = 2m+1$ jeweils nur eine leichte Verbesserung gegenüber den Formeln mit geradem $n = 2m$ liefern, dafür aber eine Funktionsauswertung mehr ausgeführt werden muss. Formeln mit geradzahligem n sind also vorzuziehen.

Vereinfacht gesagt erhöht sich also die Genauigkeit mit wachsendem n, allerdings nur dann, wenn nicht zugleich die höheren Ableitungen $|f^{(n)}|$ zunehmen. Es tauchen also die gleichen prinzipiellen Probleme wie bei den Interpolationspolynomen auf, was nicht weiter verwunderlich ist, da diese ja dem Verfahren zu Grunde liegen. Hier kommt aber noch ein weiteres Problem hinzu, nämlich kann man für $n = 8$ und $n \geq 10$ beobachten, dass einige der Gewichte α_i negative werden. Dies kann zu numerischen Problemen (z.B. Auslöschungen) führen, die man möglichst vermeiden möchte. Aus diesem Grunde ist es nicht ratsam, den Grad des zugrundeliegenden Polynoms immer weiter zu erhöhen. Stattdessen wählt man ein anderes Verfahren, das im folgenden Abschnitt beschrieben ist.

8.2 Zusammengesetzte Newton–Cotes Formeln

Der Ausweg aus den Problemen mit immer höheren Polynomgraden ist bei der Integration mittels Newton–Cotes Formeln ganz ähnlich wie bei der Interpolation — nur einfacher. Bei der Interpolation sind wir von Polynomen zu Splines, also stückweisen Polynomen übergegangen. Um dort weiterhin eine „schöne“ Approximation zu erhalten, mussten wir Bedingungen an den Nahtstellen festlegen, die eine gewisse Glattheit der approximierenden Funktion erzwingen, weswegen wir die Koeffizienten recht kompliziert über ein lineares Gleichungssystem herleiten mussten.

Bei der Integration fällt diese Prozedur weg. Wie bei den Splines verwenden wir zur Herleitung der zusammengesetzten Newton–Cotes Formeln stückweise Polynome, verzichten aber auf aufwändige Bedingungen an den Nahtstellen, da wir ja nicht an einer schönen Approximation der Funktion, sondern „nur“ an einer guten Approximation des Integrals interessiert sind. In der Praxis berechnet man die zugrundeliegenden stückweisen Polynome nicht wirklich, sondern wendet die Newton–Cotes Formeln wir folgt auf den Teilintervallen an:

Sei N die Anzahl von Teilintervallen, auf denen jeweils die Newton–Cotes Formel vom Grad n verwendet werden soll. Wir setzen

$$x_i = a + ih, \qquad i = 0, 1, \ldots, nN, \quad h = b - a$$

und zerlegen das Integral (8.1) mittels

$$\int_a^b f(x)dx = \int_{x_0}^{x_n} f(x)dx0 \int_{x_n}^{x_{2n}} f(x)dx + \ldots + \int_{x_{(N-1)n}}^{x_{Nn}} f(x)dx$$

Auf jedem Teilintervall $[x_{jn}, x_{(j+1)n}]$ wenden wir nun die Newton–Cotes Formel an, d.h. wir approximieren

$$\int_{x_{jn}}^{x_{(j+1)n}} f(x)dx \approx nh \sum_{i=0}^{n} \alpha_i f(x_{jn+i})$$

und addieren die Teilapproximationen auf, also

$$\int_a^b f(x)dx \approx nh \sum_{j=0}^{N-1} \sum_{i=0}^{n} \alpha_i f(x_{jn+i}).$$

Es ergibt sich also folgender Algorithmus:

Algorithmus 8.3
Gegeben sei eine Funktion $f : \mathbb{R} \to \mathbb{R}$, ein Integrationintervall $[a, b]$, eine Anzahl N an Unterteilungen sowie ein $n \in \mathbb{N}$ mit zugehörigen Gewichten α_i, $i = 0, \ldots, n$.

(1) Setze $h := (b - a)/(Nn)$ und $F := 0$

(2) Für i von 0 bis N

  Für j von 0 bis n

    Setze $F := F + \alpha_j f(a + (iN + j)h)$

(3) Setze $F := (b - a) \cdot F$

Der entstehende Approximationsfehler

$$F_n[f] = \int_a^b f(x)dx - nh \sum_{j=0}^{N-1} \sum_{i=0}^{n} \alpha_i f(x_{jn+i})$$

ergibt sich einfach als Summe der Fehler $F_n[f]$ auf den Teilintervallen, weswegen man auch die Abschätzung

$$\begin{aligned} |F_{N,n}[f]| &\leq \sum_{j=0}^{N-1} c_n h^{n+2} \max_{y\in[x_{jn},x_{(j+1)n}]} |f^{(n+1)}(y)| \\ &\leq N c_n h^{n+2} \max_{y\in[a,b]} |f^{(n+1)}(y)| = \frac{c_n}{n}(b-a)h^{(n+1)} \max_{y\in[a,b]} |f^{(n+1)}(y)| \end{aligned}$$

und für gerades n

$$|F_{N,n}[f]| \leq \frac{d_n}{n}(b-a)h^{n+2} \max_{y\in[a,b]} |f^{(n+2)}(y)|$$

erhält.

Im Folgenden geben wir die zusammengesetzten Newton–Cotes Formeln für $n = 1, 2, 4$ mitsamt ihren Fehlerabschätzungen an. In allen Formeln sind die Stützstellen x_i als $x_i = a + ih$, also äquidistant, gewählt.

$n = 1$, Trapez–Regel:

$$\begin{aligned} \int_a^b f(x)dx &\approx \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_i) + f(b)\right) \\ |F_{N,1}[f]| &\leq \frac{b-a}{12}h^2 \max_{y\in[a,b]} |f^{(2)}(y)|, \quad h = (b-a)/N \end{aligned}$$

$n = 2$, Simpson–Regel:

$$\begin{aligned} \int_a^b f(x)dx &\approx \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{N-1} f(x_{2i}) + 4\sum_{i=0}^{N-1} f(x_{2i+1}) + f(b)\right) \\ |F_{N,1}[f]| &\leq \frac{b-a}{180}h^4 \max_{y\in[a,b]} |f^{(4)}(y)|, \quad h = (b-a)/2N \end{aligned}$$

$n = 4$, Milne–Regel:

$$\begin{aligned} \int_a^b f(x)dx \approx \frac{2h}{45}\Bigg(&7\left(f(a) + f(b)\right) + 14\sum_{i=1}^{N-1} f(x_{4i}) \\ &+ 32\sum_{i=0}^{N-1}\left(f(x_{4i+1}) + f(x_{4i+3})\right) + 12\sum_{i=0}^{N-1} f(x_{4i+2})\Bigg) \\ |F_{N,1}[f]| \leq \frac{2(b-a)}{945}h^6 &\max_{y\in[a,b]} |f^{(6)}(y)|, \quad h = (b-a)/4N \end{aligned}$$

Zum Abschluss wollen wir an einem Beispiel die praktischen Auswirkungen der Fehlerabschätzungen illustrieren.

Beispiel 8.4
Das Integral

$$\int_0^1 e^{-x^2/2} dx$$

soll mit einer garantierten Genauigkeit von $tol = 10^{-10}$ *numerisch approximiert werden. Die Ableitungen der Funktion* $f(x) = e^{-x^2/2}$ *lassen sich leicht berechnen. Es gilt*

$$\begin{aligned} f^{(2)}(x) &= (x^2 - 1)f(x), \\ f^{(4)}(x) &= (3 - 6x^2 + x^4)f(x), \\ f^{(6)}(x) &= (-15 + 45x^2 - 15x^4 + x^6)f(x). \end{aligned}$$

Mit etwas Rechnung sieht man, dass all diese Funktionen ihr betragsmäßiges Maximum auf $[0, 1]$ *in* $y = 0$ *annehmen, woraus die Gleichungen*

$$\max_{y\in[0,1]} |f^{(2)}(y)| = 1, \quad \max_{y\in[0,1]} |f^{(4)}(y)| = 3 \quad \textit{und} \quad \max_{y\in[0,1]} |f^{(6)}(y)| = 15$$

folgen.
Löst man die oben angegebenen Fehlerabschätzungen $F_{N,n}[f] \leq tol$ *für die Trapez–, Simpson– und Milne–Regel nach* h *auf und setzt die Abschätzungen für* $\max_{y\in[0,1]} |f^{(n)}(y)|$ *ein, so erhält man die folgenden Bedingungen an* h*:*

$$h \leq \sqrt{\frac{12tol}{(b-a)|f^{(2)}(0)|}} \approx \frac{1}{28867.51} \qquad \textit{(Trapez–Regel)}$$

$$h \leq \sqrt[4]{\frac{180tol}{(b-a)|f^{(4)}(0)|}} \approx \frac{1}{113.62} \qquad \textit{(Simpson–Regel)}$$

$$h \leq \sqrt[6]{\frac{945tol}{2(b-a)|f^{(6)}(0)|}} \approx \frac{1}{26.12} \qquad \textit{(Milne–Regel)}$$

