Ausarbeitung Übung 3

Studienarbeit von Dominik Schiller, Constanze Kramer, Simon Arnold & Tobias Lingenberg
Datum: 9. Dezember 2020

Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Ausarbeitung Übung 3

Studienarbeit von Dominik Schiller, Constanze Kramer, Simon Arnold & Tobias Lingenberg

Datum: 9. Dezember 2020

Darmstadt

Inhaltsverzeichnis

1 Bearbeitung der Übungsaufgaben

1.1 Differenzenquotient

Um die erste Ableitung $\frac{\mathrm{d}f}{\mathrm{d}x}(x)$ einer hinreichend glatten skalaren Funktion $f : \mathbb{R} \to \mathbb{R}$ an der Stelle x numerisch zu approximieren kann man den rechtsseitigen (1) und den zentralen (2) Differenzenquotienten

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \tag{1}$$

$$\frac{\mathrm{d}f}{\mathrm{d}x}(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2) \tag{2}$$

nutzen wobei die Schrittweite $h \neq 0$ sehr klein zu wählen ist. Mit Hilfe der Taylorentwicklung inklusive Restgliedabschätzung wird im Folgenden die Konvergenzordnung der obigen Differenzenquotienten ermittelt. An der Stelle $f(x+h)$ lautet das allgemeine Taylorpolynom zweiten Grades

$$T_2 f(x+h) = f(x) + f'(x)h + \frac{f''(\xi)}{2!}h^2$$

mit der Fehlerabschätzung $f''(\xi)$ an einer unbekannten Stelle ξ mit $x < \xi < (x+h)$. Diese Gleichung umgestellt nach der gesuchten ersten Ableitung $f'(x)$ ergibt

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{f''(\xi)}{2}h \tag{3}$$

und mit der oberen Abschätzung für den Fehler folgt

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h). \tag{4}$$

Hier wird jetzt ersichtlich, dass die Korrektur des rechtsseitigen Differenzenquotienten linear abhängig zur Schrittweite h ist und somit eine Konvergenz 1. Ordnung hat. Konvergenz 1. Ordnung heißt in diesem Fall, dass durch eine Halbierung der Schrittweite h auch die Fehlerabweichung halbiert wird.
Aus den Taylorpolynomen dritten Grades an den Stellen $f(x+h)$, siehe (5) und $f(x-h)$, siehe (6)

$$T_3 f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2!}h^2 + \frac{f'''(\xi_r)}{3!}h^3 \tag{5}$$

$$T_3 f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2!}h^2 - \frac{f'''(\xi_l)}{3!}h^3 \tag{6}$$

mit den Unbekannten ξ_r und ξ_l, $(x-h) < \xi_l < x < \xi_r < (x+h)$, folgt nach den selben Schritten wie zuvor bei (3) und (4)

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2).$$

Der zentrale Differenzenquotient hat also eine Konvergenz 2. Ordnung, da die Korrektur quadratisch von der Schrittweite h abhängt. Konvergenz zweiter Ordnung bedeutet, dass bei einer Halbierung der Schrittweite nur noch ein Viertel des Fehlers gemacht wird.
Um einen Differenzenquotient 4.Ordnung für die erste Ableitung zu konstruieren reichen nicht mehr nur zwei Punkte zum Approximieren aus, stattdessen benötigt man jetzt 4 verschiedene Punkte $f(x-h)$, $f(x-\frac{h}{2})$, $f(x+\frac{h}{2})$ und $f(x+h)$ und kombiniert zwei zentrale Differenzenquotienten. Mit der selben Vorgehensweise wie zuvor lässt sich dann nachweisen, dass der Differenzenquotient

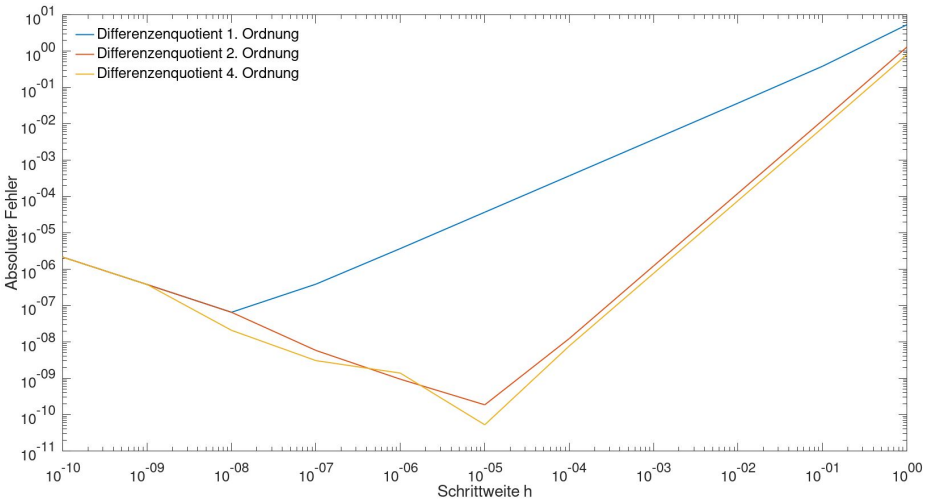
$$\frac{\mathrm{d}f}{\mathrm{d}x}(x) = \frac{f(x+h) - f(x-h) + 2f(x+\frac{h}{2}) - 2f(x-\frac{h}{2})}{4h}$$

von Konvergenz 4. Ordnung ist.

