Numerische Lösung Partieller Differentialgleichungen

Wir haben verschiedene einfache partielle Differentialgleichungen kennen gelernt – Bestimmungsgleichungen für stationäre oder zeitlich veränderliche Felder. Beispiele sind die Laplace- und Poisson-Gleichung für stationäre Felder, die Wärmeleitungs- und Diffusionsgleichung für zeitlich langsam veränderliche sowie die Wellengleichung für schnell veränderliche Felder. Solange die Geometrie des Problems nicht zu komplex ist, können wir analytische Lösungen dafür angeben, diese geben uns einen anschaulichen funktionalen Eindruck von der Struktur des Problems und der Lösung. Grundsätzlich geht das oft auch bei komplexeren Problemen, zumindest dann, wenn sie linear sind, d. h. wenn man Lösungsfunktionen ungestört überlagern kann. Die Gesamtlösung kann dann als Summe oder Integral über einfachere Lösungsfunktionen angegeben werden. Da dies jedoch rasch unübersichtlich wird, verwendet man bei technischen Problemen meist numerische Lösungsverfahren.

Einfache numerische Algorithmen beruhen auf der *Finite-Differenzen-Methode*, die Wikipedia-Beschreibung dazu sehen Sie in der ersten Box. Hier wer-

Wikipedia: Finite-Differenzen-Methode[a]

Die Finite-Differenzen-Methode ist das einfachste numerische Verfahren zur Lösung partieller Differentialgleichungen.
Zunächst wird das Gebiet, für das die Gleichung gelten soll, in eine endliche (finite) Zahl von Gitterzellen durch senkrecht aufeinander stehende Linien zerlegt. Die Ableitungen an den Gitterzellen werden dann durch Differenzen approximiert. Die partiellen Differentalgleichungen werden so in ein System von Differenzen-Gleichungen umformuliert und mittels verschiedener Algorithmen entweder implizit oder explizit gelöst.
Verfahren dieser Art finden verbreitete Anwendung bei hydrodynamischen Simulationen zum Beispiel in der Meteorologie und der Astrophysik.

[a] http://de.wikipedia.org/wiki/Finite-Differenzen-Methode

den wir uns ausschließlich mit solchen einfachen Algorithmen beschäftigen. Ein ähnlich klingendes Verfahren, die *Finite-Elemente-Methode*, sollte damit nicht verwechselt werden, nur zu Ihrer Information daher der Anfang der Wikipedia-Beschreibung zu dieser in der Technik inzwischen wichtigsten Methode zur Lösung von Differentialgleichungsproblemen in der zweiten Box. Die *Finite-Elemente-Methode* wurde ursprünglich für Probleme in der Elastizitätstheorie entwickelt; sie wird heute bei einer Vielzahl von physikalischen Problemstellungen eingesetzt – in der Regel mit aufwändigen kommerziellen Programmen.

Wikipedia: Finite-Elemente-Methode[a]

Die Finite-Elemente-Methode ist ein numerisches Verfahren zur näherungsweisen Lösung partieller Differentialgleichungen mit Randbedingungen.

Allgemeines Vorgehen: Das untersuchte Lösungsgebiet wird zunächst in Teilgebiete, die finiten Elemente eingeteilt.

$$G = \sum_{e=1}^{m} G_e \tag{1}$$

Innerhalb des Finiten Elements werden für die gesuchte Lösung je n Ansatzfunktionen definiert, die nur auf endlich vielen der Teilgebiete ungleich Null sind. Durch eine Linearkombination der n Ansatzfunktionen innerhalb des Elementes werden die möglichen Lösungen der numerischen Näherung festgelegt.

$$y|_{G_e} \approx \sum_{i=1}^{n} c_{e,n}\psi_{e,n} \tag{2}$$

Die Differentialgleichungen und die Randbedingungen werden mit Gewichtungsfunktionen multipliziert und über das Lösungsgebiet integriert. Das Integral wird durch eine Summe über einzelne Integrale der Finiten Elemente ersetzt, wobei die Integration i. d. R. durch eine näherungsweise numerische Integration ausgeführt wird. Da die Ansatzfunktionen nur auf wenigen der Elemente ungleich Null sind, ergibt sich ein dünnbesetztes, häufig sehr großes, lineares Gleichungssystem, bei dem die Faktoren der Linearkombination unbekannt sind.

Dieses Gleichungssystem könnte man zwar prinzipiell direkt (z. B. mit dem Gaußschen Eliminationsverfahren) lösen. Da der Berechnungsaufwand dort aber bei N Gleichungen $O(N^3)$ beträgt und beim Lösen die dünnbesetzte Struktur, die sich effizient speichern lässt, verloren geht, verwendet man im allgemeinen iterative Löser, die schrittweise eine Lösung verbessern.

...

[a] http://de.wikipedia.org/wiki/Finite-Elemente-Methode

1 Finite-Differenzen-Methode

Das Prinzip der *Finite-Differenzen-Methode* ist einfach und nahe liegend: Ableitungen werden durch endliche (*finite*) Differenzenquotienten ersetzt.
Für die erste Ableitung einer Funktion $f(x)$ bedeutet das

$$\frac{\mathrm{d}f}{\mathrm{d}x} \quad \Longrightarrow \quad \frac{f(x+h/2) - f(x-h/2)}{h} \,. \tag{3}$$

Entsprechend wird aus der zweiten Ableitung

$$\frac{\mathrm{d}^2 f}{\mathrm{d}x^2} \quad \Longrightarrow \quad \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \,. \tag{4}$$

Der Laplace-Operator in zwei Dimensionen wird für die Funktion $f(x,y)$ zu

$$\begin{aligned} \Delta f(x,y) \quad \Longrightarrow \quad & \frac{f(x+h_x,y) - 2f(x,y) + f(x-h_x,y)}{h_x^2} \\ & + \frac{f(x,y+h_y) - 2f(x,y) + f(x,y-h_y)}{h_y^2} \,. \end{aligned} \tag{5}$$

Wenn die beiden Koordinaten äquivalent sind, wählt man sinnvollerweise $h_y = h_x = h$, damit vereinfacht sich der Ausdruck (5) und ist in Programmen mit weniger Aufwand zu implementieren. Die Erweiterung von (5) auf drei oder mehr Dimensionen ist offensichtlich, wir werden uns hier jedoch der Einfachheit halber auf zwei Dimensionen beschränken.

Zur Anwendung des Verfahrens wird das kontinuierliche Gebiet, in dem die partielle Differentialgleichung gelöst werden soll, durch ein orthogonales, meist äquidistantes Feld aus Gitterpunkten ersetzt (Abbildung 1).

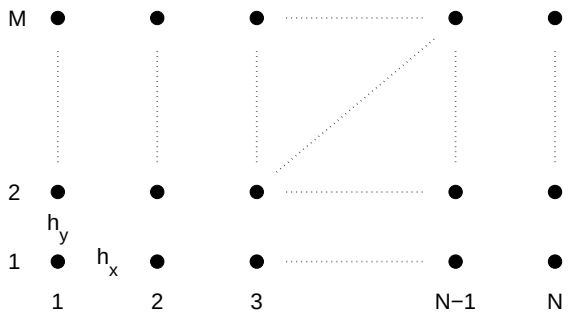

Abbildung 1: Zweidimensionales Punkteraster zur Implementierung der *Finite-Differenzen-Methode*.

Funktionswerte und Ableitungen werden für alle Berechnungen dann nur noch an den Gitterpunkten betrachtet. Für die graphische Darstellung von Ergebnisfunktionen kann – soweit notwendig – mit geeigneten Verfahren zwischen den Gitterpunkten interpoliert werden.