Der Bruch auf der rechten Seite gibt dabei die maximal erlaubte Größe für h *vor. Um diese zu realisieren, muss* $1/(nN) \leq h$ *gelten für die Anzahl* $nN + 1$ *der Stützstellen. Da* nN *ganzzahlig ist, braucht man also* 28869 *Stützstellen für die Trapez–Regel,* 115 *Stützstellen für die Simpson–Regel und* 29 *Stützstellen für die Milne–Regel.*

Kapitel 9

Differentialgleichungen

Differentialgleichungen sind aus der Modellierung von Phänomenen in fast allen Wissenschaften kaum mehr wegzudenken. Im naturwissenschaftlich–technischen Bereich können sie z.B. zur Beschreibung physikalischer, chemischer, mechanischer, elektronischer und biologischer Systeme verwendet werden.
Wir werden uns hier auf numerische Verfahren für gewöhnliche Differentialgleichungen beschränken und das Hauptaugenmerk auf die sogenannten Anfangswertprobleme legen. Eine gewöhnliche Differentialgleichung ist eine Gleichung, bei der die Ableitung einer Funktion nach einer eindimensionalen unabhängigen Variablen mit der Funktion selbst in Beziehung gesetzt wird. Wir machen hier die Konvention, dass die unabhängige Variable immer mit „t “ bezeichnet wird, was auf die physikalische Interpretation von t als Zeit anspielt.
Gesucht ist also eine Funktion $x(t)$ mit Werten $x(t) = (x_1(t), \ldots, x_n(t))^\top \in \mathbb{R}^n$, die die Gleichung

$$\frac{d}{dt}x(t) = f(t, x(t))$$

für eine vorgegebene Funktion $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ erfüllt. Statt $\frac{d}{dt}x(t)$ schreiben wir üblicherweise kurz $\dot{x}(t)$. Um unsere Betrachtungen hier etwas zu vereinfachen, werden wir uns auf zeitunabhängige gewöhnliche Differentialgleichungen beschränken, d.h. auf Gleichungen der Form

$$\dot{x}(t) = f(x(t)). \tag{9.1}$$

9.1 Anfangswertproblem

Bevor wir uns an die numerische Behandlung dieses Problems machen, sollten wir überlegen, unter welchen Voraussetzungen die Gleichung (9.1) überhaupt eine Lösung $x(t)$ besitzt. Wir zitieren den folgenden Satz und erinnern dazu daran, dass die Funktion $f : \mathbb{R}^n \to \mathbb{R}^n$ global Lipschitz stetig heißt, wenn eine Konstante $L > 0$ existiert, so dass die Ungleichung

$$\|f(x_1) - f(x_2)\| \leq L\|x_1 - x_2\|$$

für alle $x_1, x_2 \in \mathbb{R}^n$ gilt.

Theorem 9.1
Sei $f : \mathbb{R}^n \to \mathbb{R}^n$ global Lipschitz stetig. Dann existiert für jede Zeit $t_0 \in \mathbb{R}$ und jeden Wert $x_0 \in \mathbb{R}^n$ genau eine für alle $t \in \mathbb{R}$ definierte Lösung $x(t)$ der Differentialgleichung (9.1), die die Anfangsbedingung $x(t_0) = x_0$ erfüllt. Diese Lösung bezeichnen wir mit $x(t; t_0, x_0)$. Die Zeit $t_0 \in \mathbb{R}$ heißt hierbei Anfangszeit, der Wert $x_0 \in \mathbb{R}^n$ heißt Anfangswert.

Dieses Theorem legt die Minimalbedingung fest, die wir im Folgenden an f stellen wollen, nämlich globale Lipschitz–Stetigkeit. Falls die globale Lipschitz Stetigkeit nicht erfüllt ist (was in vielen Anwendungen der Fall ist), kann man sich damit behelfen, dass man diese Eigenschaft nur für die Menge fordert, in der die Lösung $x(t; t_0, x_0)$ ihre Werte annimmt; dies führt zur lokalen Lipschitz–Stetigkeit, die für die üblichen Anwendungen in der Regel erfüllt ist.

9.1.1 Einschrittverfahren

Die einfachste und am weitesten verbreitete Klasse von numerischen Lösungsverfahren für gewöhnliche Differentialgleichungen sind die sogenannten Einschrittverfahren.
Wir betrachten diese Verfahren hier zunächst mit konstanter Schrittweite h. Ein explizites Einschrittverfahren für eine gewöhnliche Differentialgleichung (9.1) ist dann gegeben durch die Iterationsvorschrift

$$x_{i+1} = \Phi(h, x_i), \qquad i = 0, 1, 2, \ldots$$

wobei x_0 der vorgegebene Anfangswert ist und Φ eine Abbildung ist, die von f aus (9.1) abhängt und im Computer auswertbar ist. Der Wert x_i stellt dann eine Approximation für den Wert $x(ih + t_0; t_0, x_0)$ dar. Der Name Einschrittverfahren kommt daher, dass man den Wert x_{i+1} in einem Schritt, d.h. mit einer Auswertung der Abbildung Φ aus dem vorhergehenden Wert x_i berechnet.

Um ein Einschrittverfahren zu konstruieren, genügt es also, eine geeignete Abbildung $\Phi(h, x)$ zu bestimmen, die angibt, wie man aus der Approximation x_i die Approximation x_{i+1} nach dem nächsten Zeitschritt berechnet.

Beispiel 9.2
Das einfachste Verfahren dieser Art ist das Euler–Verfahren, bei dem Φ gegeben ist durch

$$\Phi(h, x) = x + hf(x)$$

also

$$x_{i+1} = x_i + hf(x_i).$$

Ein etwas komplizierteres Verfahren ist das Heun–Verfahren, das gegeben ist durch

$$\Phi(h, x) = x + 2f(x) + f(x + hf(x)).$$

Wir werden später eine systematische Art und Weise zur Darstellung von Einschrittverfahren kennenlernen. Vorher wollen wir uns allerdings mit der Konvergenztheorie dieser Verfahren beschäftigen.

9.1.2 Konvergenztheorie

Wir wollen hier Bedingungen an die Abbildung Φ angeben, unter denen man beweisen kann, dass die aus einem Einschrittverfahren gewonnene Folge x_i, $i = 0, 1, 2, 3, \ldots$ tatsächlich eine Approximation der Lösung $x(t; t_0, x_0)$ der gewöhnlichen Differentialgleichung (9.1) darstellt. Genauer wollen wir zeigen, dass — und in welchem Sinne — die Werte x_i gegen die Werte $x(hi + t_0; t_0, x_0)$ konvergieren, falls $h \to 0$ geht. Da es üblicherweise nicht sinnvoll ist, die

Zeitschrittweite h beliebig groß zu wählen, wählen wir einen beliebigen Wert h_0 und betrachten Zeitschrittweiten $0 < h \leq h_0$.

Hierzu benötigen wir die folgenden Definitionen:

Definition 9.3
Ein Einschrittverfahren heißt konsistent, falls Konstanten C, $p > 0$ existieren, so dass die Ungleichung

$$\|\Phi(h, x_0) - x(h + t_0; t_0, x_0)\| \leq Ch^{p+1}$$

gilt für alle $t_0 \in \mathbb{R}$, alle $x_0 \in \mathbb{R}^n$, alle mindestens p–mal stetig differenzierbaren $f : \mathbb{R}^n \to \mathbb{R}^n$ und alle $0 < h \leq h_0$. Der Wert p wird dabei die Konsistenzordnung des Verfahrens genannt.

Die Konsistenz vergleicht die exakte und die numerische Lösung nach einem Zeitschritt und verlangt, dass diese durch eine Potenz der Form Ch^{p+1} beschränkt ist. Der direkte Nachweis dieser Eigenschaft ist nicht ganz leicht, da darin die im Allgemeinen unbekannten Lösungen $x(h + t_0; t_0, x_0)$ vorkommen.
Für die Konsistenz eines Verfahrens gibt es aber einen einfachen Test, der die Lösungen vermeidet: Falls die Bedingung

$$\lim_{h \to 0, h > 0} \left\| \frac{\Phi(h, x) - x}{h} - f(x) \right\| = 0 \tag{9.2}$$

gilt, so ist das Verfahren konsistent. Diese Bedingung ist leichter nachzuprüfen als Definition 9.3, da nur bekannte Funktionen vorkommen; allerdings kann man aus (9.2) keine Abschätzung der Konsistenzordnung gewinnen.

Beispiel 9.4
Das Euler–Verfahren ist offenbar konsistent, da

$$\left\| \frac{\Phi(h, x) - x}{h} - f(x) \right\| = \|f(x) - f(x)\| = 0$$

ist, also (9.2) *erfüllt ist. Das Heun–Verfahren ist ebenfalls konsistent, denn aus der Lipschitz–Stetigkeit von f folgt*

$$\begin{aligned} \left\| \frac{\Phi(x, x) - x}{h} - f(x) \right\| &= \left\| \frac{1}{2} \left(f(x) + f(x + hf(x)) \right) - f(x) \right\| \\ &= \frac{1}{2} \|f(x + hf(x)) - f(x)\| \\ &\leq \frac{1}{2} L \|hf(x)\| \to 0 \end{aligned}$$

Mit mehr Aufwand (den wir hier aus Zeitgründen nicht betreiben können) kann man mit Hilfe des Taylor–Satzes nachrechnen, dass das Euler–Verfahren Konsistenzordnung $p = 1$ hat, während das Heun–Verfahren die Konsistenzordnung $p = 2$ hat.