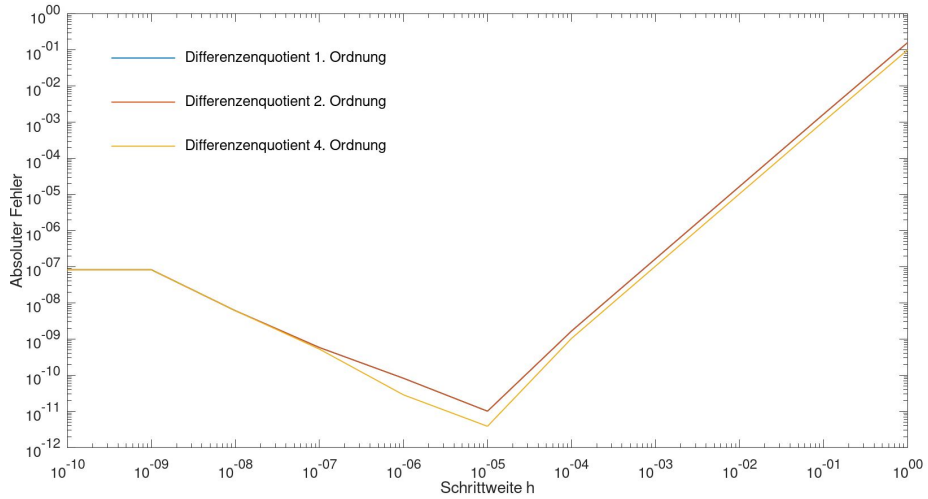
Um die Unterschiede der verschiedenen Differenzenquotienten graphisch darzustellen, bietet es sich an diese für verschieden große Schrittweiten h sowie für verschiedene Funktionen zu testen. Getestet wird mit elf verschiedenen Schrittweiten $h \in [10^{-10}, 1]$ und den Funktion $f(x) = \exp(x+1)$ (siehe Abbildung 1.1a) an der Stelle $x = 1$, $g(x) = \cos(x)$ (siehe Abbildung 1.1b) an der Stelle $x = \frac{\pi}{2}$ und der in Abbildung 1.1c dargestellte absolute Fehler der Funktion

$$h(x) = \begin{cases} 6x^2 + 4 & x > 1 \\ 4x^3 + 6 & x \leq 1 \end{cases} \tag{7}$$

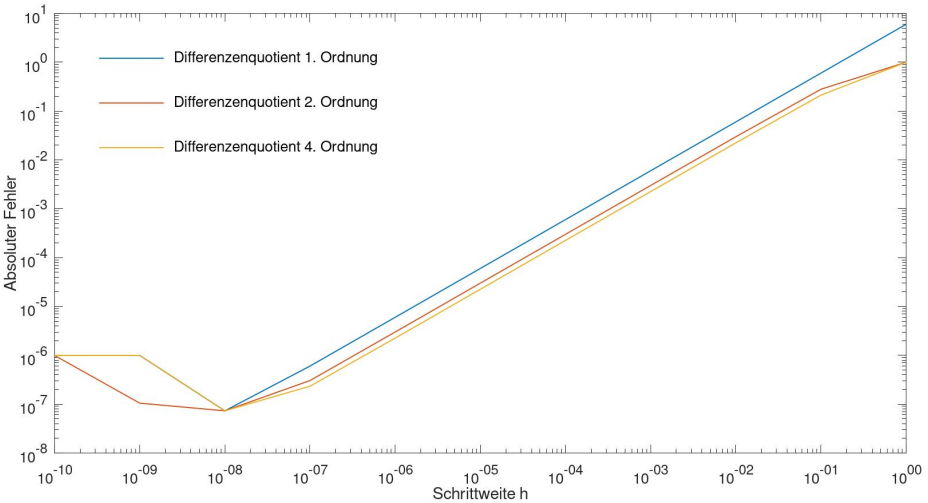
an der Stelle $x = 1$. Die im Anhang zu findende Octave-Routine `Aufgabe4_1` ermittelt den absoluten Fehler zwischen der tatsächlichen, analytisch bestimmten Ableitung der Funktionen und der numerischen Approximation mit den drei Differenzenquotienten. Zur Darstellung wird ein doppelt logarithmischer Plot verwendet, dadurch können sowohl die kleineren als auch die größeren Werte in einem Schaubild dargestellt werden.
Auffällig ist, dass bei jeder der drei Testfunktionen die kleinste Schrittweite nicht die beste Annäherung liefert, sondern es einen Tiefpunkt bei etwa $h = 10^{-5}$ bzw. $h = 10^{-8}$ gibt. Das bedeutet, dass ab diesem Punkt durch weiteres verkleinern der Schrittweite keine höhere Genauigkeit der Approximation erzielt werden kann. Außerdem erkennt man dass bei allen drei Verfahren die Größe der Fehler weitgehend ähnlich ist, mit Ausnahme der Approximation der Exponential-Funktion(1.1a), bei der der rechtsseitige Differenzenquotient einen deutlich größeren Fehler erzeugt. Für Schrittweiten größer als das Optimale h zeigen alle Graphen



(a) Fehler bei der Approximation von $f(x) = \exp(x + 1)$ an der Stelle $x = 1$



(b) Fehler bei der Approximation von $g(x) = \cos(x)$ an der Stelle $x = \frac{\pi}{2}$



(c) Fehler bei der Approximation von (7) an der Stelle $x = 1$

Abbildung 1.1: Fehler der numerischen Approximationen der ersten Ableitung im doppelt logarithmischen Koordinatensystem

lineares Verhalten. Für den Wert der Steigung ist die Konvergenzordnung ausschlaggebend, wie im Folgenden noch gezeigt wird.

