1. Übungsblatt Numerik

Büch, Lutz (Gruppe 7)
Rieck, Bastian (Gruppe 1)

Version vom 4. Mai 2006

Aufgabe 1

1. zu zeigen: $|\Delta x| \leq \sqrt{n} \cdot \epsilon$

Beweis: Da der Anfangsvektor komponentenweise um höchtens ϵ gestört wird, gilt die folgende Ungleichung:

$$|\Delta x| = \sqrt{\sum_{i=0}^{n} \Delta x_i} \leq \sqrt{\sum_{i=0}^{n} \epsilon^2} = \sqrt{n \cdot \epsilon^2} = \sqrt{n} \cdot \epsilon$$

□

2. zu zeigen: $\lim_{m\to\infty}(\Delta x) = 0$

Behauptung: Wenn $\sigma(A) < 1$, so ist insbesondere der Spektralradius von $A < 1$. Die Iteration konvergiert daher für jeden Startwert x_0 gegen den Fixpunkt x_* des Verfahrens. Somit ist der Fehler Δx_m für $m \to \infty$ zu vernachlässigen.

Beweis: Im $\mathbb{R}^N$ gilt: Wenn eine Folge bezüglich einer bestimmten Norm konvergiert, so konvergiert sie auch bezüglich aller anderen Normen (aus Analysis I/II). Wenn aber alle Eigenwerte von $A < 1$, so kann eine Norm $|\cdot|_*$ definiert werden, bei der $\|A\|_* < 1$ gilt. Damit gilt dann für den Fehler:

$$\|\Delta x_{gesamt}\|_* = \|A^m \Delta x_0\|_* \leq \|A\|_*^m \|\Delta x_0\|_* \underbrace{\longrightarrow 0}_{m\to\infty}$$

Die Behauptung wird klar, wenn die Matrix A in diagonalisierter Form vorliegt, sodass die Eigenwerte $\lambda_1 \ldots \lambda_n$ auf der Diagonalen liegen und alle anderen Stellen der Matrix 0 sind. Dann entspricht eine Multiplikation der Matrix mit einem Vektor der Streckung dieses Vektors beziehungsweise im Falle $\sigma(A) < 1$ der Stauchung des Vektors.

□

3. Für welche Matrizen bleibt der Fehler für alle m gleich?

Jede $n \times n$ Matrix $(a_{ij}), 1 \leq i \leq n, 1 \leq j \leq n$, die in jeder Reihe höchstens ein $a_{ij} = 1$ hat, wobei alle anderen a_{ij} in einer Reihe 0 sind, erfüllt die Bedingung, denn der Vektor wird aber der ersten Iteration nicht mehr verändert.

Aufgabe 2

Da g lipschitz-stetig ist, ist die Lösung der Differentialgleichung eindeutig. Vorab eine kleine Erleichterung zur Schreibweise: Statt $\partial_x f$ schreiben wir $\dot{f}$. Analog für $\dot{h}$. Zunächst führen wir eine neue Funktion $h(x)$ ein, die folgender Bedingung genügt:

$$h(x) = \dot{f}(x), h(0) = 1$$

Für $f(x)$ gilt immer noch $f(0) = 0$. Nun kann das Polygonzugverfahren durchgeführt werden. Δx sei dabei die Maschenweite:

$$\begin{aligned} h_{i+1} = h_i + \dot{h_i} \cdot \Delta x = h_i + g(x, h(x)) \\ f_{i+1} = f_i + \dot{f_i} \cdot \Delta x = f_i + h_i \cdot \Delta x \\ x_{i+1} = x_i + \Delta x \end{aligned}$$

Der folgende Perl-artige Pseudocode löst das Problem. Dabei sind die Variablen bzw. Funktionen wie folgt belegt:

n Anzahl der Schritte, die durchgeführt werden sollen.

dx $= \Delta x$ Schrittbreite.

x $= x_0 = 0$ Anfangsbedingung.

h $= h(x_0) = \dot{f}(x_0) = 1$ Anfangsbedingung.

f $= f(x_0) = 0$ Anfangsbedingung.

g(x,h) $= g(x, \dot{f}(x))$ Teil der Differentialgleichung.

In der Realität sollten hier natürlich noch diverse Ausgabeoperationen eingefügt werden. Auch ist es sinnvoll, sich zu überlegen, wie schnell und wie stark der Fehler wächst. Wenn der relative Fehler zu groß ist, sollte das Programm überarbeitet werden oder ein anderes Verfahren gewählt werden.

```
for( i = 0; i < n; i++ )
{
        h = h + dx * g(x, h );
        f = f + dx * h;

        x = x + dx;
}

```

Aufgabe 3

Das Ergebnis der Folge nähert sich sehr schnell dem Wert 25 an:

```
7.1419
24.889
25.000
25.000
25.000
```

Dies ist auch das Produkt der Eigenwerte $1, 1, 25$ der Matrix A.

Aufgabe 4

Für die Aufgabe wurde der folgende Perl-Code verwendet:

```
#!/usr/bin/perl

for( $i = 1; $i <= 37; $i++ )
{
        print ( ( 10**37 + 10**$i ) - 10**37 );
        print "\n";
}
```

Dabei ergeben sich die folgenden Ergebnisse:

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1.18059162071741e+21,9.44473296573929e+21
1.0035028776098e+23,9.99961102747647e+23
9.99961102747647e+24,9.99996520496269e+25
1.00000006227113e+27,9.99999944211969e+27
1.00000000324155e+29,9.99999999699775e+29
1.00000000005395e+31,9.99999999994923e+31
9.99999999999645e+32,1e+34,1e+35,1e+36,1e+37
```

Sie lassen sich auf die fehlende Genauigkeit der in Perl integrierten Datentypen zurückführen: Es ist wegen der beschänkten Länge der Mantisse nur möglich, sehr genaue Ergebnisse zu erhalten oder sehr große. Nicht jedoch beides. Abhilfe schafft unter anderem die Verwendung des Datentyps **BigInt**, der jedoch nicht standardmäßig in Perl verfügbar ist, oder eine andere Programmiersprache, z.B. Haskell. Der Vollständigkeit halber sei hier der Haskell-Code und die entsprechende Ausgabe beigefügt:

```
module Numerik where

main = map num04 [1..37]

power :: Integer -> Integer -> Integer
power x 0 = 1
power x 1 = x
power x n = x * power x (n - 1)
```

```
num04 :: Integer -> Integer
num04 n = ( power 10 37 + power 10 n  ) - power 10 37
```

Hier ergibt sich - da Haskell mit beliebig langen Zahlen rechnen kann - die folgende Ausgabe:

```
Numerik> main
[10,100,1000,10000,100000,1000000,10000000,100000000,1000000000,10000000
000,100000000000,1000000000000,10000000000000,100000000000000,1000000000
000000,10000000000000000,100000000000000000,1000000000000000000,10000000
000000000000,100000000000000000000,1000000000000000000000,10000000000000
000000000,100000000000000000000000,1000000000000000000000000,10000000000
000000000000000,100000000000000000000000000,1000000000000000000000000000
,10000000000000000000000000000,100000000000000000000000000000,1000000000
000000000000000000000,10000000000000000000000000000000,10000000000000000
0000000000000000,1000000000000000000000000000000000,10000000000000000000
000000000000000,100000000000000000000000000000000000,1000000000000000000
000000000000000000,10000000000000000000000000000000000000]
```