Konsistenz ist eine notwendige Eigenschaft für Konvergenz, denn damit die x_i für $h \to 0$ gegen $x(hi + t_0; t_0, x_0)$ konvergieren, ist es natürlich notwendig, dass insbesondere $x_1 = x_0 +$

$\Phi(h, x_0)$ gegen $x(h + t_0; t_0, x_0)$ konvergiert. Sie ist aber nicht hinreichend, denn sie gibt nur Auskunft über einen Schritt des Verfahrens, wenn die exakte und die approximative Lösung in t_0 übereinstimmen. Bereits nach dem ersten Schritt ist dies aber nicht mehr der Fall, denn x_2 wird auf Basis des fehlerhaften Wertes x_1 berechnet, x_3 aus dem fehlerhaften Wert x_2 usw. Um zu vermeiden, dass diese Fehler sich im Laufe der Berechnung aufschaukeln, muss das Einschrittverfahren die folgende Stabilitätsbedingung erfüllen:

Definition 9.5
Ein Einschrittverfahren heißt stabil, falls eine Konstante $M > 0$ existiert, so dass die Ungleichung

$$\|\Phi(h, x_1) - \Phi(h, x_2)\| \leq (1 + hM)\|x_1 - x_2\|$$

für alle x_1, $x_2 \in \mathbb{R}^n$ und all $0 \leq h \leq h_0$ gilt.

Beispiel 9.6
Für das Euler–Verfahren gilt

$$\begin{aligned}\|\Phi(h, x_1) - \Phi(h, x_2)\| &= \|x_1 + hf(x_1) - x_2 - hf(x_2)\| \\ &\leq \|x_1 - x_2\| + h\|f(x_1) - f(x_2)\| \leq (1 + hL)\|x_1 - x_2\|,\end{aligned}$$

also Stabilität mit $M = L$, wobei L die Lipschitz–Konstante von f ist.
Für das Heun–Verfahren gilt

$$\begin{aligned}\|\Phi(h, x_1) - \Phi(h, x_2)\| &= \Big\|x_1 + \frac{h}{2}\left(f(x_1) + f(x_1 + hf(x_1))\right) - \\ &\qquad x_2 + \frac{h}{2}\left(f(x_2) + f(x_2 + hf(x_2))\right)\Big\| \\ &\leq \|x_1 - x_2\| + \frac{h}{2}\|f(x_1) - f(x_2)\| \\ &\qquad + \frac{h}{2}\|f(x_1 + hf(x_1)) + f(x_2 + hf(x_2))\| \\ &\leq \|x_1 - x_2\| + \frac{hL}{2}\|x_1 - x_2\| + \frac{hL}{2}\|x_1 + hf(x_1) - x_2 + hf(x_2)\| \\ &\leq \|x_1 - x_2\| + hL\|x_1 - x_2\| + \frac{h^2L^2}{2}\|x_1 - x_2\| \\ &\leq (1 + h(L + h_0L^2/2))\|x_1 - x_2\|\end{aligned}$$

also Stabilität mit $M = L + h_0L^2/2$.

Das folgende Theorem besagt, dass aus Konsistenz und Stabilität die Konvergenz des Verfahrens folgt.

Theorem 9.7
Betrachte ein Einschrittverfahren mit Abbildung Φ, das konsistent und stabil ist. Dann gibt es für jedes $T > 0$ eine Konstante $K(T) > 0$, so dass für alle $0 < h \leq h_0$ und alle $i \in \mathbb{N}$ mit $ih \leq T$ die Abschätzung

$$\|x_i + x(ih + t_0; t_0, x_0)\| \leq K(T)h^p$$

gilt, falls f aus (9.1) p–mal stetig differenzierbar ist, wobei $p > 0$ gerade die Konsistenzordnung des Verfahrens ist.

Beachte, dass die Konstante $K(T)$ immer größer wird, je größer T wird. Üblicherweise muss daher der Zeitschritt h immer kleiner gewählt werden, je größer das Intervall wird, auf dem man die Lösung berechnen will.

9.1.3 Runge–Kutta Verfahren

Die direkte Art, in der wir das Heun–Verfahren aufgeschrieben haben, wird für kompliziertere Verfahren, in denen viele Auswertungen von f in einem Schritt durchgeführt werden, sehr unübersichtlich. Wir wollen daher ein Verfahren beschreiben, mit dem man auch sehr komplizierte Einschrittverfahren noch übersichtlich aufschreiben kann. Die Klasse der Verfahren, die man so darstellen kann, wird — nach den Entwicklern dieser Darstellung — Runge–Kutta Verfahren genannt. Um die Verfahren in ihrer üblichen Allgemeinheit zu behandeln, erlauben wir in diesem Abschnitt, dass f von t abhängt, womit dann auch Φ von t abhängen muss.
Wir betrachten zunächst das Heun–Verfahren, das für zeitabhängiges f gegeben ist durch h

$$\Phi(h,t,x) = x + \frac{h}{2}\big(f(t,x) + f(t+h, x+hf(t,x))\big).$$

Mit der Abkürzung

$$\begin{aligned} k_1 &= f(t,x) \\ k_2 &= f(t+h, x+hk_1) \end{aligned}$$

lässt sich dies Verfahren auch als

$$\Phi(h,t,x) = x + h\left(\frac{1}{2}k_1 + \frac{1}{2}k_2\right)$$

schreiben.
Indem man weitere k_i–Terme einführt, kann man weitere Auswertungen von f hinzufügen. Das sogenannte klassische Runge–Kutta Verfahren (das die Konsistenzordnung $p = 4$ besitzt) ist z.B. gegeben durch

$$\begin{aligned} k_1 &= f(t,x) \\ k_2 &= f(t+h/2, x+h/2k_1) \\ k_3 &= f(t+h/2, x+h/2k_2) \\ k_4 &= f(t+h, x+hk_3) \end{aligned}$$

und

$$\Phi(h,t,x) = x + h\left(\frac{1}{6}k_1 + \frac{2}{6}k_2 + \frac{2}{6}k_3 + \frac{1}{6}k_4\right).$$

Alle diese Verfahren lassen sich durch eine rekursive Konstruktion der folgenden Art darstellen (der Vollständigkeit halber hängt f hier von t ab)

$$\begin{aligned} k_1 &= f(t + hc_1, x) \\ k_2 &= f(t + hc_2, x + h\alpha_{21}k_1) \\ k_3 &= f(t + hc_3, x + h\alpha_{31}k_1 + h\alpha_{32}k_2) \\ &\vdots \\ k_m &= f(t + hc_m, x + h\alpha_{m1}k_1 + \ldots + h\alpha_{mm-1}k_{m-1}) \end{aligned}$$

und

$$\Phi(h,t,x) = x + h(\beta_1 k_1 + \beta_2 k_2 + \dots \beta_m k_m).$$

Ein Verfahren dieser Art wird m–stufiges explizites Runge–Kutta Verfahren bezeichnet und ist durch die Koeffizienten α_{ij} und β_i eindeutig bestimmt. Die Zahl m ist hierbei gerade die Anzahl der Auswertungen von f. Zur Angabe eines solchen Runge–Kutta Verfahrens genügt es, die Koeffizienten anzugeben, was üblicherweise in der Form eines Butcher–Tableaus gemacht wird:

$$\begin{array}{c|ccccc} c_1 & & & & & \\ c_2 & \alpha_{21} & & & & \\ \vdots & \vdots & \vdots & \ddots & & \\ c_m & \alpha_{m1} & \alpha_{m2} & \cdots & \alpha_{mm-1} & \\ \hline & \beta_1 & \beta_2 & \cdots & \beta_{m-1} & \beta_m \end{array}$$

In dieser Schreibweise sind also z.B. das Euler–, Heun– und das klassische Runge–Kutta Verfahren gegeben durch die Butcher–Tableaus (von links nach rechts)

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \qquad \begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad \begin{array}{c|ccc} 0 & & & \\ \frac{1}{2} & \frac{1}{2} & & \\ \frac{1}{2} & 0 & \frac{1}{2} & \\ 1 & 0 & 0 & 1 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array}$$

Algorithmisch kann dies wie folgt implementiert werden:

Algorithmus 9.8
Gegeben sei ein Vektorfeld $f : \mathbb{R}\times\mathbb{R}^n \to \mathbb{R}^n$, ein Anfangswert $x \in \mathbb{R}^n$, eine Schrittweite $h \in \mathbb{R}_0^+$, eine Schrittzahl $N \in \mathbb{N}$ und eine Runge–Kutta Verfahren mit m Stufen.

(1) Setze $x_0 := x$ und $i := 0$

(2) Für i von 1 bis N

(a) Für i von 1 bis m

Berechne $k_i = f(t_{i-1} + hc_i, x_{i-1} + h\alpha_{i1}k_1 + \dots + h\alpha_{ii-1}k_{i-1})$

(b) Setze $x_i := x_{i-1} + h(\beta_1 k_1 + \beta_2 k_2 + \dots \beta_m k_m)$ und $t_i := t_{i-1} + h$

Bemerkung 9.9
(i) Mit Hilfe des Taylor–Satzes kann man ein systematisches Verfahren zur Herleitung von Runge–Kutta Verfahren von im Prinzip beliebig hoher Konsistenzordnung p erhalten, das — vereinfacht gesagt — auf ein Gleichungssystem zur Berechnung der Koeffizienten im Butcher–Tableau führt. Die obigen Verfahren legen nahe, dass man mit einem m–stufigen Verfahren immer die Konsistenzordnung $p = m$ erreichen kann. Leider ist dies nicht so; die folgende Tabelle gibt die zum Erreichen einer vorgegebenen Konsistenzordnung p minimal nötige Stufenzahl m an.