Um in einem doppelt logarithmischem Koordinatensystem die Steigung einer Geraden zu berechnen (siehe Abbildung 1.2), reicht es nicht wie bei einem linearen Koordinatensystem die Werte Δx und Δy des allgemeinen Steigungsdreiecks durch die Differenz der auf den Achsen angegebenen Werte zu ermitteln.
Stattdessen wird auf die Werte, die man an den jeweiligen Stellen auf den Achsen ablesen kann, der Logarithmus zur selben Basis angewandt, der zur Erzeugung der logarithmischen Skala verwendet wurde. In diesem Beispiel zur Basis 10. Der Quotient zur Ermittlung der Steigung lautet dann

$$\frac{\Delta y}{\Delta x} = \frac{\log(C(10^{\Delta r+r})^p) - \log(C(10^r)^p)}{\log(10^{\Delta r+r}) - \log(10^r)}$$

und unter Anwendung der Logarithmus Regeln $\log(u \cdot v) = \log u + \log v$ und $\log(10^w) = w$ folgt

$$\frac{\Delta y}{\Delta x} = \frac{(p(\Delta r + r) - pr)}{r + \Delta r - r} = p.$$

Die Steigung der Fehlerabschätzung $\epsilon(h)$ ist somit genau gleich mit der Ordnung p des jeweiligen numerischen Verfahrens zur Approximation.

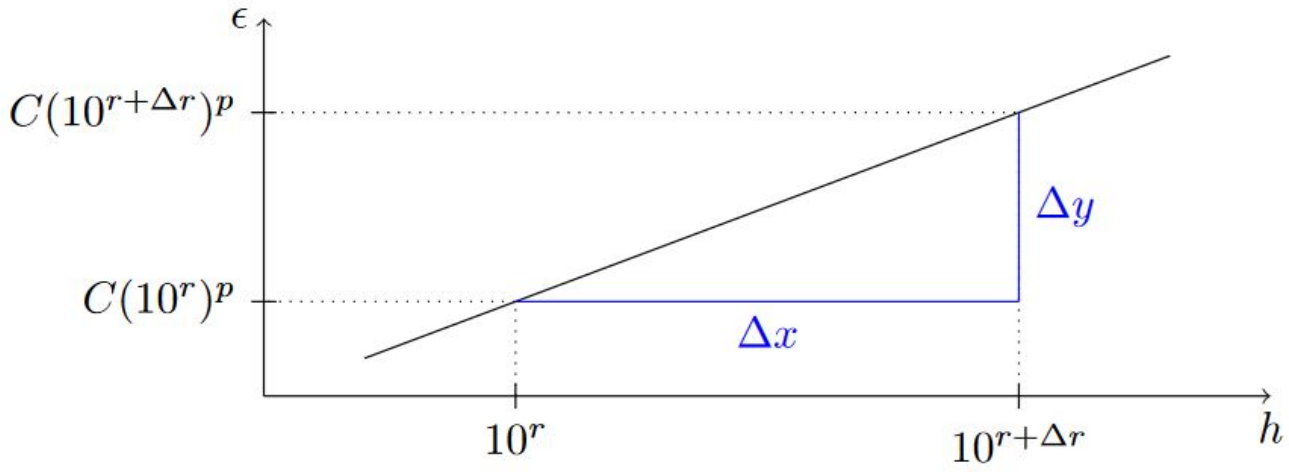


Abbildung 1.2: Allgemeines Steigungsdreieck in einem doppelt Logarithmischen Koordinatensystem

Hat das logarithmische Koordinatensystem nicht die Basis 10, sondern eine beliebige andere, ist die Steigung für $\epsilon(h)$ weiterhin durch die Ordnung p gegeben. Zur Berechnung des allgemeinen Steigungsdreiecks wird, wie bereits beschrieben, der Logarithmus zur selben beliebigen Basis auf alle Werte angewendet, somit ist die Basis des Logarithmus unerheblich für die Steigung.

1.2 Implizites Euler Verfahren

Zur numerischen Berechnung einfacher Differenzialgleichungen kann das implizite Euler Verfahren

$$\underbrace{\left(\frac{1}{h}\mathbf{M}+\mathbf{K}\right)}_{:=\mathbf{A}}\mathbf{x_i}=\underbrace{\frac{1}{h}\mathbf{M}\mathbf{x_{i-1}}+\mathbf{r}}_{:=\mathbf{b_{i-1}}} \tag{8}$$

verwendet werden. Hierzu wurde die Methode `odebwe_simple` (siehe Listing:2.1) in Octave implementiert. Zurückgegeben wird eine Lösungsmatrix $\mathbf{x}$, in der als Spaltenvektoren die verschiedenen Lösungen in Abhängigkeit der Zeit t enthalten sind.

