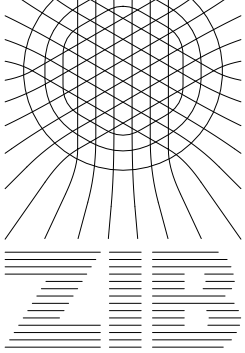
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7, D-14195 Berlin-Dahlem



MATLAB–Programme zur Lösung von Eigenwertproblemen mit einem adaptiven, nichtlinearen Mehrgitter–Verfahren

Tilmann Friese

e-mail: friese@zib.de

Technical Report TR 97–01 (Februar 1997)

Inhaltsverzeichnis

Einleitung

Die vorliegenden MATLAB–Programme entstanden im Rahmen der **„Förderung von anwendungsorientierten Verbundprojekten auf dem Gebiet der Mathematik“** des BMBF in dem Projekt *„Entwicklung von adaptiven Mehrgitter–Methoden zur Berechnung von Eigenlösungen der Helmholtzgleichung für Halbleiterstrukturen der integrierten Optik“* (Projektnummer 03-DE7ZIB-5). Das Projekt unter Leitung von Prof. Deuflhard wurde in Kooperation mit der Siemens AG München und dem Heinrich–Hertz–Institut für Nachrichtentechnik Berlin GmbH durchgeführt.
Ausgehend von einem monotonen Mehrgitter–Verfahren zur Minimierung des Rayleigh–Quotienten im Falle der reellen Helmholtzgleichung [1] wurde ein Algorithmus für die komplexe Helmholtzgleichung entwickelt. Wesentliche Idee ist der Übergang zur Schur–Zerlegung des komplexen Helmholtz–Operators. Analog den Eigenvektoren im reellen Fall bilden die Schurvektoren im allgemeinen Fall unter bestimmten Voraussetzungen eine Orthonormalbasis des zugrundeliegenden Hilbertraumes. Eine detailierte Beschreibung der Mehrgitter–Methode zur Lösung der Eigenwertaufgabe der komplexen Helmholtzgleichung ist Gegenstand der Arbeit [2].
Das Programmpaket wurde für die Simulation integriert–optischer Komponenten entwickelt. Der physikalische Aufbau dieser Komponenten ist gekennzeichnet durch Schichtenfolgen mit extrem unterschiedlicher Dicke sowie große und abrupte Brechzahlübergänge, wobei im verlustbehafteten Fall komplexe Brechzahlen mit großen Imaginärteilen auftreten. Der zugrunde liegende Algorithmus gestattet die direkte Berechnung der geführten Moden einer integriert–optische Komponente. Das bedeutet insbesondere für komplexe Brechzahlprofile, daß bekannte Techniken wie Pfadverfolgung oder Störungsrechnung nicht benötigt werden. Die Verwendung der simultanen Iterationsmethode ermöglicht zudem die Bestimmung entarteter und fast entarteter Eigenmoden. Durch den gewählten allgemeinen Zugang erschließen sich neben der integrierten Optik weitere Anwendungsgebiete wie beispielsweise die Behandlung von Eigenwertproblemen aus der Moleküldynamik.

Danksagung

An dieser Stelle möchte ich Herrn Dr. Reinhard März von der Siemens AG München für die stets gute und anregende Kooperation während meiner Tätigkeit im Projekt danken.

Kurzbeschreibung des Algorithmus

Im vorgestellten MATLAB–Programmpaket ist ein adaptives, nichtlineares Mehrgitter–Verfahren zur Berechnung von Eigenlösungen von partiellen Differentialoperatoren der Form

$$-\Delta u(x,y) - f(x,y)\,u(x,y) = \lambda\,u(x,y)\,, \quad (x,y) \in \Omega \tag{1}$$

implementiert. Dabei ist Ω ein beschränktes, polygonal begrenztes Gebiet im $\mathbb{R}^2$ und f eine beschränkte, u. U. komplexwertige Funktion auf Ω. Als Randbedingungen können die Dirichlet–Randbedingung