Konsistenzordnung p	1	2	3	4	5	6	7	8	≥ 9
minimale Stufenzahl m	1	2	3	4	6	7	9	11	$\geq p+3$

(ii) In den hier betrachteten expliziten Runge–Kutta Verfahren sind alle Koeffizienten α_{ij} mit $i \leq j$ gleich Null, weswegen wir sie in den Butcher–Tableaus einfach weggelassen haben. Bei den sogenannten impliziten Runge–Kutta Verfahren dürfen diese Koeffizienten auch ungleich Null sein, dies führt dann auf ein nichtlineares Gleichungssystem für die k_i, das z.B. durch das Newton–Verfahren gelöst werden kann. Dies bewirkt einen großen zusätzlichen numerischen Aufwand; es gibt aber eine Klasse von Differentialgleichungen (die sogenanten steifen Differentialgleichungen), bei denen die impliziten Verfahren viel besser funktionieren so dass sich dieser zusätzliche Aufwand durchaus lohnt.

9.1.4 Schrittweitensteuerung

Bisher haben sind wir in der Diskussion von Einschrittverfahren immer davon ausgegangen, dass die Schrittweite $h > 0$ vorgegeben ist und während der gesamten Rechnung konstant bleibt. Dies wird im Allgemeinen nicht besonders effizient sein, da die Lösungen gewöhnlicher Differentialgleichungen in manchen Abschnitten recht kompliziert zu lösen sein können (wegen schneller Änderung, starker Krümmung etc.), in anderen Bereichen aber sehr einfache Struktur haben. Wählt man die Schrittweite $h > 0$ konstant, so muss man bei der Wahl vom schlimmsten Fall ausgehen und rechnet dann gezwungenermaßen auch in „einfachen“ Regionen mit sehr aufwändiger hoher Genauigkeit.
Das Konzept der Schrittweitensteuerung dient dazu, diesen Nachteil aufzuheben, und funktioniert wie folgt in drei Schritten:

1. Nach Durchführung eines Schrittes mit Schrittweite h wird aus der numerischen Lösung der vorhandene numerische Fehler geschätzt.

2. Wenn der Fehler über einer vorgegebenen Genauigkeitsschranke tol liegt, wird auf Basis des Fehlers eine kleinere Schrittweite h_{neu} für den aktuellen Schritt berechnet und der aktuelle Schritt wird wiederholt.

3. Auf Basis des geschätzten Fehlers wird eine optimale Schrittweite h für den nächsten Schritt errechnet.

Wahlweise kann der 2. Schritt wegfallen; in diesem Fall wird die Fehlerschätzung nur dazu verwendet, eine gute Schrittweite für den nächsten Schritt zu berechnen, ohne dass dabei die Größe des Fehlers tatsächlich überprüft wird. Wir besprechen nun die einzelnen Schritte im Detail.

Schritt (1): Fehlerschätzung

Die Idee der Fehlerschätzung im ersten Schritt liegt darin, ausgehend von der numerischen Lösung x_i die Approximation x_{i+1} mit zwei verschiedenen Verfahren Φ und $\widehat{\Phi}$ zu berechnen, wobei $\widehat{\Phi}$ das genauere Verfahren sein soll, also höhere Konsistenzordnung besitzen soll. Da bei der Schrittweitensteuerung nur die Schrittweiten des aktuellen und der zukünftigen Schritte gesteuert werden, sollten Fehler im Wert x_i, die vorher angefallen sind, nicht berücksichtigt werden. In der Fehlerschätzung wollen wir nur den Anteil des Fehlers messen, der der numerischen Lösung mittels Φ im aktuellen Schritt von x_i nach x_{i+1} hinzugefügt wird. Dies erreichen

wir, indem wir so tun als ob x_i die exakte Lösung wäre und dann die Norm $\|\varepsilon\|$ für

$$\varepsilon := x(t_{i+1}; t_i, x_i) - \Phi(h, t_i, x_i)$$

betrachten, wobei t_i und $t_{i+1} = t_i + h$ die zu x_i und x_{i+1} gehörigen Zeiten sind und h die (im vorherigen Schritt bestimmte, bzw. für den ersten Schritt vom Anwender vorgegebene) aktuelle Schrittweite ist. Zur Schätzung dieses (unbekannten) Fehlers verwenden wir den Fehler des genaueren Verfahrens

$$\hat{\varepsilon} := x(t_{i+1}; t_i, x_i) - \widehat{\Phi}(h, t_i, x_i)$$

sowie die Differenz der Verfahren

$$\varepsilon_d := \widehat{\Phi}(h, t_i, x_i) - \Phi(h, t_i, x_i).$$

Beachte, dass ε_d die einzige Größe ist, die wir wirklich messen können, weswegen am Ende unserer Umformungen ein Ausdruck stehen sollte, in dem nur ε_d vorkommt. Da $\widehat{\Phi}$ nach Annahme höhere Konsistenzordnung besitzt, wird

$$\theta = \frac{\|\hat{\varepsilon}\|}{\|\varepsilon\|}$$

immer kleiner, wenn h gegen Null geht. Hiermit folgt

$$\frac{\|\varepsilon - \varepsilon_d\|}{\|\varepsilon\|} = \theta$$

und damit

$$\|\varepsilon - \varepsilon_d\| = \|\varepsilon\|\theta.$$

Aus den Dreiecks–Ungleichungen

$$\|\varepsilon\| - \|\varepsilon_d\| \leq \|\varepsilon - \varepsilon_d\| \leq \|\varepsilon\| + \|\varepsilon_d\|$$

folgt

$$\frac{1}{1+\theta}\|\varepsilon_d\| \leq \|\varepsilon\| \leq \frac{1}{1-\theta}\|\varepsilon_d\|.$$

Für θ nahe Null gilt also

$$\|\varepsilon_d\| \approx \|\varepsilon\|,$$

weswegen $\|\varepsilon_d\|$ ein brauchbarer Schätzwert für den Fehler $\|\varepsilon\|$ ist.

Schritt (2): Schrittweitenberechnung für aktuellen Schritt

Wenn $\|\varepsilon_d\|$ größer als eine vorgegebene Fehlertoleranz tol ist, soll die verwendete Schrittweite h verkleinert werden und der durchgeführte Schritt nochmals mit der neuen Schrittweite h_{neu} wiederholt werden.

Aus der Konsistenzabschätzung wissen wir, dass für den Fehler die Abschätzung

$$\|\varepsilon\| \leq Ch^{p+1}$$

gilt, wobei p bekannt, C aber unbekannt ist. Im schlechtesten Fall gilt hierbei Gleichheit. Wir wollen nun aus der verwendeten Schrittweite h und der Fehlerschätzung $\|\varepsilon_d\|$ eine Schrittweite h_{neu} berechnen, so dass der erwartete Fehler $\|\varepsilon_{\text{neu}}\|$ für h_{neu} die Abschätzung

$$\|\varepsilon_{\text{neu}}\| \leq C h_{\text{neu}}^{p+1} \leq \text{tol}$$

für eine vorgegebene Fehlertoleranz tol gilt. Aus $\|\varepsilon_d\| \approx \|\varepsilon\| = Ch^{p+1}$ erhalten wir $C \approx \frac{\|\varepsilon_d\|}{h^{p+1}}$. Um die Bedingung

$$C h_{\text{neu}}^{p+1} \approx \frac{\|\varepsilon_d\|}{h^{p+1}} h_{\text{neu}}^{p+1} \leq \text{tol}$$

zu erfüllen, muss also

$$h_{\text{neu}} \leq \sqrt[p+1]{\frac{\text{tol}}{\|\varepsilon_d\|}}\, h$$

gewählt werden. Um etwaige Ungenauigkeiten auszugleichen, wird hierbei üblicherweise

$$h_{i+1} = \eta \sqrt[p+1]{\frac{\text{tol}}{\|\varepsilon_d\|}}\, h_i.$$

gesetzt, wobei η eine Konstante kleiner 1 ist, z.B. $\eta = 0.9$. Außerdem muss sichergestellt werden, dass $h_{\text{neu}} \leq h_0$ für eine vorgegebene maximale Schrittweite $h_0 > 0$ ist.

Mit dieser Wahl der Schrittweite wird (im Rahmen der Genauigkeit der Fehlerschätzung) garantiert, dass die Abschätzung

$$\|\varepsilon\| \leq \text{tol}$$

gilt. Beachte, dass $\|\varepsilon\|$ hier die Genauigkeit der Lösung $x_{i+1} = \Phi(h_{\text{neu}}, t_i, x_i)$ des weniger genauen Verfahrens Φ ist. Nachdem man zur Berechnung von $\|\varepsilon_d\|$ aber sowieso schon einen Schritt mit dem genaueren Verfahren $\widehat{\Phi}$ durchgeführt hat, bietet es sich an, $x_{i+1} = \widehat{\Phi}(h_{\text{neu}}, t_i, x_i)$ zu setzen, womit man in der Regel noch einen weiteren Genauigkeitsgewinn erzielen kann.