Zum Test der Methode soll nun das Dahlquist-Problem

$$x'(t)=-kx(t) \tag{9}$$

für $k=1/5$ und $x(t_0)=2$ berechnet werden. Die analytisch exakte Lösung ist in diesem Fall durch die Funktion $x(t)=2e^{-kt}$ beschrieben. Mithilfe des Skripts Listing 2.2 wurde die analytische Lösung mit der numerischen verglichen (siehe Abbildung 1.3):

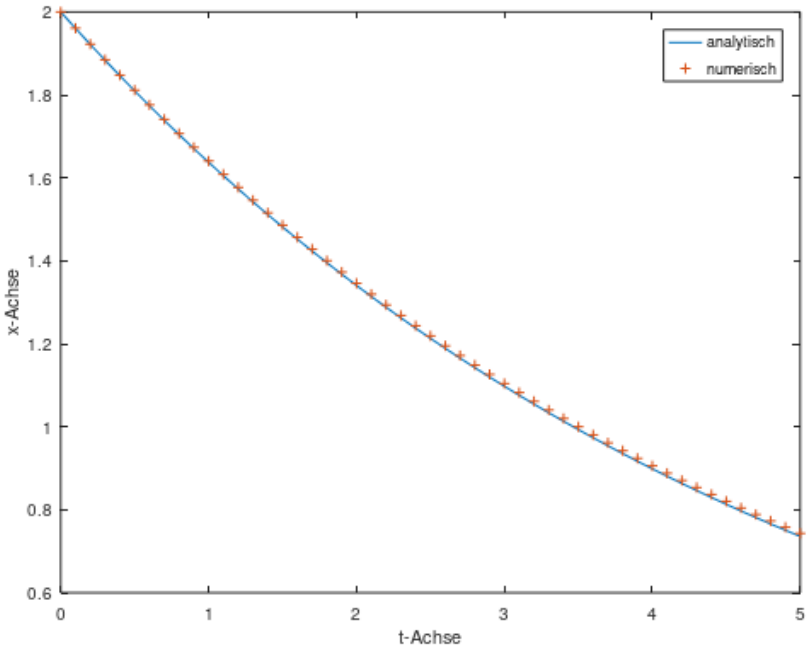


Abbildung 1.3: Numerische und analytische Lösung der Gleichung 9

Die Methode `odebwe_simple` scheint korrekt zu funktionieren.

Zuletzt soll das Koaxial Kabel aus Aufgabe 3.1 (Abbildung:1.4) des letzten Übungsblatts erneut berechnet werden.

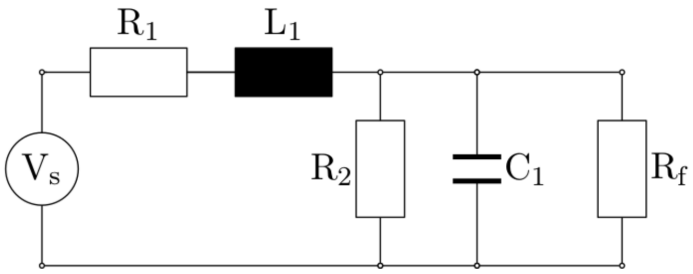


Abbildung 1.4: Ersatzschaltbild eines Koaxialkabels

Zum Lösen der Differenzialgleichung wird im Skript (Listing:2.3) die zuvor implementierte Methode `odebwe_simple` verwendet. Durch Plotten der dritten Zeile der Lösungsmatrix x erhält man den zeitlichen Verlauf der Spannung am Lastwiderstand R_f. Das sich ergebene Bild (Abbildung:1.5) gleicht den Ergebnissen des letzten Übungsblatts.

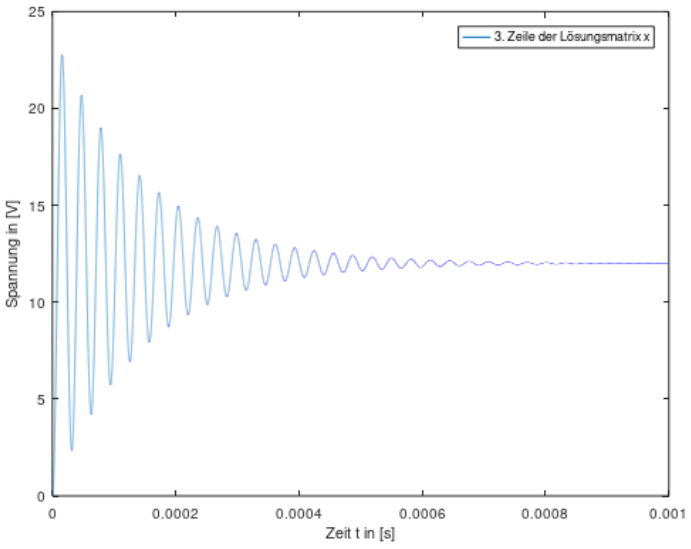
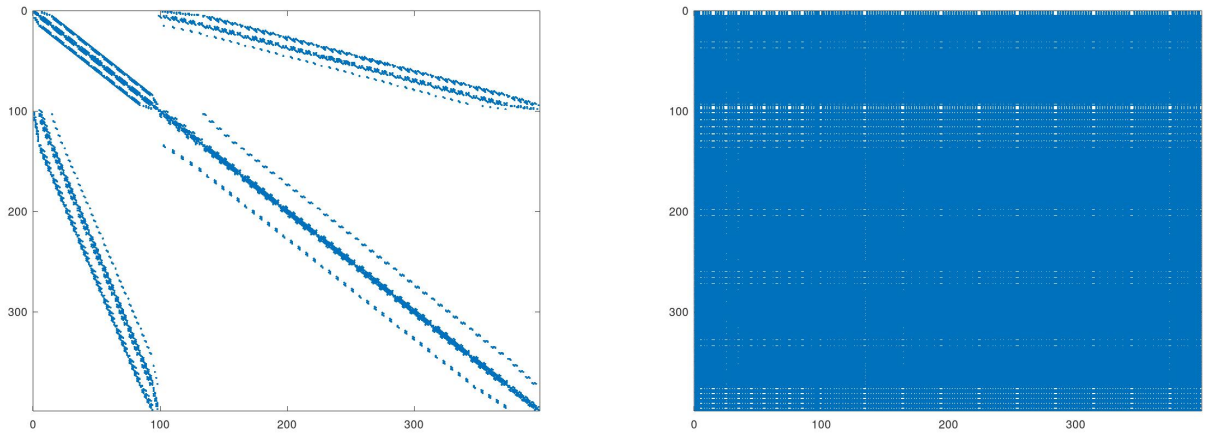