Statt der kontinuierlichen Variablen x und y werden Feldindizes $k = 1 \ldots N$ und $j = 1 \ldots M$ verwendet, damit wird (5) zu

$$\Delta A \implies \begin{array}{c} \dfrac{A_{j,k+1} - 2A_{j,k} + A_{j,k-1}}{h_x^2} \\ + \dfrac{A_{j+1,k} - 2A_{j,k} + A_{j-1,k}}{h_y^2} \end{array} \tag{6}$$

oder in MATLAB®-Schreibweise und mit $h_x = h_y =$ `h`

```
DeltaA = (A(j,k+1) + A(j,k-1) ...
        + A(j+1,k) + A(j-1,k) ...
        - 4*A(j,k))/h^2;
```
(7)

2 Laplace: Temperaturverteilung

Die oben genannte Strategie wenden wir auf das Lehrbuchbeispiel der Temperaturverteilung auf einer rechteckigen Platte an. Die Temperatur ist an allen Rändern vorgegeben – Dirichletsche Randbedingungen. Die stationäre Temperaturverteilung ist zu berechnen, d. h. für den Innenbereich ist die Laplace-Gleichung zu lösen.

2.1 Analytische Lösung

Zunächst die Lehrbuchlösung, um einen Vergleich zu den numerischen Verfahren zu haben (Abbildung 2).

Die wesentlichen Zeilen aus dem MATLAB®-Skript zur Berechnung der Temperaturverteilung mit der im Lehrbuch angegebenen Summation über Lösungsfunktionen (for-Schleife im Skript):

```
a = 20;
b = 20;
N = 20;
X = linspace(0,a,N);
Y = linspace(0,b,N);
```

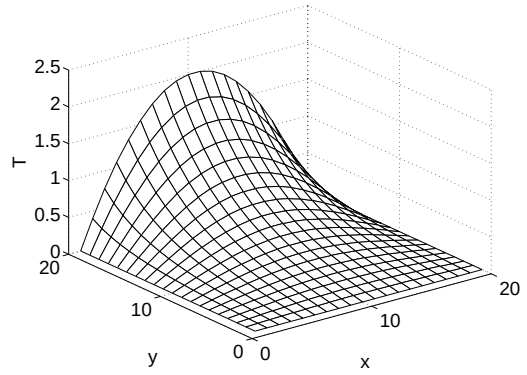

Abbildung 2: Stationäre Temperaturverteilung auf einer rechteckigen Platte bei Dirichletschen Randbedingungen (vgl. Lehrbuch).

```
[x,y] = meshgrid(X,Y);
T = zeros(size(x));
for n = 1:2:7,
  T = T + 1/n^3*sinh(n*pi*y/a) ...
      ./sinh(n*pi*b/a).*sin(n*pi*x/a);
end;
mesh(T);
```

2.2 Iterationsverfahren

Im stationären Fall gilt für das Temperaturfeld $\Delta T = 0$. Der Ausdruck (6) lässt sich damit für $A = T$ und $h_x = h_y$ umschreiben zu

$$T_{j,k} = 0.25(T_{j,k+1} + T_{j,k-1} + T_{j+1,k} + T_{j-1,k})\,. \tag{8}$$

Daraus wird eine Iterationsvorschrift, wenn man rechts den Zustand nach n, links den nach $n + 1$ Iterationen einsetzt:

$$T^{(n+1)}_{j,k} = 0.25(T^{(n)}_{j,k+1} + T^{(n)}_{j,k-1} + T^{(n)}_{j+1,k} + T^{(n)}_{j-1,k})\,. \tag{9}$$

Wichtig ist die richtige Behandlung der Randbedingungen. Dazu gibt es (mindestens) zwei Möglichkeiten: Man spart den Rand aus dem Wertebereich der Indizes j und k aus oder man regeneriert die Randbedingungen nach jedem Iterationsschritt. Die erste Möglichkeit wird man immer dann wählen, wenn nur eine rechteckige äußere Begrenzung mit festen Randwerten vorliegt (wie dies bei unserem Beispiel der Temperaturverteilung der Fall ist). Die zweite dann, wenn der Randverlauf komplizierter ist oder wenn zusätzliche Randbedingungen im Innenbereich zu berücksichtigen sind.

Die Iterationsvorschrift der Gleichung (9) lässt sich sehr leicht in ein Rechnerprogramm umsetzen, als Beispiel nachstehend ein Fragment in der Programmiersprache C, das aus dem Skriptum zu einer Numerik-Vorlesung an der Universität Bayreuth kopiert wurde:

```
for(iter=0; iter<1000; iter++)    /* iterations */
  for(i=1; i<(max-1); i++)        /* x-direction */
    for(j=1; j<(max-1); j++)      /* y-direction */
      p[i][j] = 0.25*(p[i+1][j]+p[i-1][j]
                     +p[i][j+1]+p[i][j-1]);
```

Drei ineinander geschachtelte Schleifen, eine über die Anzahl der Iterationen, eine über die Indizes in x und eine über y – das ist alles. Bei den for-Schleifen über x und y ist jeweils darauf geachtet, dass die Indizes nur über den Innenbereich laufen, der Rand bleibt daher während der Iteration unverändert auf den vor der Iteration eingestellten Werten.

In MATLAB® wird die Iterationsschleife deutlich kompakter, wenn man ausnutzt, dass MATLAB® mit dem kompletten Temperaturfeld als Matrix rechnen kann. Hier das MATLAB®-Codefragment, in dem zunächst das gesamte Feld auf Null gesetzt wird und am hinteren Rand `T(end,:)` ein parabelförmiger Temperaturverlauf als Randbedingung eingestellt wird.

```
%%%%% Start- und Randbedingungen
M = 20;
N = 20;
T = zeros(M,N);
x = linspace(-1,1,N);
T(end,:) = 2*(1-x.*x);
%%%%% Indizes der inneren Punkte
J = 2:size(T,1)-1;
K = 2:size(T,2)-1;
%%%%% Iteration
for n = 1:NumberOfIterations
    T(J,K) = 0.25*(T(J,K+1) + T(J,K-1)...
        + T(J+1,K) + T(J-1,K));
end
```

Die Indexvektoren `J` und `K` adressieren den Innenbereich, damit ist gewährleistet, dass die Randbedingungen bei der Iteration nicht gestört werden. Das Ergebnis der Rechnung entspricht dann auch voll unseren Erwartungen (Abbildung 3).

Konvergenz und Genauigkeit: Die in den Abbildungen 2 und 3 dargestellten Resultate sehen identisch aus, gewisse Unterschiede stellt man jedoch fest, wenn die beiden Temperaturfelder übereinander geplottet werden. Dies ist in Abbildung 4 gemacht.

Offensichtlich liegt der mit dem Iterationsverfahren berechnete Temperaturverlauf noch unter dem exakten Ergebnis. Wir untersuchen das genauer, indem wir uns einen Schnitt durch das Temperaturfeld ansehen. Dort, wo die Abweichungen

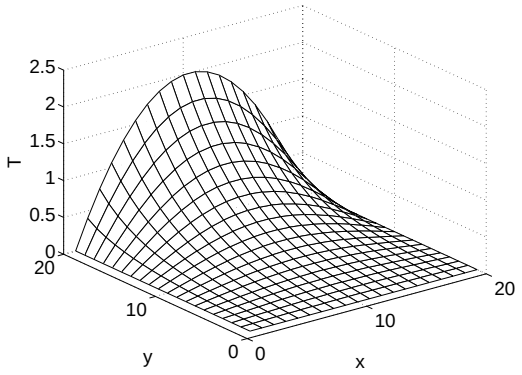

Abbildung 3: Stationäre Temperaturverteilung auf einer rechteckigen Platte bei Dirichletschen Randbedingungen, berechnet mit dem beschriebenen Iterationsverfahren.