$$u(x,y) = 0\,, \quad (x,y) \in \partial\Omega$$

und die Neumann–Randbedingung

$$\frac{\partial u}{\partial \nu}(x,y) = \nu \cdot \nabla u(x,y) = 0\,, \quad (x,y) \in \partial\Omega$$

mit dem Richtungsvektor der äußeren Normalen ν auftreten. Die Diskretisierung aus der Variationsformulierung von (1) mit linearen finiten Elementen über einem Dreiecksgitter (Triangulierung) führt auf das verallgemeinerte algebraische Eigenwertproblem

$$(B^*SB)\,u = \lambda\,(B^*MB)\,u\,, \quad u \in \mathbb{C}^N\,,$$

wobei S die Steifigkeitsmatrix, M die Massenmatrix und B die Randbedingungsmatrix ist.
Das MATLAB–Programmpaket gestattet die Berechnung der q Eigenwerte mit kleinstem Realteil. Zentrale Idee des Algorithmus ist die Nutzung der Schur–Zerlegung

$$\begin{aligned}(B^*SB)\,U &= (B^*MB)\,U\,T\\ U^*\,(B^*MB)\,U &= I\end{aligned}$$

des Matrixpaares $\Big((B^*SB),(B^*MB)\Big)$. Dabei ist T eine obere $(N \times N)$–Dreiecksmatrix mit den Eigenwerten in der Diagonalen und I die Identität.

Bemerkung

Die ausschließliche Benutzung der Neumann–Randbedingung bei über ganz Ω konstantem f erfordert einige Zusätze in der Implementierung, die in dieser Version nicht enthalten sind (Grund: die konstante Funktion $u \equiv \frac{1}{\sqrt{|\Omega|}}$ ist Eigenfunktion zum Eigenwert mit dem kleinsten Realteil, näheres siehe [2]).

Beschreibung der einzelnen Routinen

Grundsätzliches

Zentrale Routine des Programmpaketes ist die Funktion `main`. Ihr Argument ist eine Zeichenkette mit dem Problemnamen (z. B. `problem = 'lshape'`), welche die partielle Differentialgleichung und deren Diskretisierung spezifiziert. Der Differentialoperator und die Diskretisierung werden beschrieben durch

- die Funktion f und
- die Triangulierung,

die jeweils in einem eigenen Verzeichnis abgespeichert werden. Befindet sich die Routine `main` z. B. im Verzeichnis `~/eigen`, so sind die Unterverzeichnisse `~/eigen/fun` und `~/eigen/geo` für die Funktion f bzw. die Triangulierung anzulegen.
Die Funktion f wird als MATLAB–`m`–File im Verzeichnis `~/eigen/fun` abgespeichert. Der Name dieses Files muß dabei identisch mit dem Problemnamen sein (also z. B. `lshape.m`). Einige Beispiele für solche Funktions–Files sind:

- die Nullfunktion

```
function f = lshape(x, y)
f = zeros(size(x));
```

- eine stetige Funktion

```
function f = oscillator(x, y)
f = -(x.^2 + 2i*x.*y + y.^2);
```

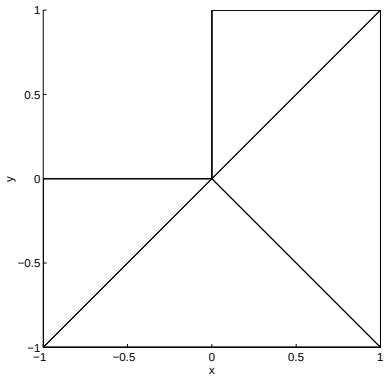
- eine Sprungfunktion

```
function f = pot1(x, y)
f = zeros(size(x));
j = find(x > 3/8 & x < 5/8 & y > 3/8 & y < 5/8);
f(j) = (100 + 250i) * ones(size(j));
```