Abbildung 1.5: Spannung an Lastwiderstand R_f (siehe Abbildung:1.4)

1.3 Numerische Lösung linearer Gleichungssysteme

Bei numerischen Berechnungen spielen lineare Gleichungssysteme der Form $\mathbf{Ax}=\mathbf{y}$ oft eine überaus wichtige Rolle. Daher ist es sinnvoll und von Nöten effiziente Lösungsverfahren für diese Gleichungssysteme zu entwickeln und anzuwenden. In diesem Abschnitt werden verschiedene Verfahren in Octave implementiert und an beispielhaften Gleichungssystemen getestet und gegeneinander verglichen.

Zunächst wird eine vorgegebene quadratische Matrix betrachtet. Mit Hilfe des Befehls `spy` lassen sich Matrizen in Octave graphisch darstellen. In Abb. 1.6a ist diese Grafik zu sehen, die blauen Punkte in dem Plot sind Einträge, die ungleich null sind. Zählt man diese mit dem Befehl `nnz` findet man heraus, dass 3.678 der 158.404 Einträge ungleich null sind. Weiterhin lässt sich aus der Abbildung und der Darstellung der Werte in Octave herausfinden, dass es sich um eine symmetrische Matrix handelt, das heißt die Einträge der Matrix sind spiegelsymmetrisch bezüglich der Hauptdiagonalen.



(a) Graphische Darstellung der mit Octave eingelese- (b) Berechnete Inverse der eingelesenen Matrix
ne Beispielmatrix

Abbildung 1.6: Graphische Darstellung von Matrizen, die blauen Punkte stellen Einträge dar, die ungleich null sind

Nun soll ein erstes Lösungsverfahren in Octave implementiert werden. Das GAUSS 'SCHE-Eliminationsverfahren wurde in Listing 1.3 implementiert. Die Routine `gaussElim` erhält einen stehenden Vektor $\mathbf{b}$ mit Länge n und eine quadratische Matrix $\mathbf{A}$ der Größe $n \times n$. Die Rückgabe ist ein Vektor $\mathbf{x}$ mit Länge n, der die berechneten Lösungen $x_1, \ldots, x_n$ des linearen Gleichungssystems enthält.

Zunächst wird `n`, also die Größe der Matrix bestimmt, daraufhin wird der Ergebnisvektor $\mathbf{x}$ mit entsprechender Länge erzeugt. In Zeile 4 und 5 werden zwei Laufvariablen initialisiert, `j` läuft hierbei über die Spalten, `i` über die Zeilen. Das Ziel des Algorithmus ist es schrittweise aus der übergebenen Matrix eine untere Dreiecksmatrix zu erstellen. Hierzu werden sogenannte Eliminationsfaktoren bestimmt, siehe Listing 1.3 Zeile 6. Mit Hilfe einer weiteren Laufvariable `k`, die über die Spalten von `j` bis `n` läuft. In der Schleife, Zeile 7 bis 9, werden Zeilenadditionen durchgeführt, um so Spalte für Spalte Nulleinträge zu erzeugen. Auch der Vektor $\mathbf{b}$ wird

entsprechend angepasst.
In den Zeilen 14 bis 17 wird das Gleichungssystem final gelöst, die einzelnen $x_1, \ldots, x_n$ werden von unten nach oben mit Hilfe der Matrix und dem Vektor **b** berechnet.
Durch die drei ineinander geschachtelten For-Schleifen in den Zeilen 4 bis 12 erhalten wir eine kubische Anzahl an Rechenschritten, $n^3 - n^2$, Zeile 14 bis 17 führen zu einem zusätzlichen Rechenaufwand von $n^2 - n$, daraus ergibt sich die Abschätzung $\mathcal{O}(n^3)$.

```
1  function [x] = gaussElim (A,b)
2    n = size(A)
3    x = zeros(n,1)
4    for j=1:n-1
5      for i = j+1:n
6        l = A(i,j) / A(j,j);
7        for k = j:n
8          A(i,k) = A(i,k)  - l * A(j,k);
9        endfor
10     b(i) = b(i) - l*b(j);
11     endfor
12   endfor
13
14   for i = n:-1:1
15     f = @(k) A(i,k) * x(k);
16     x(i)= (1/A(i,i))*(b(i)-sum(f([i+1:n])));
17   endfor
18
19 endfunction
```