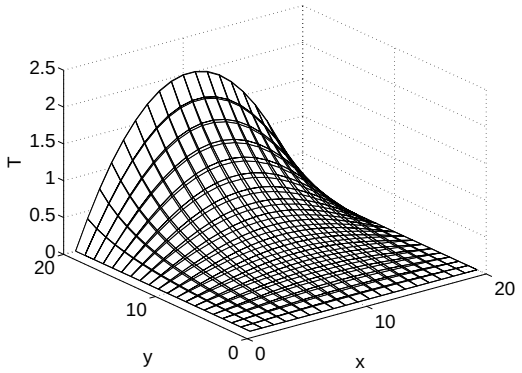

Abbildung 4: Ergebnis der Iterationsrechnung und exaktes Ergebnis übereinander geplottet.

am auffälligsten sind – in der Symmetrieebene, die mittendurch geht (das ist die durch die y- und T-Richtung aufgespannte Fläche bei $x = 10$). In Abbildung 5 ist der Temperaturverlauf in der Schnittebene skizziert, im linken Teilbild für das bisher betrachtete Temperaturfeld mit 20×20 Punkten. Die graue Kurve repräsentiert das exakte Ergebnis, die schwarzen Kurven Iterationsrechnungen mit 30, 150 bzw. 500 Iterationen. Man sieht, dass erst bei einigen hundert Iterationen das Ergebnis die gewünschte Genauigkeit erreicht. Noch aufschlussreicher wird der Vergleich, wenn man mit einem feineren Punkteraster arbeitet. Das ist im rechten Teilbild gemacht, das Temperaturfeld besteht dort aus 100×100 Punkten. Grau wieder das exakte Ergebnis und schwarz die Resultate der Iterationsrechnungen – nun aber mit 500, 2000 bzw. 10000 (!) Iterationen. In Tabelle 1 sind die Maximalabweichungen vom exakten Ergebnis für die verschiedenen Szenarios zusammen gefasst.

Wir lernen daraus:

- Das Iterationsverfahren zur numerischen Lösung der Laplace-Gleichung ist außerordentlich einfach zu implementieren, insbesondere in MATLAB®.
- Das Konvergenzverhalten ist problematisch, vor allem bei hoher Auflösung.

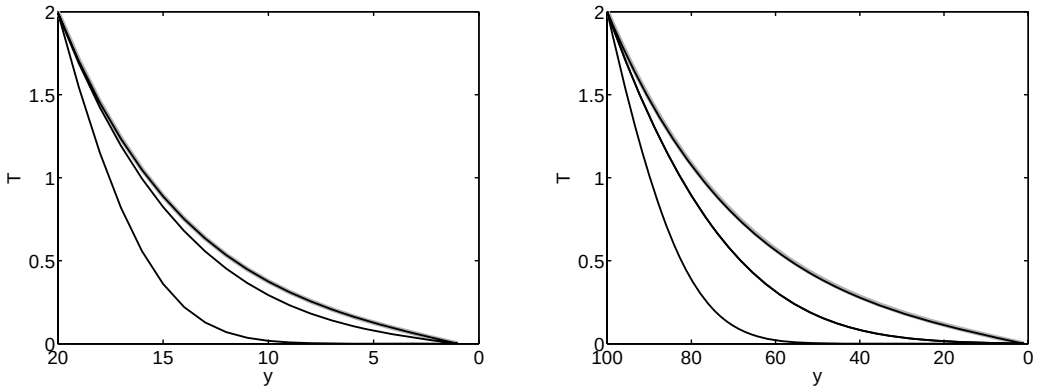

Abbildung 5: Berechneter Temperaturverlauf in der Mitte der rechteckigen Platte. Grau der exakt berechnete Verlauf, schwarz die Ergebnisse von Iterationsrechnungen mit unterschiedlicher Anzahl der Iterationen. Links für ein Punkteraster von 20×20 Punkten und 30, 150 bzw. 500 Iterationen. Rechts für ein Punkteraster von 100×100 Punkten und 500, 2000 bzw. 10000 Iterationen.

Feldgröße	20×20			100×100		
Zahl der Iterationen	30	150	500	500	2000	10000
Maximale Abweichung	0.532	0.084	0.001	0.712	0.253	0.006

Tabelle 1: Maximalabweichungen der Iterationsrechnungen vom exakten Ergebnis für unterschiedliche Punktraster und unterschiedliche Anzahl von Iterationen.

- Wenn man das Verfahren anwendet, sollte man zuerst an einem Testproblem Genauigkeit und Konvergenz überprüfen, um die Zahl der notwendigen Iterationen festlegen zu können.
- Stattdessen kann man das Verfahren auch so lange laufen lassen, bis sich die Ergebnisse aufeinander folgender Iterationen nicht mehr unterscheiden, d. h. bis die Unterschiede unter einer vorher festgelegten Grenze liegen.

2.3 Lineares Gleichungssystem

Wenn wir den Ausdruck (6) für $A = T$ und $h_x = h_y$ auf andere Weise umschreiben als in Gleichung (8), definieren wir damit ein homogenes lineares Gleichungssystem für die $T_{j,k}$

$$T_{j,k+1} + T_{j,k-1} + T_{j+1,k} + T_{j-1,k} - 4T_{j,k} = 0\,, \tag{10}$$

das für die Punkte (j, k) gilt, die nicht durch Randbedingungen festgelegt sind.

Dazu kommen einfache Zuweisungen, welche die Randpunkte den Randbedingungen gemäß festlegen. Eine solche Zuweisung können wir als inhomogene lineare Gleichung auffassen.

Packen wir alle Gleichungen zusammen, erhalten wir ein inhomogenes linearen Gleichungssystem, das aus $M \times N$ Gleichungen besteht, wenn das betrachtete Gebiet aus $M \times N$ Punkten besteht. Pro Punkt mithin entweder eine Gleichung wie (10) oder die Zuweisung eines Randwerts. Damit ist die numerische Lösung der Laplace-Gleichung auf die Lösung eines linearen Gleichungssystems zurückgeführt. Dafür gibt es Standardverfahren.

Zur vereinfachten Darstellung wird das Gleichungssystem in Matrixschreibweise formuliert

$$\mathsf{L} * T = R\,. \tag{11}$$

L ist die Koeffizientenmatrix, T das zu einem (eindimensionalen) Vektor umgeformte Temperaturfeld, R fasst die Konstanten auf der rechten Seite des Gleichungssystems – Randbedingungen oder Null – zu einem Vektor zusammen. Überlegen Sie, welche Lösungsverfahren für das so formulierte Problem Sie kennen.

Dünn besetzte Matrizen: Bei großen Feldern, d. h. bei einigermaßen guter Ortsauflösung, landet man sehr schnell bei umfangreichen Koeffizientenmatrizen. Eine Ortsauflösung von 100×100 beispielsweise führt zu 10000 Gleichungen – und damit zu einer Koeffizientenmatrix mit 10^8 Elementen. Analysiert man jedoch die Struktur dieser Matrix, so wird offensichtlich, dass nur etwa 50000 dieser Elemente ungleich Null und damit relevant sind, alle anderen verschwinden. Da dünn besetzte Matrizen (englisch: *sparse matrices*) in Physik und Technik bei sehr vielen numerischen Problemen auftreten (für Berechnungen ist fast immer nur die nähere Umgebung wichtig), wurden dafür spezielle Algorithmen entwickelt. MATLAB® verwendet solche Algorithmen, wenn man Matrizen als *sparse* gekennzeichnet hat. Sie werden dann auch als Listen $\{j_i, k_i, \ldots, x_i\}$ mit den von Null verschiedenen Elementen x_i und den zugehörigen Indizes $j_i, k_i, \ldots$ im Speicher abgelegt, um Platz zu sparen. Über die verschiedenen Möglichkeiten, die MATLAB®-Funktion `sparse` zu verwenden, gibt `help sparse` oder `doc sparse` Auskunft.

Die Lösung eines linearen Gleichungssystems kann in MATLAB® durch die formale Umkehrung von Gleichung (11) veranlasst werden

$$\mathsf{L} * T = R \quad \Longrightarrow \quad T = \mathsf{L} \backslash R\,. \tag{12}$$

MATLAB® berechnet die Lösung des Gleichungssystems dann mit dem Gauß-Verfahren.