Die Triangulierung wird als MATLAB–`mat`–File im Verzeichnis `~/eigen/geo` abgespeichert. Auch hier muß der Name des Files identisch mit dem Problemnamen sein (also z. B. `lshape.mat`). Eine Triangulierung wird beschrieben durch:

- die Punktmatrix `p`
- die Randkantenmatrix `e`
- die Dreiecksmatrix `t`

Die Punktmatrix `p` besteht aus zwei Zeilen. Die erste Zeile enthält die x–Koordinaten, die zweite Zeile die y–Koordinaten der Gitterpunkte.
Die Randkantenmatrix `e` ist dreizeilig. Die ersten beiden Zeilen enthalten die Indizes der Start– bzw. Endpunkte der Randkanten bezüglich `p`, die dritte Zeile die Randbedingungen auf den Kanten. Dabei entspricht 0 der Dirichlet–Randbedingung, 1 der Neumann–Randbedingung.
Die Dreiecksmatrix `t` besteht aus drei Zeilen. Diese enthalten die Indizes der Eckpunkte der Dreiecke, wobei im Gegenuhrzeigersinn numeriert wird.
Ein Beispiel für eine Triangulierung eines L–Gebietes ist:




```
p = [-1    1    1    0    0   -1
     -1   -1    1    1    0    0]

e = [ 1    2    3    4    5    6
      2    3    4    5    6    1
      1    1    0    0    0    0]

t = [ 1    2    3    1
      2    3    4    5
      5    5    5    6]
```

Die Speicherung der Gitterdaten erfolgt mit dem MATLAB–Kommando `save`. Ist der aktuelle MATLAB–Pfad `~/eigen`, so ist beispielsweise zur Speicherung obiger Daten die Sequenz `save geo/lshape p e t` auszuführen.
Eine Triangulierung für Rechteck–Gebiete kann mit der Routine `rectmesh` erzeugt werden.

Bemerkung

Bei Sprungfunktionen ist die Triangulierung so zu wählen, daß jede Kante, an der die Funktion f nicht stetig ist, durch eine Dreieckskante erfaßt wird.

Beispiele

Am Beispiel des Gebietes $\Omega = (0,1)^2$ und der Sprungfunktion

$$f(x,y) = \begin{cases} 100 + 250\mathrm{i}, & (x,y) \in \left(\frac{3}{8}, \frac{5}{8}\right)^2 \\ 0, & \text{sonst} \end{cases}$$

sollen die einzelnen Schritte eines Programmlaufes im Zusammenhang aufgezeigt werden. Als Randbedingung soll die Dirichlet–Randbedingung verwendet werden. Das MATLAB–Funktionsfile ist auf Seite 3 angegeben und wird im Verzeichnis `~/eigen/fun` unter dem Namen `pot1.m` abgelegt. Danach wird MATLAB im Verzeichnis `~/eigen` gestartet. Zur Generierung und Speicherung einer Start–Triangulierung sind folgende Schritte auszuführen:

```
>> x = [0 3/16 3/8 1/2 5/8 13/16 1];
>> y = [0 3/16 3/8 1/2 5/8 13/16 1];
>> [p, e, t] = rectmesh(x, y, 7);
>> save geo/pot1 p e t
```

Da die Routine `rectmesh` als Voreinstellung die Dirichlet–Randbedingung benutzt, sind keine weiteren Eingaben durchzuführen. Soll die Neumann–Randbedingung verwendet werden, sind die entsprechenden Komponenten in der Matrix `e` abzuändern. Die grafische Darstellung der Start–Triangulierung erfolgt mit

```
>> meshplot(p, e, t); axis('square');
```