Lineare Gleichungssysteme lassen sich auch mit Hilfe der Inversen einer Matrix bestimmen. In Octave geht dies mit dem Befehl `inv(A)`. Wie aus Abbildung 1.6b, verglichen mit Abbildung 1.6a, hervorgeht, hat die Invertierte Matrix deutlich mehr Einträge, die ungleich null sind. Die Berechnung der Inversen benötigt sehr viel Arbeitsspeicher und es werden deutlich mehr Rechenoperationen ausgeführt, um die Lösungen zu berechnen.

Eine weitere Methode mit der lineare Gleichungssysteme effizient berechnet werden können ist die LUPQ-Zerlegung. Bei der LUPQ-Zerlegung wird die Matrix **A** mit Hilfe des bereits vorgestellten Gauss'schen-Eliminationsverfahren in eine obere und eine untere Dreiecksmatrizen aufgeteilt, es gilt $\mathbf{A} = \mathbf{LU}$. Mit Hilfe von Vorwärts-/Rückwärtseinsetzen lässt sich nun das Gleichungssystem lösen. Zum Lösen wird zunächst vorwärts

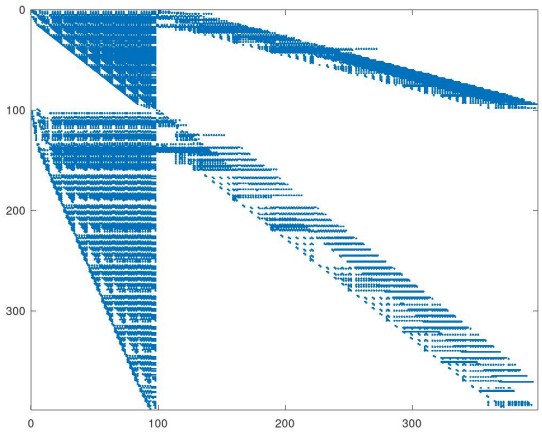
$$\mathbf{Ly} = \mathbf{b}$$

eingesetzt und anschließend rückwärts

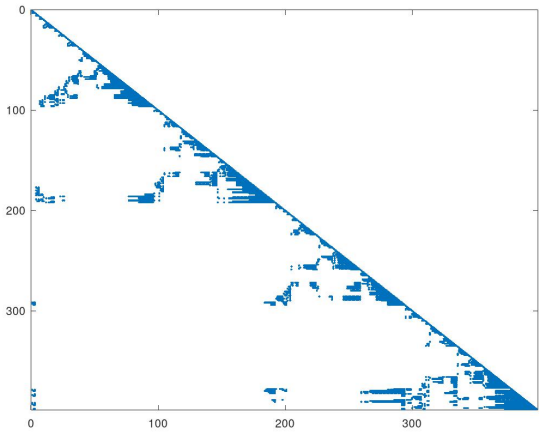
$$\mathbf{Ux} = \mathbf{y}$$

berechnet. Zum berechnen unterschiedlicher Seiten **b** muss die Matrix nicht neu zerlegt, sondern nur das Vorwärts-/Rückwärtseinsetzen betrachtet werden. Auch dieses Verfahren ist in Octave bereits vorimplementiert und lässt sich mit dem Befehl `lu(A)` aufrufen. Es ergeben sich bei dieser Methode Unterschiede, die durch die Anzahl der zur Verfügung gestellten Variable in denen die Lösung gespeichert werden soll, hervorgerufen

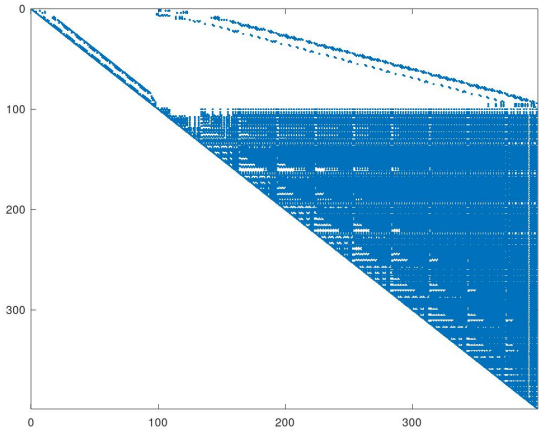
werden.
Gibt man der Methode zwei Variablen zum Speichern der Ergebnisse vor, so entsteht nicht wie erwartet eine obere und eine untere Dreiecksmatrix, sondern die in Abbildung 1.7a zu sehende Matrix **L** und die in Abb. 1.7c Matrix **U**. Stellt man der Routine vier Variablen `[L, U, P, Q]` zum Speichern zur Verfügung so entsteht wie in Abb. 1.7b und 1.7d zu sehen ist jeweils eine obere und untere Dreiecks Matrix. Zudem haben diese Matrizen deutlich weniger Einträge. Bei **P** und **Q** handelt es sich um Permutationsmatrizen. Mit diesen werden Zeilen und Spalten der Ausgangsmatrix **A** vertauscht, um zu kleine Pivot Elemente, die zu großen Rundungsfehlern führen können, zu verhindern und zu garantieren, dass die Zerlegung in eine obere und untere Dreiecksmatrix möglich ist.