Die Aufgabe liegt nun darin, das Differentialgleichungsproblem für MATLAB® ‘mundgerecht’ zu formulieren, d. h. die Matrix **L** und den Vektor R richtig zu konstruieren.

Die Geometrie des Problems wird durch eine Matrix `G` beschrieben; die Werte auf dem Rand werden mit den Randbedingungen besetzt, die Werte im Innern zunächst mit *Not-a-Number* (`NaN`)[1]:

```
M = 20;
N = 20;
G = NaN(M,N);
G(1,:) = 0;
G(:,[1,end]) = 0;
x = linspace(-1,1,N);
G(end,:) = 2*(1-x.*x);
```

Für die von Null verschiedenen Werte der Koeffizientenmatrix **L** legen wir eine Tabelle an, in der die Indizes `j, k` und die zugehörigen Werte `s` stehen. **L** hat die Größe `(M*N)`×`(M*N)`, die Indizes können mithin die Werte `1...(M*N)` annehmen. Alle Diagonalelemente (`k=j`) von **L** werden mit 1.0 belegt.

```
j = (1:M*N)';
k = j;
s = ones(size(j));
```

Die Matrixelemente, die den im Innern liegenden (`isnan(G)`) direkt benachbart sind (`[-1,1,-M,M]`), werden mit -0.25 besetzt (Laplace). Zur richtigen Adressierung müssen die beiden Indexfelder `u` und `v` in den entsprechenden `index` in der Matrix **L** umgerechnet werden. In der Schleife wird die Wertetabelle um die Nichtdiagonalelemente ergänzt. Schließlich wird die `NaN`-Markierung entfernt, die Innenwerte werden auf Null gesetzt (rechte Seiten der Gleichungen für die Innenpunkte).

```
[u,v] = find(isnan(G));
index = (v-1)*M+u;
for h = [-1,1,-M,M],
    j = [j;index];
    k = [k;index+h];
    s = [s;-0.25*ones(size(index))];
end;
G(isnan(G)) = 0;
```

[1] MATLAB® speichert Zahlenwerte entsprechend dem IEEE-Standard 754. In dieser Norm sind auch spezielle Bitkombinationen der 64 bzw. 32 Bit definiert, die $+\infty$, $-\infty$ und *Not-a-Number* entsprechen. MATLAB® verwendet diese Werte z. B. als Ergebnisse von Rechenoperationen wie 1/0 (=Inf), 0/0 (=NaN) oder Inf-Inf (=NaN). NaN bleibt bei allen Rechenoperationen NaN, daher kann dieser Wert sehr gut als Markierung verwendet werden.

Das Weitere sieht dann sehr ‘sparsam’ aus, aus der Tabelle wird die Matrix `L`, aus der zweidimensionalen Beschreibung `G` der rechten Seite der Vektor `R`. Und nachdem das Gleichungssystem gelöst ist, muss der Ergebnisvektor noch in eine Matrix der ursprünglichen Größe `M*N` umgeformt werden.

```
L = sparse(j,k,s);
R = G(:);
y = L \ R;
T = reshape(y,M,N);
```

Das Resultat (Abbildung 6) ist von der exakten Lösung nicht zu unterscheiden.

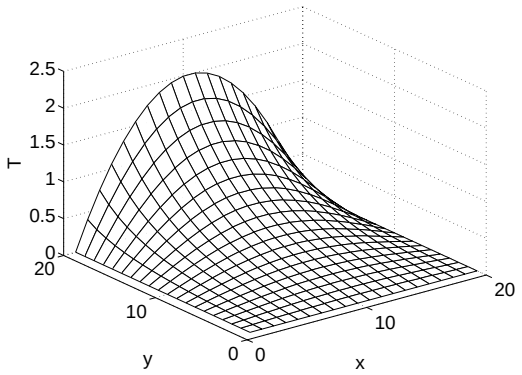

Abbildung 6: Stationäre Temperaturverteilung auf einer rechteckigen Platte bei Dirichletschen Randbedingungen, berechnet mit MATLAB® durch Lösung des äquivalenten linearen Gleichungssystems.

3 Laplace: Elektrolytischer Trog

Nach der einfachen Platte wird nun eine etwas komplexere Problemstellung numerisch gelöst. Der *Elektrolytische Trog* ist ein Laborversuch im Anfängerpraktikum. Aufgabe bei diesem Versuch ist es, Potenzialverteilungen um verschiedene Elektrodenanordnungen auszumessen. Dazu werden in einer mit Wasser gefüllten Plastikwanne Kupferelektroden angeordnet, an denen Gleichspannungen angelegt werden (Größenordnung 10 V). Mit einem empfindlichen Messsystem werden dann Äquipotenziallinien bestimmt.

Eine solche Elektrodenanordnung mit drei Elektroden ist in Abbildung 7 skizziert. Mit dieser Anordnung wird die Potenzialverteilung in einer Elektronenröhre mit drei Elektroden – Kathode (K), Gitter (G) und Anode (A) – modelliert. Solche *Trioden* waren bis zur späten Mitte des letzten Jahrhunderts die in der Elektronik überwiegend verwendeten aktiven Bauelemente. Die Kathode besteht aus einem Material mit geringer Elektronenaustrittsarbeit, sie wird auf Rotglut aufgeheizt und liefert so Elektronen. Diese wandern zum positiven Potenzial der Anode, sofern sie genügend kinetische Energie besitzen, um das durch das Gitter aufgebaute Sperrpotenzial zu überwinden. Durch die Gitterspannung lässt sich somit der

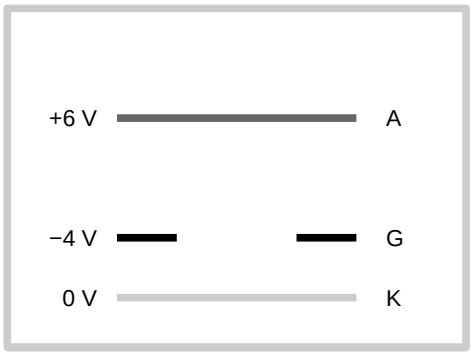

Abbildung 7: Dreielektrodenanordnung im Elektrolytischen Trog. Mit der Anordnung werden die Potenzialverhältnisse an einer Triode mit Kathode K, Gitter G und Anode A modelliert. Links die im Modell angelegten Spannungen (bei einer realen Elektronenröhre sind die Spannungen um zwei Größenordnungen höher).

Anodenstrom regeln. Heute findet man das Prinzip noch bei Bildröhren (Helligkeitssteuerung).

Es ist wieder die Laplace-Gleichung $\Delta V = 0$ für das ganze Gebiet numerisch zu lösen, neben den Randbedingungen auf der Außenbegrenzung ($V = 0$) sind nun auch innere Randbedingungen zu berücksichtigen.

Die Beschreibung der Geometrie wird in eine eigene m-Datei ausgelagert, das vereinfacht die Wiederverwendung des eigentlichen Programms.

```
function G = geotriode(N)
if nargin==0, s=6; else s = round(N/8); end;
xm = 8*s-1;
ym = 6*s-1;
G = NaN(ym,xm);
G([1,end],:) = 0;
G(:,[1,end]) = 0;
G(s,2*s:6*s) = 0;
G(2*s,[2*s:3*s,5*s:6*s]) = -4;
G(4*s,2*s:6*s) = 6;
```

Das Feld `G` wird mit den Randbedingungen, ansonsten mit `NaN` vorbesetzt. Mit `M` bzw. dem Skalierungsfaktor `s` wird die Größe des Feldes und somit die Auflösung festgelegt.

3.1 Iterationsverfahren

Trotz der schon diskutierten Nachteile schauen wir uns zuerst nochmals die Implementierung des Iterationsalgorithmus an.