Schritt (3): Schrittweitenberechnung für nächsten Schritt

Der Schrittweitenvorschlag für den nächsten Schritt wird mit der gleichen Formel wie in Schritt (2) berechnet: Wenn h_i den im vorherigen Schritt verwendeten Zeitschritt bezeichnet, so wählt man

$$h_{i+1} = \eta \sqrt[p+1]{\frac{\text{tol}}{\|\varepsilon_d\|}}\, h_i.$$

Beachte, dass diese Berechnung wichtig für eine effiziente Wahl der Schrittweite ist, da in Schritt (2) nur verkleinert wird, während hier auch eine Vergrößerung möglich ist, wenn der Fehler $\|\varepsilon_d\|$ größer als tol ist.

9.1.5 Eingebettete Verfahren

Zur Berechnung der neuen Schrittweite muss — neben der Auswertung des Verfahrens höherer Konsistenzordnung $\widehat{\Phi}$ — in jedem Schritt auch das Verfahren niedrigerer Konsistenzordnung

Φ ausgewertet werden, was einen hohen Aufwand darstellt, da sich die Anzahl der nötigen Auswertungen von f (die im Allgemeinen den „teuren “ Anteil des Verfahrens ausmachen) in etwa verdoppelt.

Der Aufwand dieser Berechnung kann durch die Verwendung sogenannter eingebetteter Runge–Kutta Verfahren deutlich reduziert werden. Die Idee dabei ist, dass diejenigen k_i, die zur Berechnung von $\widehat{\Phi}$ benötigt werden, auch in der Berechnung von Φ vorkommen, so dass $\widehat{\Phi}$ ohne weitere Auswertungen von f aus Φ berechnet werden kann. Hierdurch erhöht sich zwar die zum Erreichen einer vorgegebenen Konsistenzordnung p nötige Stufenzahl zwar im Vergleich zur Tabelle in Bemerkung 9.9(i) geringfügig, was durch die effiziente Programmierung der Fehlerschätzung aber mehr als ausgeglichen werden kann. Zudem können durch geschickte Wahl der Koeffizienten die Berechnung verschiedener Werte in einem Schritt ausgeführt werden.

Diese eingebetteten Verfahren lassen sich ebenfalls in einem Butcher–Tableau darstellen, wobei wir jetzt zwei β_i–Zeilen erhalten, nämlich eine für $\widehat{\Phi}$ und eine für Φ. Typische Verfahren sind ein auf dem klassischen Runge–Kutta Verfahren basierende Verfahren mit den Konsistenzordnungen 4(3) (links, hiebei ist Φ gerade das klassische Runge–Kutta Verfahren) und das Dormand–Prince 5(4) Verfahren (rechts). Dieses Verfahren steht in MATLAB unter `ode45` zur Verfügung.

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
1	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{1}{6}$	
Φ	$\frac{1}{6}$	$\frac{2}{6}$	$\frac{1}{6}$	0
$\widehat{\Phi}$	$\frac{1}{6}$	$\frac{2}{6}$	0	$\frac{1}{6}$

0							
$\frac{1}{5}$	$\frac{1}{5}$						
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					
$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	
Φ	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	
$\widehat{\Phi}$	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$

Eine entsprechende algorithmische Umsetzung gestaltet sich wie folgt:

Algorithmus 9.10
Gegeben sei ein Vektorfeld $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, Anfangswerte $x \in \mathbb{R}^n$, $t \in \mathbb{R}$, eine Fehlertoleranz tol und der Endzeitpunkt T.

(1) Setze $x_0 := x$, $i := 0$, $t_0 := t$, $h_0 := T - t$

(2) Falls $t_i = T$ beende den Algorithmus

(3) Falls $t_i + h_i > T$ setze $h_i := T - \tau_i$

(4) Setze $t_{i+1} := t_i + h_i$ und berechne $\Phi(h_i, t_i, x_i)$ und $\widehat{\Phi}(h_i, t_i, x_i)$

(5) Berechne ε_d und $h_{i,\text{neu}}$

(6) Falls $\varepsilon_d >$ tol setze $h_i := h_{i,\text{neu}}$ und gehe zu (3)

(7) Falls $\varepsilon_d \leq$ tol setze $x_{i+1} := \widehat{\Phi}(h_i, t_i, x_i)$, $h_{i+1} := h_{i,\text{neu}}$, $i := i + 1$ und gehe zu (2)

9.2 Randwertprobleme

Bisher haben wir uns ausschließlich mit der Lösung von Anfangswertproblemen

$$\dot{x}(t) = f(t, x(t)), \qquad x(t_0) = x_0$$

beschäftigt. In diesem Abschnitt wollen wir eine weitere Problemstellung bei gewöhnlichen Differentialgleichungen betrachten, nämlich die sogenannten Randwertprobleme. Ein Randwertproblem für eine gewöhnliche Differentialgleichung (9.1) im $\mathbb{R}^n$ besteht darin, eine Lösung $x^\star(t)$ der Gleichung zu finden, die für Zeiten $t_0 < t_1$ und eine Funktion $r(x, y)$, $r : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ die Bedingung

$$r(x^\star(t_0), x^\star(t_1)) = 0$$

erfüllt.

Beispiel 9.11
Betrachte die zweidimensionale Gleichung

$$\begin{pmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{pmatrix} = \begin{pmatrix} x_2(t) \\ -kx_2(t) - \sin(x_1(t)) \end{pmatrix}$$

die die Bewegung eines Pendels beschreibt, bei dem x_1 den Winkel des Pendels und x_2 die Winkelgeschwindigkeit des Pendels beschreibt. Die Konstante $k \geq 0$ gibt die Stärke der Reibung an, der das Pendel unterliegt.
Bei einem Anfangswertproblem gibt man nun eine Zeit t_0 und eine Anfangsbedingung $x_0 = (x_1^0, x_2^0)^\top$ vor, was bedeutet, dass man Position und Geschwindigkeit des Pendels im Zeitpunkt t_0 festlegt und dann errechnet, wie sich das Pendel ausgehend von diese Anfangsbedingung in der Zukunft bewegt.
Bei einem Randwertproblem ist die Problemstellung anders: Hier gibt man sich zwei Zeitpunkte $t_0 < t_1$ vor, einen Anfangs- und einen Endzeitpunkt, und stellt zu beiden Zeitpunkten Bedingungen an die Lösung. Im Pendelmodell könnte man also zum Beispiel Winkel x_1^0 und x_1^1 vorgeben und nun eine Lösung $x^\star(t) = (x_1^\star(t), x_2^\star(t))^\top$ der Pendelgleichung berechnen wollen, für die $x_1^\star(t_0) = x_1^0$ und $x_1^\star(t_1) = x_1^1$ gilt. Gesucht ist also eine Pendelbewegung, die im Zeitpunkt t_0 den Winkel x_1^0 und im Zeitpunkt t_1 den Winkel x_1^1 annimmt. Die zugehörigen Geschwindigkeiten sind nicht festgelegt, sondern spielen hier vielmehr die Rolle freier Parameter, die während der numerischen Lösung so bestimmt werden müssen, dass die zugehörige Lösung die geforderten Bedingungen auch erfüllt.
Für dieses Beispiel ergibt sich die Funktion

$$r(x, y) = \begin{pmatrix} x_1 - x_1^0 \\ y_1 - x_1^1 \end{pmatrix}.$$

9.2.1 Lösbarkeit

Ob ein gegebenes Randwertproblem tatsächlich lösbar ist, ist im Allgemeinen sehr schwer zu überprüfen. Wir beschränken uns daher hier auf einen Existenzsatz für den speziellen Fall linearer Differentialgleichungen und beweisen im allgemeinen nichtlinearen Fall nur einen lokalen Eindeutigkeitssatz.

Aus der Theorie der Differentialgleichungen ist bekannt, dass die Lösungen linearer homogener Differentialgleichungen der Form

$$\dot{x}(t) = A(t)x(t) \tag{9.3}$$

als

$$x(t; t_0, x_0) = \Phi(t; t_0)x_0$$

geschrieben werden können, wobei die sogenannte Fundamentalmatrix $\Phi(t; t0) \in \mathbb{R}^{n\times n}$ eine Lösung des matrixwertigen Anfangswertproblems

$$\dot{\Phi}(t) = A(t)\Phi(t), \qquad \Phi(t_0) = \mathrm{Id} \tag{9.4}$$

ist. Bezeichnet man die i-te Spalte dieser Matrix mit $\Phi_i(t; t_0)$, so sieht man leicht, dass Φ_i Lösung des Anfangswertproblems

$$\dot{\Phi}_i(t) = A(t)\Phi_i(t), \qquad \Phi_i(t_0) = e_i$$

ist, bei dem e_i den i-ten Einheitsvektor bezeichnet. Auf diese Weise kann man die Spalten der Matrix $\Phi(t; t0)$ auch numerisch berechnen.