Es sollen die 4 Eigenwerte mit kleinstem Realteil und die zugehörigen Schurvektoren berechnet werden, wobei 5 Gitterverfeinerungen und eine relative Genauigkeit von 10^{-5} zur Lösung der diskreten Systeme gewünscht sind. Die entsprechenden Werte sind nach dem Start der Routine `main` einzugeben: dabei ist `q` die Anzahl der zu bestimmenden Eigenwerte, `lmax` die Anzahl der adaptiven Gitterverfeinerungen und `rtol` die relative Genauigkeit. Die Voreinstellungen für diese Werte sind `q = 1`, `lmax = 5` und `rtol = 1e-5`. Sie werden durch eine leere Eingabe (nur Return drücken) zugewiesen. Der Start von `main` und die Eingabe der Größen (in diesem Beispiel nur `q`) ergibt die folgende Sequenz:

```
>> [p, e, t, U, T, err] = main('pot1');
```

```
Number of desired eigenvalues:
q = 4

Number of adaptive mesh refinements:
lmax =
Default: lmax = 5

Relative accuracy of the discrete solutions:
rtol =
Default: rtol = 1e-5

Level 0: |p| = 49, |t| = 72

Level 1: |p| = 157, |t| = 272
cycle: 1 2 3 4

Level 2: |p| = 321, |t| = 592
cycle: 1 2 3 4

Level 3: |p| = 445, |t| = 824
cycle: 1 2 3 4

Level 4: |p| = 925, |t| = 1768
cycle: 1 2 3 4 5

Level 5: |p| = 1441, |t| = 2768
cycle: 1 2 3 4

cputime = 253.7 s
flops = 778594213
```

Die grafische Darstellung der End–Triangulierung erfolgt wie oben, die der Schurvektoren mit

```
>> for j = 1:4
     polyplot(p, t, U(:,j).*conj(U(:,j))); axis('square');
     pause;
   end;
```

Die gesuchten Eigenwerte sind die Diagonalelemente der Dreiecks–Matrix `T`:

```
>> T

T =

1.0e+02 *

0.3936-1.8283i  0.2982+0.8580i -0.0000-0.0000i  0.0000+0.0000i
     0          0.5088-0.1306i  0.0000-0.0000i -0.0000-0.0000i
     0               0          0.5458-0.1068i -0.0000+0.0000i
     0               0               0          0.5458-0.1068i
```

Hier sind die Eigenwerte $\lambda_3 =$ `T(3,3)` und $\lambda_4 =$ `T(4,4)` identisch (entartetes Eigenwertproblem). Die relativen Residuen pro Mehrgitter–Zyklus zur End–Triangulierung werden mit

```
>> ncycle = 4;
>> semilogy(0:ncycle, err); hold on;
>> semilogy(0:ncycle, err, 'o'); hold off;
```

dargestellt. Die Berechnung der Eigenvektoren `V` aus den Schurvektoren `U` zu den gefundenen Eigenwerten wird mit

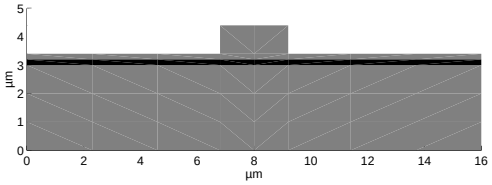
```
>> [X, D] = eig(T, 'nobalance');
>> V = U * X;
```

durchgeführt. Die grafische Darstellung der Eigenvektoren erfolgt wie die der Schurvektoren. Im Fall einer reellwertigen Funktion f kann der letzte Schritt entfallen, da Eigenvektoren und Schurvektoren identisch sind.
Neben diesem und einigen weiteren Standardbeispielen sind im Programmpaket zwei Beispiele aus der integrierten Optik enthalten:

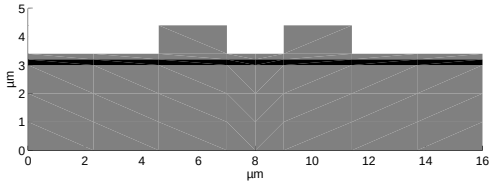
- eine Rippen–Wellenleiter–Struktur (`problem = 'waveguide'`)
- eine Koppler–Struktur (`problem = 'coupler'`)