Von der Geometrie `G` der Elektrodenanordnung wird das Startpotenzial `V` abgeleitet.

```
G = geotriode;
V = G;
V(find(isnan(G))) = 0;
```

Warum dafür eine zusätzliche Variable verwendet wird, erklärt sich später.

Die Indexvektoren `J` und `K` adressieren wieder den Innenbereich.

```
J = 2:size(V,1)-1;
K = 2:size(V,2)-1;
```

In der Iterationsschleife kommt eine Anweisung dazu, die in der Potenzialmatrix `V` den Anfangszustand aller Randbedingungen restauriert. Dazu wird `G` nochmals benötigt.

```
for n = 1:nIterations
    V(J,K) = 0.25*(V(J,K+1) + V(J,K-1)...
        + V(J+1,K) + V(J-1,K));
    V(find(~isnan(G))) = G(find(~isnan(G)));
end
```

In Abbildung 8 sind die aus dem Resultat der Iterationen, dem Potenzial, berechneten Äquipotenziallinien gezeichnet. MATLAB® macht das mit dem Funktionsaufruf

```
contour(V,8,'EdgeColor','k');
```

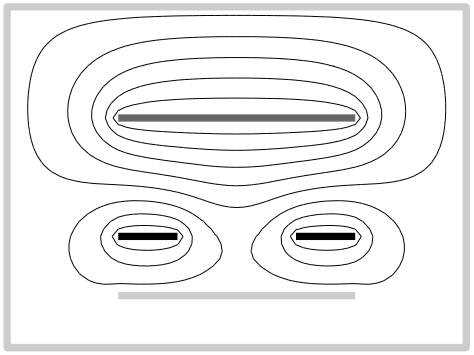

Abbildung 8: Äquipotenziallinien um die Dreielektrodenanordnung der Abbildung 7.

So ähnlich sollte auch das Ergebnis des Praktikumsversuchs aussehen.

3.2 Lineares Gleichungssystem

Gegenüber der in Abschnitt 2.3 beschriebenen Vorgehensweise ändert sich nur der Geometrieteil:

```
G = geotriode;
[M,N] = size(G);
```

Das restliche MATLAB®-Skript bleibt so wie dort beschrieben.

Abbildung 9 zeigt das berechnete Potenzial, diesmal als Potenzialgebirge für Elektronen dargestellt. Man erkennt deutlich das 'Tal', durch das die Elektronen zur Anode gelangen können. Es wird auch evident, dass sich die Hindernishöhe durch die Gitterspannung steuern lässt.

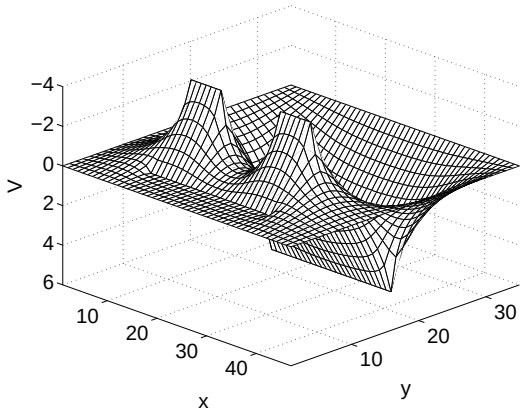

Abbildung 9: Potenzialverlauf um die Dreielektrodenanordnung der Abbildung 7.

4 Wärmeleitungsgleichung

Diffusionsgleichung und Wärmeleitungsgleichung gehören zu den partiellen Differentialgleichungen des Typs

$$\frac{\partial T}{\partial t} = K\,\Delta T\,. \tag{13}$$

Beschränken wir uns auf den eindimensionalen Fall, wird daraus

$$\frac{\partial T}{\partial t} = K\,\frac{\partial^2 T}{\partial x^2}\,. \tag{14}$$

Zur numerischen Berechnung werden die Differentialquotienten durch Differenzenquotienten ersetzt

$$\frac{T(x,t+\tau)-T(x,t)}{\tau} = K\,\frac{T(x-h,t)-2T(x,t)+T(x+h,t)}{h^2}\,. \tag{15}$$

Von einer Anfangssituation $T(x,t_0)$ ausgehend, können wir somit die zeitliche Entwicklung berechnen, indem wir Gleichung (15) nach $T(x,t+\tau)$ auflösen

$$T(x,t+\tau) = T(x,t) + \frac{\tau\,K}{h^2}\left(T(x-h,t)-2T(x,t)+T(x+h,t)\right). \tag{16}$$

Die Vorgehensweise entspricht dem Euler-Verfahren, die Genauigkeit hängt von der Wahl der Schrittweite τ ab.

Die kontinuierlichen Variablen x und t werden durch Indizes im Temperaturfeld T ersetzt

$$T_{j,k+1} = T_{j,k} + \frac{\tau K}{h^2} \left(T_{j-1,k} - 2T_{j,k} + T_{j+1,k} \right). \tag{17}$$

Zu Vektoren und Matrizen zusammengefasst wird daraus

$$\begin{bmatrix} T_{1,k+1} \\ T_{2,k+1} \\ \vdots \\ \vdots \\ T_{M,k+1} \end{bmatrix} = \begin{bmatrix} T_{1,k} \\ T_{2,k} \\ \vdots \\ \vdots \\ T_{M,k} \end{bmatrix} + \frac{\tau K}{h^2} \begin{bmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} T_{1,k} \\ T_{2,k} \\ \vdots \\ \vdots \\ T_{M,k} \end{bmatrix}. \tag{18}$$

Wichtig ist nun noch zu überlegen, was an den Rändern passiert. Die Matrix in Gleichung (18) hört am Rand einfach auf. Das ist gleichbedeutend damit, dass die Temperaturwerte außerhalb mit dem Gewicht Null berücksichtigt werden

$$T_{1,k+1} = T_{1,k} + \frac{\tau K}{h^2} (0 \cdot T_{0,k} - 2 \cdot T_{1,k} + 1 \cdot T_{2,k}). \tag{19}$$

Statt $0 \cdot T_{0,k}$ können wir auch $1 \cdot T_{0,k}$ schreiben, wenn wir annehmen, dass $T_{0,k} = 0$. Gleichung (18) legt also implizit Randbedingungen fest, die Randwerte – genauer gesagt, die Werte gerade außerhalb – sind Null. Wenn man andere feste Randwerte vorgeben will, muss man nach jedem Rechenschritt die Randwerte $T_{1,k+1}$ und $T_{M,k+1}$ auf die Vorgabewerte zurück setzen und damit die gewünschten Dirichletschen Randbedingungen durchsetzen.

Beispiel: δ-Injektion: Ein langer Draht wurde punktuell erhitzt, beide Enden liegen fest auf Nulltemperatur. Die Implementierung in MATLAB®:

```
M = 100;
K = 0.2;
x = zeros(M,1);
x(round(M/3)+[0,1]) = 50;
T = x;
L = diag(ones(M-1,1),-1) - 2*diag(ones(M,1))...
    + diag(ones(M-1,1),1);
for n = 1:3000,
    x = x + K*L*x;
    if mod(n,100)==0
        T = [T,x];
    end
end;
```

Alle Konstanten sind zu einem einzigen `K` zusammengefasst, das damit implizit sowohl die physikalischen Größen enthält, die für die Wärmeleitung zuständig sind, wie auch die Größen, die die Diskretisierung festlegen. Die δ-förmige Erhitzung wird bei einem Drittel der Drahtlänge angenommen. Als Summe aus drei Diagonalen ist die Matrix `L` konstruiert, die Matrixdarstellung für den Laplace-Operator. Insgesamt 3000 Zeitschritte werden berechnet, der Übersichtlichkeit halber wird jedoch nur jede hundertste Situation im Temperaturfeld abgelegt.

Das Ergebnis, die Zeitabhängigkeit der Temperaturverteilung im Draht ist in Abbildung 10 dargestellt. Erwartungsgemäß wird aus der Delta-Funktion eine gaußähnliche.