Theorem 9.12
Gegeben sei eine inhomogene lineare Differentialgleichung der Form

$$\dot{x}(t) = A(t)x(t) + b(t) \tag{9.5}$$

und eine Randbedingung der Form

$$r(x, y) = Bx + Cy + d$$

für Matrizen $A(t)$, B, $C \in \mathbb{R}^{n\times n}$ und Vektoren $b(t)$, $d \in \mathbb{R}^n$. Es sei Φ die Fundamentalmatrix der zugehörigen homogenen Gleichung (9.3) *und $x(t; t_0, x_0)$ eine Lösung von* (9.5) *mit beliebigem Anfangswert $x_0 \in \mathbb{R}^n$. Dann ist*

$$x^\star = x(t; t_0, x_0^\star)$$

genau dann eine Lösung des Randwertproblems, wenn der Anfangswert $x^\star$ eine Lösung des linearen Gleichungssystems

$$(B + C\Phi(t1, t0))(x_0^\star - x_0) = -(Bx_0 + Cx(t_1; t_0, x_0) + d) \tag{9.6}$$

ist. Insbesondere existiert also genau dann eine eindeutige Lösung des Randwertproblems, wenn die Matrix $B + C\Phi(t_1, t_0)$ vollen Rang besitzt.

Beweis. Für zwei beliebige Anfangswerte $x_0, x_0^\star \in \mathbb{R}^n$ gilt für die Differenz der zugehörigen Lösungen von (9.5)

$$\begin{aligned}\frac{d}{dt}(x(t; t_0, x_0^\star) - x(t; t_0, x_0)) &= A(t)x(t; t_0, x_0^\star) + b(t) - A(t)x(t; t_0, x_0) - b(t)\\ &= A(t)(x(t; t_0, x_0^\star) - x(t; t_0, x_0))\end{aligned}$$

und damit

$$x(t; t_0, x_0^\star) - x(t; t_0, x_0) = \Phi(t, t_0)(x_0^\star - x_0)$$

und folglich auch

$$x(t; t_0, x_0^\star) = x(t; t_0, x_0) + \Phi(t, t_0)(x_0^\star - x_0).$$

Einsetzen in die Randbedingung ergibt

$$\begin{aligned} 0 &= Bx(t_0; t_0, x_0^\star) + Cx(t_1; t_0, x_0^\star) + d \\ &= Bx_0^\star + C\left(x(t_1; t_0, x_0) + \Phi(t_1, t_0)(x_0^\star - x_0)\right) + d \\ &= \left(B + C\Phi(t_1, t_0)\right)(x_0^\star - x_0) + Bx_0 + Cx(t_1; t_0, x_0) + d \end{aligned}$$

Die Randbedingung ist also genau dann erfüllt, wenn $x_0^\star$ eine Lösung des linearen Gleichungssystems (9.6) ist. □

Beachte, dass sich das Gleichungsystem (9.6) deutlich vereinfacht, wenn wir $x_0 = 0$ wählen. Wir werden später sehen, warum es dennoch nützlich ist, den Satz für allgemeine $x_0 \in \mathbb{R}^n$ zu formulieren.

Beispiel 9.13
Wenn wir die Pendelgleichung aus Beispiel 9.11 durch die lineare Pendelgleichung

$$\begin{pmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{pmatrix} = \begin{pmatrix} x_2(t) \\ -kx_2(t) - x_1(t) \end{pmatrix}$$

ersetzen, so erhalten wir ein Problem der Form aus Theorem 9.12 mit

$$A = \begin{pmatrix} 0 & 1 \\ -1 & -k \end{pmatrix},\ b = 0; B = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix},\ C = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \text{ und } d = \begin{pmatrix} -x_1^0 \\ -x_1^1 \end{pmatrix}.$$

Für allgemeine nichtlineare Differentialgleichungen ist ein solcher Satz nicht beweisbar. Wir können aber, wenn wir annehmen, dass eine Lösung $x^\star(t)$ des Randwertproblems existiert, zumindest Bedingungen für die lokale Eindeutigkeit der Lösung angeben und beweisen. Dazu — aber auch für die numerische Lösung des Problems im nächsten Abschnitt — benötigen wir die partielle Ableitung

$$\frac{\partial}{\partial x_0} x(t; t_0, x_0)$$

der Lösung eines Anfangswertproblems. Diese Ableitung kann über die sogenannte *Variationsgleichung*

$$\dot{y}(t) = A(t)y(t), \qquad A(t) = \frac{\partial f}{\partial x}(t, x(t; t_0, x_0)) \tag{9.7}$$

berechnet werden kann. Genauer gilt, dass die Fundamentalmatrix $\Phi(t, t_0)$ (vgl. (9.4)) der Variationsgleichung (9.7) gerade die (matrixwertige) Ableitung nach dem Anfangswert ist: Es gilt

$$\frac{\partial}{\partial x_0} x(t; t_0, x_0) = \Phi(t; t_0).$$

Diesen Zusammenhang nutzen wir in dem folgenden Satz.

Theorem 9.14
Es sei $x^\star : [t_0, t_1] \to \mathbb{R}^n$ eine Lösung des Randwertproblems mit $f \in \mathcal{C}^1(\mathbb{R} \times \mathbb{R}^n, \mathbb{R}^n)$ und $r \in \mathcal{C}^1(\mathbb{R}^n \times \mathbb{R}^n, \mathbb{R}^n)$. Es sei $\Phi^\star(t, t_0)$ die Fundamentalmatrix (9.4) der Variationsgleichung (9.7) mit $x(t) = x^\star(t)$. Zudem definieren wir die $n \times n$–Matrizen

$$B^\star := \frac{\partial r}{\partial x}(x^\star(t_0), x^\star(t_1)) \quad \text{und} \quad C^\star := \frac{\partial r}{\partial y}(x^\star(t_0), x^\star(t_1)) \tag{9.8}$$

über die Ableitungen der Randwertfunktion $r(x, y)$. Dann gilt: Falls die Sensitivitätsmatrix

$$E^\star(t) := B^\star \Phi^\star(t_0, t) + C^\star \Phi^\star(t_1, t)$$

für ein $t = \tau_0 \in [t_0, t_1]$ vollen Rang besitzt, so besitzt sie für alle $t \in [t_0, t_1]$ vollen Rang und $x^\star$ ist eine lokal eindeutige Lösung des Randwertproblems.

Beweis. Wir zeigen zunächst die lokale Eindeutigkeit. Definieren wir für eine beliebige Lösung $x(t; \tau_0, x_0)$ und die Randbedingungsfunktion r die Funktion

$$F(x_0) = r(x(t_0; \tau_0, x_0), x(t_1; \tau_0, x_0)),$$

so ist eine beliebige Lösung $x(t)$ der Differentialgleichung genau dann eine Lösung des Randwertproblems, wenn

$$F(x(\tau_0)) = 0 \tag{9.9}$$

gilt. Um die lokale Eindeutigkeit der Lösung zu zeigen, müssen wir also beweisen, dass eine Umgebung U um $x^\star(\tau_0)$ existiert, so dass

$$F(x) \neq 0 \qquad \text{für alle } x \in U \setminus \{x^\star(\tau_0)\}$$

gilt. Gleichung (9.9) ist ein nichtlineares Gleichungssystem mit n Gleichungen und n Unbekannten. Nach dem Satz über inverse Funktionen gibt es genau dann eine lokal eindeutige Lösung, wenn die Jacobi-Matrix $DF(x^\star(\tau_0))$ vollen Rang besitzt. Diese ist aber für $x_0 = x^\star(\tau_0)$ gerade gegeben durch

$$\begin{aligned} DF(x_0) &= \frac{d}{dx_0} r(x(t_0; \tau_0, x_0), x(t_1; \tau_0, x_0)) \\ &= \frac{\partial r}{\partial x}(x(t_0; \tau_0, x_0), x(t_1; \tau_0, x_0)) \frac{\partial}{\partial x_0} x(t_0; \tau_0, x_0) \\ &\quad + \frac{\partial r}{\partial y}(x(t_0; \tau_0, x_0), x(t_1; \tau_0, x_0)) \frac{\partial}{\partial x_0} x(t_1; \tau_0, x_0) \\ &= B^\star \Phi^\star(t_0, \tau_0) + C^\star \Phi(t_1, \tau_0) \end{aligned}$$

und besitzt daher vollen Rang. Daraus folgt die lokale Eindeutigkeit.
Würde nun ein $\tau_1 \in [t_0, t_1]$ existieren, für dass die Sensitivitätsmatrix keinen vollen Rang besitzt, so würden nach dem Satz über implizite Funktionen Werte x_0 beliebige nahe an $x^\star(t)$ existieren, so dass $x(t; \tau_1, x_0)$ das Randwertproblem löst. Damit hätten wir Werte $x(\tau_0; \tau_1, x_0)$ gefunden, die in beliebig kleinen Umgebungen von $x^\star(\tau_0)$ liegen und (9.9) lösen, was ein Widerspruch zur lokalen Eindeutigkeit ist. □

Auch wenn die Bedingungen dieses Satzes i.A. schwer zu überprüfen sind, so liefert er doch die Begründung dafür, dass eine numerische Berechnung der Lösung des Randwertproblems möglich ist, da das Problem zumindest lokal eine eindeutige Lösung besitzt und damit wohldefiniert ist. Zudem liefert er eine wichtige Einsicht in die Struktur des Problems, die wir im folgenden Abschnitt numerisch nutzen werden.