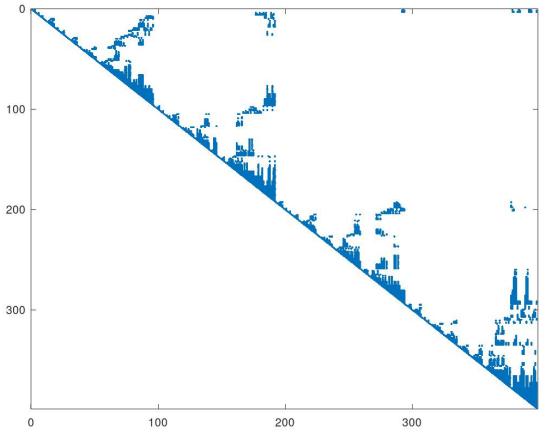
(a) Matrix **L** nach LU-Zerlegung



(b) Matrix **L** nach LUPQ-Zerlegung



(c) Matrix **U** nach LU-Zerlegung



(d) Matrix **U** nach LUPQ-Zerlegung

Abbildung 1.7: Darstellung der unterschiedlichen Matrizen, die abhängig der zur Verfügung gestellten Lösungsvariablen entstehen

Um den zeitlichen Aufwand der einzelnen Lösungsverfahren besser vergleichen zu können wird im Folgenden eine Messung der benötigten Rechenzeit in Octave durchgeführt und beschrieben. Die zu vergleichenden Lösungsverfahren zur Berechnung linearer Gleichungssysteme sind die LUPQ-Zerlegung, Lösung mit Hilfe des Backslash-Operators und die Berechnung mit der Inversen.

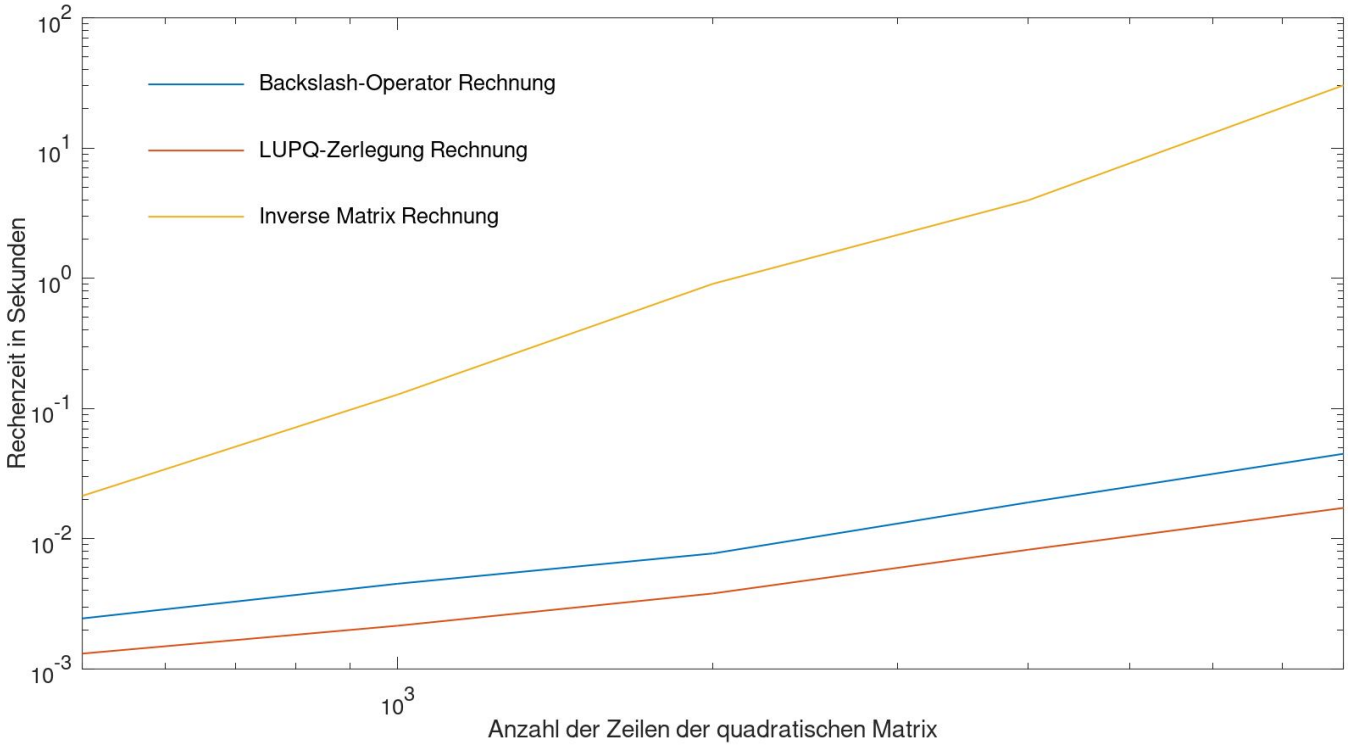


Abbildung 1.8: Graphische Darstellung der Rechenzeit, die unterschiedliche Lösungsverfahren zum Lösen linearer Gleichungen benötigen