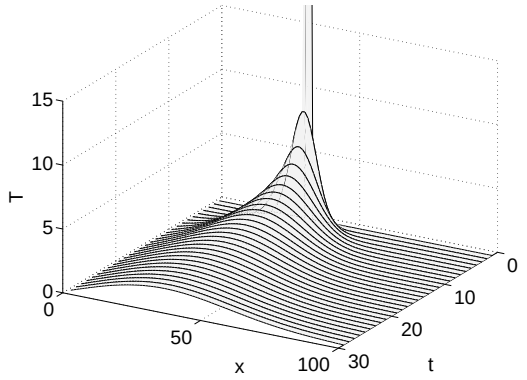

Abbildung 10: Zeitlicher Verlauf der Temperaturverteilung in einem Draht nach δ-förmiger Erhitzung.

Von Neumannsche Randbedingungen: Bisher wurden konstante Funktionswerte am Rand vorausgesetzt (Dirichletsche Randbedingungen). Bei Wärmeleitungsproblemen bedeutet das beliebig leistungsfähige Wärmequellen oder -senken am Rand auf fest vorgegebenen Temperaturen. Geht man stattdessen davon aus, dass der Wärmestrom am Rand konstant ist, so muss man dort konstante erste Ableitungen des Funktionswerts (Temperatur) annehmen (von Neumannsche Randbedingungen). Formal wird dies dadurch implementiert, dass man ein etwas vergrößertes Gebiet betrachtet und die nächsten außerhalb des ursprünglichen Gebiets liegenden Punkte zusätzlich berücksichtigt. Ihr Funktionswert wird aus dem nächstliegenden inneren Punkt und der ersten Ableitung an dieser Stelle berechnet. Für jeden zusätzlichen Punkt (hier Index 0) ergibt sich somit eine zusätzliche Gleichung der Art

$$\frac{T_{2,k} - T_{0,k}}{2h} = m\,, \tag{20}$$

in der h die Schrittweite und m die Steigung bedeuten. Besonders einfach wird das, wenn die Steigung Null ist. Für das Beispiel der Wärmeleitung bedeutet dies verschwindenden Wärmestrom an dieser Stelle, d. h. gute thermische Isolation.

Der Außenpunkt kann dann dadurch berücksichtigt werden, dass in der Matrix `L` an der Position $(2, k)$ eine 2 statt einer 1 steht.

Zeitlich veränderliche Randbedingungen: Da im MATLAB®-Programm die einzelnen Rechenschritte explizit ausgeführt werden, können die Randbedingungen nach Belieben verändert werden. Ein einfaches Beispiel soll dies veranschaulichen. Eine Wand[2] wird auf einer Seite unterschiedlichen Temperaturen ausgesetzt und ist auf der anderen Seite isoliert. Im MATLAB®-Progamm sind dafür nur marginale Änderungen nötig:

```
x = zeros(M,1);
...
L(end,end-1) = 2;
for n = 1:3000,
    x = x + K*L*x;
    if n<1500
        x(1) = -20;
    else
        x(1) = 20;
    end
    if mod(n,100)==0
        T = [T,x];
    end
end;
```

Die Ausgangssituation ist eine konstante Temperatur von Null im gesamten Bereich. Auf einer Seite wird mit `L(end,end-1)=2` die horizontale Tangente gefordert, auf der anderen Seite werden Temperatursprünge auf -20 und +20 eingestellt.

Das Ergebnis – Abbildung 11 – zeigt, wie eine Wand Temperatursprünge dämpft.

MATLAB®, die Funktion *pdepe*: Zur Lösung von partiellen Differentialgleichungsproblemen dieser Art (erste Ableitung nach der Zeit, erste und zweite Ortsableitung in einer Dimension) bietet MATLAB® die Funktion *pdepe* (*P*artial *D*ifferential *E*quation of *P*arabolic or *E*lliptic Type). Diese MATLAB®-Funktion löst Differentialgleichungen der allgemeinen Struktur

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right)+s\left(x,t,u,\frac{\partial u}{\partial x}\right) \quad (21)$$

[2]Nicht nur Probleme, die im geometrischen Sinne eindimensional sind wie Drähte o. ä. können eindimensional berechnet werden, sondern auch solche, die in zwei Raumdimensionen näherungsweise konstant und nur in einer veränderlich sind. Das sind meist die technisch interessanteren (Fenster, Wände, Erdatmosphäre, Meeresoberfläche, Erdkruste etc.).

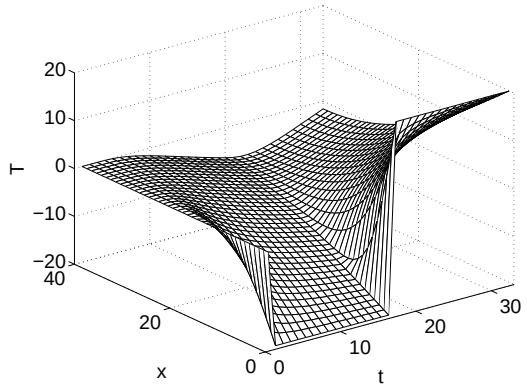

Abbildung 11: Wärmeleitung in einer Wand – Reaktion auf einen Temperatursprung.

mit $m = 0, 1, 2$ für ebene, zylindrische oder sphärische Geometrien. Zur Berechnung verwendet MATLAB® einen *solver* für gewöhnliche Differentialgleichungen (`ode15s`), der unter anderem auch eine automatische Schrittweitenanpassung bei den Zeitschritten macht. Eine ausführliche Beschreibung findet man in der MATLAB®-Hilfe unter `doc pdepe`.

Um Ihnen eine Eindruck davon zu geben, wie man damit arbeitet, folgt hier das 'Wandproblem' in der Formulierung mit *pdepe*:

```
function hT = wallpdepe
m = 0;
x = 1:40;
t = linspace(0,3000,30);
sol = pdepe(m,@pdepde,@pdeic,@pdebc,x,t);
T = sol(:,:,1);
hT = mesh(T','Edgecolor','k');
% --------------------------------------------------
function [c,f,s] = pdepde(x,t,u,DuDx)
c = 1;
f = 0.2*DuDx;
s = 0;
% --------------------------------------------------
function u0 = pdeic(x)
u0 = x*0;
% --------------------------------------------------
function [pl,ql,pr,qr] = pdebc(xl,ul,xr,ur,t)
pl = ul+20-40*((t>1500)*(t-1500)-(t>1501)*(t-1501));
ql = 0;
pr = 0;
qr = 1;
```

Struktur der Differentialgleichung, Anfangs- und Randbedingungen werden jeweils durch spezielle Funktionen beschrieben, die beim Aufruf an `pdepe` als

Parameter übergeben werden. `pdepde` definiert die Differentialgleichung, d. h. die Parameter `c`, `f` und `s` darin, `pdeic` die Anfangsbedingungen und `pdebc` die Randbedingungen. `pl`, `ql`, `pr`, `qr` sind darin implizite Formulierungen für links- und rechtsseitige Funktionswerte und Ableitungen. Die Beschreibung des Temperatursprungs in `pl` ist etwas umständlich, weil der Algorithmus keine abrupten Änderungen der Randbedingungen verträgt.

Das von *pdepe* produzierte Ergebnis (Abbildung 12) entspricht natürlich unseren Erwartungen.

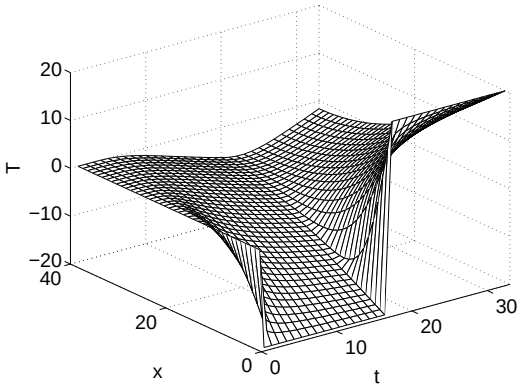

Abbildung 12: Wärmeleitung in einer Wand, berechnet mit der MATLAB®-Funktion *pdepe* – Reaktion auf einen Temperatursprung.