9.2.2 Schießverfahren

Der Beweis von Satz 9.14 zeigt bereits die Richtung auf, die wir bei der numerischen Lösung des Problems einschlagen können. Das Problem, eine Lösungsfunktion zu finden, die zwei vorgegebene Punkte verbindet, wurde dort reduziert auf das Problem, einen Anfangswert $x_0 \in \mathbb{R}^n$ zu finden, der das n–dimensionale nichtlineare Gleichungssystem (9.9) löst. Die dort definierte Abbildung F vereinfacht sich für $\tau_0 = t_0$ zu

$$F(x_0) = r(x_0, x(t_1; t_0, x_0)) \tag{9.10}$$

Diese Form wollen wir im Folgenden verwenden.

Unser Ziel ist nun, das Problem zu lösen, indem wir das Nullstellenproblem (9.10) numerisch lösen. Dieses Vorgehen — also die Lösung eines Randwertproblems durch die Lösung eines durch ein Anfangswertproblem bestimmtes Gleichungssystem — wird als Schießverfahren bezeichnet. Ursprung dieses etwas martialischen Namens ist tatsächlich das Schießen im militärischen Sinne, genauer die Artillerie. Auch hier hat man eine Endbedingung gegeben (nämlich ein zu treffendes Ziel) und variiert die Anfangsbedingung (Winkel des Geschützes oder Schussstärke), um die Endbedingung zu erfüllen.

Algorithmen zur Lösung nichtlinearer Gleichungssysteme kennen wir aus der Einführung in die Numerik, nämlich die Fixpunktiteration und das Newton–Verfahren. Während erstere nur unter relativ einschränkenden Bedingungen funktioniert (die wir hier realistischerweise nicht unbedingt annehmen können), funktioniert die zweite lokal immer, benötigt aber die Information über die Ableitung von F. Hier kommt als weitere Schwierigkeit hinzu, dass die Definition von F neben der — gegebenen — Abbildung r auch die — im Allgemeinen unbekannte — Lösung $x(t_1; t_0, x_0)$ enthält. Wie können F und die Ableitung DF aber numerisch auswerten, denn da $x(t_1; t_0, x_0)$ und $\Phi(t_1, t_0)$ ja gerade die Lösung von Anfangswertproblemen sind, können wir diese mit jedem der bisher behandelten Algorithmen berechnen.

Zunächst erinnern wir an das Newton-Verfahren im $\mathbb{R}^n$: Gegeben sei eine Funktion $F : \mathbb{R}^n \to \mathbb{R}^n$, ihre Ableitung $DF : \mathbb{R}^n \to \mathbb{R}^{n\times n}$ sowie ein Startwert $x(0) \in \mathbb{R}^n$ und eine gewünschte Genauigkeit $\mathrm{tol} > 0$. Setze $i = 0$.

(1) Löse das lineare Gleichungssystem $DF(x^{(i)})\Delta x^{(i)} = F(x^{(i)})$
und berechne $x^{(i+1)} = x^{(i)} - \Delta x^{(i)}$

(2) Falls $\|\Delta x^{(i)}\| < \mathrm{tol}$, beende den Algorithmus,
ansonsten setze $i = i + 1$ und gehe zu (1).

Um dies auf unser Problem anzuwenden, müssen wir nun klären, wie wir F und DF numerisch berechnen.

Die Berechnung von F aus (9.10) stellt dabei kein größeres Problem dar: Für gegebenes $x^{(i)}$ berechnen wir numerisch die Lösung $\tilde{x} = x(t_1; t_0, x^{(i)})$ mittels eines Ein– oder Mehrschrittverfahrens und berechnen damit

$$F(x^{(i)}) \approx r(x^{(i)}, \tilde{x}).$$

Komplizierter ist die Berechnung von DF . Zunächst gilt nach der Rechnung im Beweis von Theorem 9.14 mit $\tau = t_0$

$$DF(x^{(i)}) = B + C\Phi(t_1; t_0)$$

mit Matrizen B und C gegeben durch

$$B = \frac{\partial r}{\partial x}(x^{(i)}, x(t; t_0, x^{(i)}) \quad \text{und} \quad C = \frac{\partial r}{\partial y}(x^{(i)}, x(t; t_0, x^{(i)})$$

für die Randwertfunktion $r(x, y)$. Die i–te Spalte der Matrix $\Phi(t_1; t_0)$ ist nun gerade die Lösung des Anfangswertproblems

$$\dot{y}_i(t) = \frac{\partial f}{\partial x}(t, x(t_1; t_0, x^{(i)})y_i(t), \quad y_i(t_0) = e_i,$$

wobei e_i der i–te Einheitsvektor ist.

Die numerische Berechnung von F und DF kann also wie folgt geschehen: Vorab berechnen wir (analytisch) die Ableitungen

$$\frac{\partial f}{\partial x}(t, x), \quad \frac{\partial r}{\partial x}(x, y), \quad \frac{\partial r}{\partial y}(x, y).$$

In jedem Schritt des Newton-Verfahrens approximieren wir dann numerisch die Lösung $z(t_1)$ des $n(n + 1)$–dimensionalen Anfangswertproblems

$$\dot{z}(t) = g(t, z(t)), \qquad z(t_0) = z_0$$

mit

$$z(t) = \begin{pmatrix} x(t) \\ y_1(t) \\ \vdots \\ y_n(t) \end{pmatrix}$$

und

$$g(t, z(t)) = \begin{pmatrix} f(t, x(t)) \\ \frac{\partial f}{\partial x}(t, x(t))y_1(t) \\ \vdots \\ \frac{\partial f}{\partial x}(t, x(t))y_n(t) \end{pmatrix}, \qquad z_0 = \begin{pmatrix} x^{(i)} \\ e_1 \\ \vdots \\ e_n \end{pmatrix}.$$

Mit Hilfe der numerischen Approximation

$$\tilde{z} = \begin{pmatrix} \tilde{x} \\ \tilde{y}_1 \\ \vdots \\ \tilde{y}_n \end{pmatrix}$$

berechnen wir dann die Approximationen

$$F(x^{(i)}) \approx r(x^{(i)}, \tilde{x})$$

und

$$DF(x^{(i)}) \approx \widetilde{B} + \widetilde{C}\widetilde{\Phi}(t_1; t_0)$$

mit

$$\widetilde{B} = \frac{\partial r}{\partial x}(x^{(i)}, \tilde{x}),\ \widetilde{C} = \frac{\partial r}{\partial y}(x^{(i)}, \tilde{x}) \quad \text{und} \quad \widetilde{\Phi}(t_1; t_0) = (\tilde{y}_1, \ldots, \tilde{y}_n).$$

Damit kann das Newton-Verfahren nun vollständig implementiert werden.

In Beispiel 9.11 lautet das zu lösende Differentialgleichungssystem also

$$\dot{z}(t) = \begin{pmatrix} z_2(t) \\ -kz_2(t) - \sin(z_1(t)) \\ z_4(t) \\ -kz_4(t) - \sin(z_3(t)) \\ z_6(t) \\ -kz_6(t) - \sin(z_5(t)) \end{pmatrix} \qquad \text{mit} \quad z(t_0) = z_0 = \begin{pmatrix} x_1^{(i)} \\ x_2^{(i)} \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

Interessant ist, was im Falle eines linearen Problems im Sinne von Theorem 9.12 passiert. In diesem Fall ist ein Newton–Schritt ausgehend von einem beliebigen Startwert $x^{(0)}$ gerade äquivalent zu dem linearen Gleichungssystem (9.6). Wir erhalten die (bis auf numerische Diskretisierungsfehler) exakte Lösung des Randwertproblems also nach genau einem Schritt des Newton–Verfahrens. Auf die genauen Auswirkungen der Diskretisierungsfehler im Linearen und im Nichtlinearen können wir hier aus Zeitgründen nicht genauer eingehen.

Es ergibt sich folgender Algorithmus

Algorithmus 9.15
Gegeben sei ein Vektorfeld $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, ein Randwertbedingung $r : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$, $(x_0, x(t_1; t_0, x_0)) \mapsto r\left(x_0, x(t_1; t_2, x_0)\right)$, Ableitungen $\frac{\partial f(t, x(t; t_0, x_0))}{\partial x}$, $\frac{\partial r(x_0, x(t_1; t_0, x_0))}{\partial x_0}$ und $\frac{\partial r(x_0, x(t_1; t_0, x_0))}{\partial x(t_1; t_0, x_0)}$, eine Anfangswertschätzung $x^{(0)}$, eine Anfangs– und Endzeit t_0 und t_1 sowie eine Abbruchschranke $\mathrm{tol} > 0$.