Verschiedene tridiagonale Testmatrizen **A** der Größe $n \times n$ mit $n \in [500, 1000, 2000, 4000, 8000]$ haben jeweils 10 unterschiedliche, zufällige Seiten **b** zugewiesen bekommen. Die Zeit zur Berechnung jeder einzelnen Seite wurde gespeichert, die Berechnung der Inversen bzw. der LUPQ Zerlegung fließt nur bei der ersten Berechnung mit in die Rechenzeit ein.
Die Rechenzeit, die benötigt wurde um alle 10 Seiten zu berechnen wurde schließlich durch 10 geteilt, um eine durchschnittliche Rechenzeit zu erhalten. Die Ergebnisse wurden abschließend auf einem doppelt logarithmischen Koordinatensystem geplottet, wobei die x-Achse die Größe n der Matrix ist und die y-Achse die zur Berechnung einer Seite benötigte Rechenzeit.

Wie aus Abbildung 1.8 erkenntlich ist, benötigt die LUPQ-Zerlegung die geringste Zeit, um die Gleichungssysteme zu lösen. Der Backslash Operator benötigt mehr Zeit, ist aber, verglichen zur Berechnung mit Hilfe der Inversen, immer noch effizient.

2 Anhang

Methode odebwe_simple

```
function x =  odebwe_simple(t,x0,M,K,r)
  % initialize solution vector
  x = zeros(length(x0),length(t));
  x(:,1) = x0;

  for i = 2:length(t)
    % time step size
    h = t(i) - t(i-1);
    % system matrix
    A = (1/h)*M + K;
    % right hand side
    b = (1/h)*M*x(:,i-1)+r;
    % solve the linear system Ax=b
    x(:,i) = A\b;
  end
  x
endfunction
```

Listing 2.1: Methode `odebwe_simple` in Octave

Dahlquist Problem

```
h = 1e-1;
t = 0:h:5;
k = 1/5;
xana =@(t) (2*exp(-k*t));
% function x =  odebwe_simple(t,x0,M,K,r)
xnum = odebwe_simple(t,2,1,k,0);
plot(t,xana(t),t,xnum,'+');
legend('analytisch','numerisch');
xlabel('t-Achse');
ylabel('x-Achse');
```

Listing 2.2: Lösen des Dahlquist-Problem $x'(t) = -kx(t)$ mit $k = 1/5$ in Octave

Simulation Koaxialkabel

```
t = [0:1e-8:1e-3];

x0 = zeros(5,1);
x0(1) = 12;
x0(2) = 12;

M = zeros(5,5);
M(3,3) = 1e-7;
M(4,4) = 25e-5;

K = [1000 -1000 0 0 1; -1000 1000 0 1 0; 0 0 1/750 -1 0; 0 -1 1 0 0; -1  0 0 0 0];

r = zeros(5,1);
r(5) = -12;

% Funktion ist deutlich langsamer als die zuvor verwendeten Methoden
x = odebwe_simple(t,x0,M,K,r);

plot(t,x(3,:));
xlabel('Zeit t in [s]', 'interpreter', 'tex')
ylabel('Spannung in [V]', 'interpreter', 'tex')
legend('3. Zeile der Loesungsmatrix x')
```

Listing 2.3: Nummerische Lösung in Octave der bereits berechneten Aufgabe 3.1 a), nun aber mit eigener Methode `odebwe_simple`.

```
function val = diffquot(f,i,x,h)
  val = 0;
  switch (i)
    case 1
      val = (f(x+h)-f(x))/h;
    case 2
      val = (f(x+h)-f(x-h))/(2*h);
    case 4
      val = (f(x+h)-f(x-h)+2*f(x+h/2)-2*f(x-h/2))/(4*h);
  endswitch

endfunction
```

data/diffquot.m

```
h = [10e-11,10e-10,10e-9,10e-8,10e-7,10e-6,10e-5,10e-4,10e-3,10e-2,10e-1];
err1 = zeros(1,11);
err2 = zeros(1,11);
err4 = zeros(1,11);
f = @(x) exp(x+1);
for i = 1 : 11
  err1(1,i) = abs(12-diffquot(f,1,1,h(1,i)));
  err2(1,i) = abs(12-diffquot(f,2,1,h(1,i)));
  err4(1,i) = abs(12-diffquot(f,4,1,h(1,i)));
endfor

loglog(h,err1,'Linewidth',2,h,err2,'Linewidth',2,h,err4,'Linewidth',2);
l = legend('Differenzenquotient 1. Ordnung','Differenzenquotient 2. Ordnung','
    Differenzenquotient 4. Ordnung',"location", "northwest");
```

```
14 set(l,'Fontsize',22,'color','none');
15 legend boxoff;
16 xlabel('Schrittweite h','fontsize',22)
17 ylabel('Absoluter Fehler','fontsize',22)
18 set(gca,'Fontsize',20);
```

data/Aufgabe4_1.m

Abbildungsverzeichnis