5 Wellengleichung – Eigenmoden

Zitat aus dem Lehrbuch: »Die *Wellengleichung* enthält die zweite zeitliche Ableitung, d. h. nicht die zeitliche Veränderung des Feldes (also eine Geschwindigkeit) sondern die zeitliche Änderung dieser Änderung (also eine Beschleunigung):

$$\Delta A = \frac{1}{c^2}\frac{\partial^2 A}{\partial t^2}\,. \tag{22}$$

Sie beschreibt schnell veränderliche, in der Regel periodische Vorgänge.«

Mit dem *Separationsansatz* $A = R(x,y,z)T(t)$ und der *Separationskonstante* λ werden aus Gleichung (22) die beiden Gleichungen

$$\frac{\mathrm{d}^2 T}{\mathrm{d}t^2} + \lambda c^2 T = 0 \tag{23}$$

$$\Delta R + \lambda R = 0\,. \tag{24}$$

Die Differentialgleichung für die Zeitabhängigkeit lässt sich immer abseparieren, die entstandene Schwingungsgleichung hat die üblichen Lösungen. Bei der weiteren Separation des räumlichen Teils muss man die Geometrie des Problems berücksichtigen. Bei einfachen Geometrien (Quader, Rechteck, Kreisscheibe, Zylinder, Kugel) kann man – mit geeignet gewählten Koordinaten – weiter separieren

und landet letztlich bei einem System von gewöhnlichen Differentialgleichungen, die durch Separationskonstanten miteinander verknüpft sind.

Als Beispiele für die *numerische* Behandlung sehen wir uns ein einfaches eindimensionales Problem an, die schwingende Saite, sowie einige zweidimensionale, Rechteckmembran, Kreismembran und gekoppelte Membranen.

5.1 Eindimensional: Saite

Im eindimensionalen Fall wird der räumliche Teil eine gewöhnliche Differentialgleichung

$$\frac{\mathrm{d}^2 R}{\mathrm{d}x^2} + \lambda R = 0\,. \tag{25}$$

Zur numerischen Behandlung wird die zweite Ableitung wie üblich durch Differenzenquotienten ersetzt und wir erhalten die Matrixgleichung

$$\mathsf{L}\, R + \lambda h^2\, R = 0\,. \tag{26}$$

Darin ist h die Schrittweite im Vektor R, h^2 kann zur Rechnung mit λ zusammengefasst werden.

$$\mathsf{L}\, R + \Lambda\, R = 0\,. \tag{27}$$

Gleichung (27) hat die triviale Lösung $R = 0$, außerdem eine der Länge des Vektors R entsprechende Anzahl von nichttrivialen Lösungen – die *Eigenvektoren* R_n und die zugehörigen *Eigenwerte* Λ_n. Bei der Saite entsprechen die Eigenvektoren den Schwingungsmoden. Zur Berechnung der Eigenvektoren und der Eigenwerte einer Matrix gibt es in MATLAB® die Funktionen `eig` für übliche Matrizen und `eigs` speziell für dünn besetzte Matrizen. Die Anwendung von `eig` auf das eindimensionale Problem veranschaulicht nachstehendes Fragment:

```
M = 100;
L = diag(ones(M-1,1),-1) - 2*diag(ones(M,1))...
    + diag(ones(M-1,1),1);
[u,v] = eig(L);
```

Die Konstruktion von `L` läuft wie beim Wärmeleitungsproblem, die Rückgabewerte von `eig` sind die Eigenfunktionen `u` und die Eigenwerte `v`. In Abbildung 13 sind die so berechneten ersten 4 Eigenschwingungen der Saite skizziert.

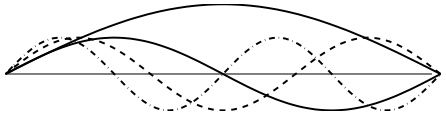

Abbildung 13: Grundschwingung und Oberschwingungen einer Saite.

5.2 Zweidimensional: Rechteckmembran

Zur numerischen Lösung wird der räumliche Teil der Differentialgleichung nicht weiter separiert. $\Delta R + \lambda R = 0$ (Gleichung (24)) wird nach der Diskretisierung zum *Eigenwertproblem*

$$\mathbf{L}\, R = \Lambda\, R\,. \tag{28}$$

Der Vektor R umfasst alle inneren Punkte der Membran eindimensional aufgereiht, in $\mathbf{L}$ sind die Laplace-Operatoren – ausgedrückt durch finite Differenzen – zusammengefasst.

Während wir bei der eindimensionalen Problemstellung der Saite noch bequem mit der kompletten Matrix rechnen konnten, würde das bei zwei oder mehr Dimensionen sehr platzaufwändig, insbesondere bei etwas besserer Auflösung. Daher ist es zweckmäßig, mit dünn besetzten Matrizen zu arbeiten und damit mit der MATLAB®-Funktion `eigs` zur Berechnung der Eigenvektoren und Eigenwerte[3].

Die numerische Lösung zerlegen wir in drei Teile – die Beschreibung der Geometrie, die Aufstellung der Matrix $\mathbf{L}$ und die eigentliche Lösung des Eigenwertproblems.

Die Geometrie der Membran soll durch eine Matrix beschrieben werden, alle Ränder liegen auf Null, ebenso alle Punkte außerhalb der Membran, alle inneren Punkte werden auf Eins gesetzt. Die rechteckige Membran beschreiben wir somit durch

```
m = 30;
n = 40;
G = zeros(m,n);
G(2:m-1,2:n-1) = 1;
```

Aufwändiger ist das Zusammenstellen von $\mathbf{L}$. Das wird der besseren Wiederverwendbarkeit wegen in ein getrenntes Skript ausgelagert, das als Eingabe die Geometriematrix erhält und die Matrix `L` zurück liefert, außerdem zwei Vektoren `J` und `K`, die beschreiben, wie die Indizes in `L` auf die Indizes im Geometriefeld umgerechnet werden

[3]Während `eig` exakt rechnet, verwendet `eigs` Näherungsverfahren zur Berechnung. Die Ergebnisse sind zwar gut, aber nicht im strengen Sinne exakt. Man sollte daher nur dort zu dünn besetzten Matrizen wechseln, wo dies nötig ist.

```
function [L,J,K] = laps(G)
% sparse Laplacian
[M,N]=size(G)
Nv=0;
for j=1:M
    for k=1:N
        if G(j,k)==1
            Nv=Nv+1;
            J(Nv)=j;
            K(Nv)=k;
            A(j,k)=Nv;
        end
    end
end
L=sparse(Nv,Nv);
for a=1:Nv
    j=J(a);
    k=K(a);
    L(a,a)=-4;
    if G(j+1,k)==1
        L(a,A(j+1,k))=1;
    end
    if G(j-1,k)==1
        L(a,A(j-1,k))=1;
    end
    if G(j,k+1)==1
        L(a,A(j,k+1))=1;
    end
    if G(j,k-1)==1
        L(a,A(j,k-1))=1;
    end
end
```

Im ersten Teil werden die inneren Punkte der Geometrie gezählt und durchnummeriert, außerdem die Indexzuordnung erstellt. Danach wird die dünn besetzte Matrix `L` angelegt und deren von Null verschiedene Elemente besetzt (Diagonalelemente mit -4, innere in der Geometrie benachbarte mit 1).

Die Lösung des Eigenwertproblems erledigt dann die MATLAB®-Funktion `eigs`, schließlich wird der eindimensionale Ergebnisvektor wieder in die richtige Geometrie eingefüllt:

```
[u,v] = eigs(L,N,'sm');
for z = 1:size(u(:,N)),
    G(J(z),K(z)) = u(z,N);
```

```
end;
```

Einige Moden sind in Abbildung 14 zusammengestellt.