(0) Setze $i := 0$

(1) Löse das Anfangswertproblem

$$\dot{x}(t; t_0, x_0) = f(t, x(t; t_0, x_0)) \quad \text{mit} \quad x(t_0; t_0, x_0) = x^{(i)}$$

auf dem Intervall $[t_0, t_1]$ und simultan das matrixwertige Anfangswertproblem

$$\dot{x}_{x_0}(t; t_0, x_0) = \frac{\partial}{\partial t}\frac{\partial}{\partial x_0}x(t; t_0, x_0) = \frac{\partial x_{x_0}(t; t_0, x_0)}{\partial t} = \frac{\partial f(t, x(t; t_0, x_0))}{\partial x} \cdot x_{x_0}(t; t_0, x_0)$$

mit Anfangswerten $x_{x_0}(t_0; t_0, x^{(i)}) = \mathrm{Id}^{n \times n}$

(2) Falls $\|r(x^{(i)}, x(t_1; t_0, x^{(i)})\| < \mathrm{tol}$ beende Algorithmus

(3) Berechne eine Newton Iteration

$$\left[\frac{\partial r\left(x^{(i)}, x(t_1; t_0, x^{(i)})\right)}{\partial x_0} + \frac{\partial r\left(x^{(i)}, x(t_1; t_0, x^{(i)})\right)}{\partial x(t_1; t_0, x_0)}\frac{\partial x(t_1; t_0, x^{(i)})}{\partial x_0}\right]\Delta x^{(i)} = -r\left(x^{(i)}, x(t_1; t_0, x^{(i)})\right)$$

und setze $x^{(i+1)} := x^{(i)} + \Delta x^{(i)}$, $i := i + 1$ und gehe zu (1)

Anhang A

Weitere Computermathematiksysteme

A.1 Weitere Softwarepakete

- MathCad (Matlab-Clone, MathSoft, Inc./PTC = Parametric Technology GmbH) www.ptc.com/products/mathcad, verbreitet unter Ingenieuren, Berechnung und Dokumentation von Berechungs- und Konstruktionsarbeiten

- MuPad (SciFace Software GmbH & Co. KG, Paderborn/Mathworks) www.mupad.com, www.mupad.de, www.mathworks.com/products/symbolic/description4.html, www.mathworks.com/support/faq/mupad.html, Weiterentwicklung, Vertreibung als Symbolic Math Toolbox von Matlab

- Scilab (Matlab-Clone vom Scilab Consortium/Digiteo Research Network auf Initiative von INRIA (Institut national de recherche en Informatique et en automatique) in Paris, Bordeaux, Grenoble, ...) www.scilab.org, www.digiteo.fr/en, www.inria.fr/index.en.html, hochkompatibel mit Matlab, viele Toolboxen, gute Dokumentation und Support

- Octave (Matlab-Clone, GNU-Projekt) www.gnu.org/software/octave/, weitgehend kompatibel mit Matlab, bessere Wahl für Unix-Systeme

- RlabPlus/Rlab (Matlab-Clone von Marijan Kostrun bzw. Ian Searle) rlabplus.sourceforge.net, rlab.sourceforge.net, am wenigsten mit Matlab kompatibel (war auch nicht das Ziel), Entwicklung für Rlab wurde eingestellt

A.2 Bekannte Archive mit Mathematik-Libraries/Software

- Netlib Repository (AT & T Bell Labs, University of Tennessee, Knoxville und Oak Ridge National Laboratory, USA) www.netlib.org

- GAMS = Guide to Available Mathematical Software (NIST = National Institute of Standards and Technology, USA) gams.nist.gov, Weboberfläche zur Auswahl der Funktion nach Problemtyp

- MathProg/Elib am ZIB Berlin www.zib.de/Software, elib.zib.de/pub/Packages/mathprog/, elib.zib.de/pub/elib, Libraries zu verschiedensten Bereichen (Numerische Analysis, Optimierung, Informatik, ...) Beispielprojekte: MIPLIB, Porta, SCIP, SoPlex, ZIMPL

- ACM Transactions on Mathematical Software (ACM = Association for Computing Machinery) www.netlib.org/toms, liinwww.ira.uka.de/bibliography/Math/toms.html, teilweise integriert in Netlib Referenzen auf mathematische Publikationen zu Software
- Math Archives archives.math.utk.edu/software.html, Fokus stärker auf Schüler
- Numerical Recipes Software www.nr.com, bekannt durch gleichnamiges Buch „Numerical Recipes: The Art of Computing“ von William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Cambridge University Press, 1256 Seiten

A.3 Programmbibliotheken

- NAG = Numerical Algorithms Group (Oxford, UK) www.nag.com, www.nag.co.uk, 1970: Gründung von the Nottingham Algorithms Group als gemeinnützige Softwarefirma durch Brian Ford und andere, Sitz in Oxford, UK; Zusammenschluss der 4 Universitäten Nottingham, Manchester, Oxford und Birmingham, 1971: 1. Version der NAG Algol 60- und NAG Fortran-Library die meistverbreiteste kommerzielle, umfassendste Library (Fortran 77/90 und C), Unterstützung von Parallelisierung, Entwicklung eigener Compiler (erster Fortran90-Compiler) Toolboxen/Interfaces zu Maple und Matlab
- IMSL Numerical Library = International Mathematics and Statistics Library (Visual Numerics, Inc., Houston, USA) www.vni.com/products/imsl, www.visual-numerics.de, weitere bekannte kommerzielle Library (Fortran, C, C#/.NET, Java) mit vielen Algorithmen der Mathematik und Statistik
- GSL = GNU Scientific Library www.gnu.org/software/gsl, seit 2002: Entwicklung von GSL mehr als 1000 Funktionen zu Eigenwerten, Integration, Interpolation, Statistik, Minimierung, Nullstellenberechnung, Zufallszahlen, Differentialgleichungslöser, ..., enthält BLAS, CBLAS, PSPP, ...
- ACML (AMD Core Math Library) developer.amd.com/cpu/Libraries/acml, enthält BLAS, LAPACK und FFT, optimiert auf AMD-Prozessoren
- Intel MKL (Math Kernel Library) software.intel.com/en-us/intel-mkl, enthält BLAS, CBLAS, LAPACK und FFT, optimiert auf Intel-Prozessoren
- GLPK = GNU Linear Programming Kit (Fortran 77, 90/95, C, C++) www.gnu.org/software/glpk, Löser für große lineare, gemischt–ganzzahlige und vergleichbare Optimierungsprobleme (primaler/dualer Simplex, Innere-Punkte-Methoden, Branch & Bound-Methoden, ...)
- BLAS = Basic Linear Algebra Subprograms (Fortran/C) www.netlib.org/blas, www.netlib.org/sparse-blas, (Sparse-BLAS) de facto API (= Application Programming Interface) für Basisoperationen in der Linearen Algebra (Matrix mal Vektor, ...), wird oft in anderen Libraries verwendet, portable, hocheffiziente BLAS-Implementierung: ATLAS = Automatically Tuned Linear Algebra Software inkl. CBLAS math-atlas.sourceforge.net
- LAPACK = Linear Algebra PACKage (Fortran 77, 90/95, C, C++) www.netlib.org/lapack, www.netlib.org/lapack, www.netlib.org/clapack, www.netlib.org/lapack++, parallelisierte Varianten: PLAPACK = Parallel Linear Algebra Package www.cs.utexas.

edu/users/plapack, ScaLAPACK www.netlib.org/scalapack, Library für Lösung linearer Gleichungssysteme, Ausgleichsprobleme, Eigenwert- und Singulärwertbestimmung, QR-Zerlegung, Nachfolger von EISPACK und LINPACK, basiert auf BLAS

- EISPACK = A Package for Solving Matrix Eigenvalue Problems (Fortran 77) basiert auf Algol-Prozeduren, entwickelt in den 60er Jahren, zusammengestellt von J.H. Wilkinson und C. Reinsch in ihrem Band „Linear Algebra in the Handbook for Automatic Computation“ www.netlib.org/eispack, Berechnung von Eigenwerten und Eigenvektoren und singulären Werten verschiedener Grundklassen von Matrizen (komplexe hermitesche, reelle symmetrische, reelle symmetrische tridiagonale, ...), durch LAPACK überholt

- LINPACK = Library for Linear Algebra von Jack Dongarra, Jim Bunch, Cleve Moler und Pete Stewart www.netlib.org/linpack, für Supercomputer der 70er und 80er Jahre, dient auch als Benchmark für Prozessoren basiert auf BLAS, durch LAPACK überholt

- ARPACK = Arnoldi Package (Fortran 77, ARPACK++ für C++) www.caam.rice.edu/software/ARPACK, Funktionen zur Berechnung von Eigenwerten großer Matrizen

- Boost (C++) www.boost.org, Library, die C++-Standards und neue, geplante C++-Erweiterungen umsetzt, inkl. Level 3 BLAS-Library: www.boost.org/doc/libs/1_35_0/libs/numeric/ublas/doc/blas.htm

Literaturverzeichnis

[1] W. Cody, J. Coonen, D. Gay, K. Hanson, D. Hough, W. Kahan, R. Karpinski, J. Palmer, F. Ris, and D. Stevenson, *A proposed radix- and word-length-independent standard for floating-point arithmetic*, IEEE Micro (1984), 86–100.

[2] P. Deuflhard and A. Hohmann, *Numerische Mathematik. I*, second ed., de Gruyter Textbook, Walter de Gruyter & Co., Berlin, 2003 (German).

[3] D.A. Patterson, J.L. Hennessy, and D. Goldberg, *Computer architecture: A quantitative approach*, Morgan Kaufman Publishers, 1990.

[4] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, third ed., Texts in Applied Mathematics, vol. 12, Springer-Verlag, New York, 2002 (English), Translated from the German by R. Bartels, W. Gautschi and C. Witzgall.