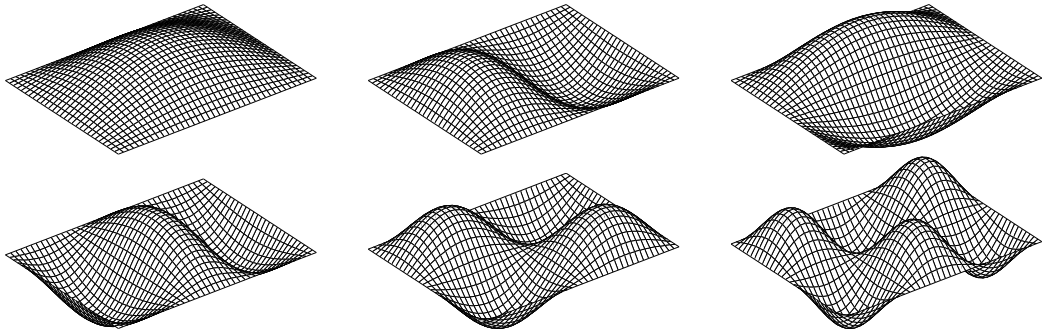

Abbildung 14: Numerische berechnete Eigenmoden 1 bis 6 der schwingenden Rechteckmembran.

5.3 Schwingende Kreismembran – Bessel-Funktionen

Zunächst zum Vergleich die analytische Beschreibung mit Bessel-Funktionen. MATLAB® kennt die Bessel-Funktionen, die benötigten der ersten Art werden über `besselj` aufgerufen. Wir können uns daher ohne großartigen Programmieraufwand ein paar Schwingungsmoden skizzieren lassen. Das Skript dazu erstellt mit `meshgrid` ein zweidimensionales Koordinatensystem, in dem dann gerechnet wird. Die Funktion `besselzero`, die die Nullstellen der Besselfunktionen berechnet, ist im Internet zu finden[4].

```
function besselmembrane(n,k,phi)
% Eigenmoden einer Kreismembran
if nargin<3, phi = 0; end
if nargin<2, k = 2; end
if nargin<1, n = 0; end
N = 64;
x = linspace(-1.1,1.1,N);
[X,Y] = meshgrid(x);
R = sqrt(X.*X+Y.*Y);
PHI = acos(X./R);
xi = besselzero(n,k,1);
u = besselj(n,xi(k)*R).*(R<=1).*cos(phi*PHI);
u(find(R>1.05)) = NaN;
```

[4] `http://www.mathworks.com/matlabcentral/fileexchange/` – Dank an Peter Hertel für den Hinweis.

```
mesh(x,x,0.6*u,'Edgecolor','k');
axis equal off tight
```

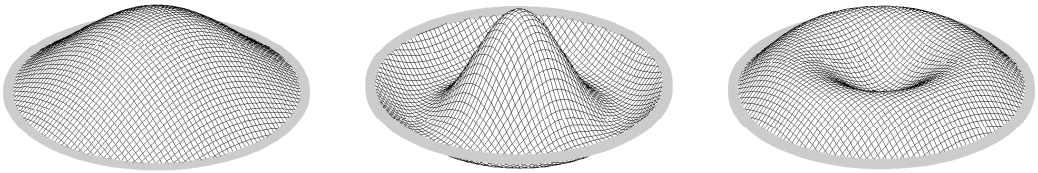

Abbildung 15: Rotationssymmetrische Schwingungsmoden der Kreismembran, links und in der Mitte 0. Ordnung rechts 2. Ordnung.

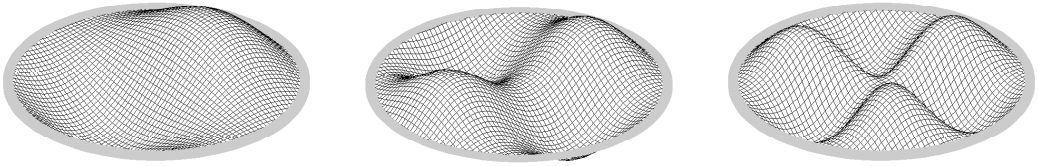

Abbildung 16: Nicht rotationssymmetrische Schwingungsmoden der Kreismembran, links 1. Ordnung Winkelabhängigkeit $\cos(\phi)$, in der Mitte 2. Ordnung $\cos(2\phi)$, rechts 2. Ordnung $\cos(3\phi)$.

5.4 Schwingende Kreismembran – numerische Lösung

Die numerische Lösung funktioniert wie bei der Rechteckmembran – einzig die Erstellung der Geometrie wird etwas aufwändiger als dort:

```
M = 64;
x = linspace(-1.1,1.1,M);
[X,Y] = meshgrid(x);
R = sqrt(X.*X+Y.*Y);
G = zeros(M);
G(find(R<=1)) = 1;
```

Ein paar Resultate zeigt Abbildung 17.

5.5 Gekoppelte Membranen

Als komplexeres Beispiel berechnen wir die Eigenmoden eines Systems aus zwei gekoppelten Kreismembranen. Die Geometrie ist in Abbildung 18 dargestellt, zwei gleich große Membranen sind durch eine schmale Verbindung miteinander gekoppelt. Beschrieben wird das durch

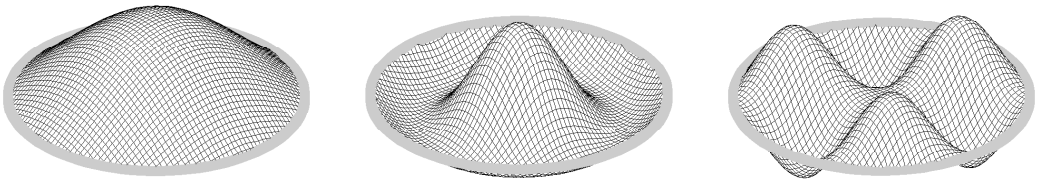

Abbildung 17: Numerische Lösungen: Eigenmoden 1, 6 und 7 der schwingenden Kreismembran.

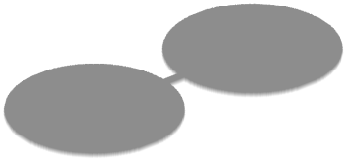

Abbildung 18: Geometrie der zwei gekoppelten Kreismembranen.

```
S = 20;
M = 2*S;
N = 4*S;
x = linspace(-2.2,2.2,N);
y = linspace(-1.1,1.1,M);
[X,Y] = meshgrid(x,y);
G = zeros(M,N);
R = sqrt((X-1.1).^2+Y.^2);
G(find(R<=1)) = 1;
R = sqrt((X+1.1).^2+Y.^2);
G(find(R<=1)) = 1;
G(S+[-1:1],2*S+[-S:S]) = 1;
```

Die Lösung dann wieder wie gewohnt, einige Ergebnisse in Abbildung 19.

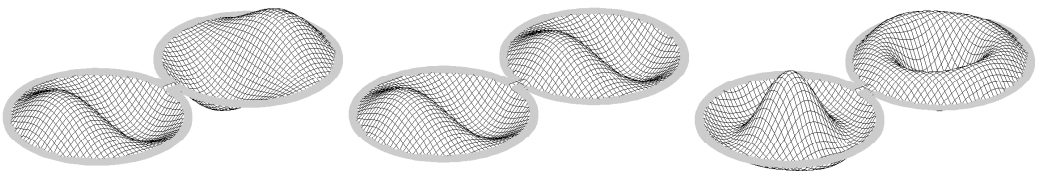

Abbildung 19: Eigenmoden 3, 4 und 12 der gekoppelten Kreismembranen. Die Moden 3 und 4 wären bei Einzelmembranen identisch, durch die Kopplung werden die Eigenwerte, d. h. die zugehörigen Energien bzw. Frequenzen unterschiedlich.