Die Komponenten bestehen jeweils aus einem III–V–Mischhalbleitersystem (InP, GaInAsP) und haben folgende Querschnitte:

Rippen–Wellenleiter–Struktur



Koppler–Struktur



Die Werte für die Dielektrizitätszahlen bestimmen die Funktion f:

$$f(x,y) = \begin{cases} k_0^2, & \text{Luft: weiß} \\ (3.17k_0)^2, & \text{InP: grau} \\ (3.38k_0)^2, & \text{GaInAsP: schwarz} \end{cases}$$

Dabei ist $k_0 = \frac{2\pi}{1.55}$ die Wellenzahl von Licht der Wellenlänge 1.55 μm im Vakuum. Da der Koppler symmetrisch ist, existieren bei dieser Struktur zwei geführte, fast entartete Moden (symmetrischer und asymmetrischer Eigenmode). Bei Start der Routine `main` ist daher die Anzahl der zu bestimmenden Eigenwerte `q = 2` zu setzen (simultane Berechnung der geführten Moden).

Überblick

Das MATLAB–Programmpaket enthält die folgenden Routinen, die sich nach ihrer jeweiligen Funktion in zwei Gruppen einteilen lassen:

Direkte Routinen für den Anwender	
`main`	Lösung des Eigenwertproblems mit einem adaptiven Mehrgitter–Verfahren
`meshplot`	Grafische Darstellung der Triangulierung
`polyplot`	Grafische Darstellung von Funktionen über polygonal begrenzten Gebieten
`rectmesh`	Erzeugung einer Start–Triangulierung für Rechteck–Gebiete

Interne Routinen des Programmpaketes	
`adjac`	Bestimmung der Adjazenz–Matrix der Triangulierung
`assemmat`	Kompilation der System–Matrizen
`diagsort`	Sortieren der Eigenwerte in der Schur–Zerlegung
`errorind`	Fehlerschätzer für adaptive Gitterverfeinerung
`jacobi`	Jacobi–Verfahren für Eigenwertprobleme
`mgm`	Rekursive Mehrgitter–Routine
`polybound`	Bestimmung der Randbedingungs–Matrix
`polyrefinemesh`	Verfeinerung einer Triangulierung eines polygonal begrenzten Gebietes
`res`	Bestimmung des Residuums
`smoother`	Glättungs–Routine für das Mehrgitter–Verfahren
`solvese`	Lösung der Sylvester–Gleichung

Eine genaue Beschreibung jeder einzelnen Routine ist auf den folgenden Seiten enthalten.

• adjac

Zweck:

Bestimmung der Adjazenz–Matrix der Triangulierung

Syntax:

```
A = adjac(p, t)
```

Beschreibung:

`A = adjac(p, t)` gibt die Adjazenz–Matrix der durch `p` und `t` spezifizierten Triangulierung zurück. Dabei ist `A(i,j)` genau dann ungleich Null, falls die Punkte i und j durch eine Dreieckskante verbunden sind bzw. falls $i = j$.

• assemmat

Zweck:

Kompilation der System–Matrizen

Syntax:

```
[S, M, scal] = assemmat(p, t, problem)
```

Beschreibung:

`[S, M, scal] = assemmat(p, t, problem)` assembliert die Steifigkeitsmatrix `S`, die Massenmatrix `M` und die Skalierungsgröße `scal`. Die Eingabeparameter `p` und `t` beschreiben die Triangulierung, der Eingabeparameter `problem` ist eine Zeichenkette mit dem Problemnamen.

• diagsort

Zweck:

Sortieren der Eigenwerte in der Schur–Zerlegung

Syntax:

`[Q, T] = diagsort(Q, T, q)`

Beschreibung:

`[Q, T] = diagsort(Q, T, q)` sortiert die in der Diagonalen von `T` stehenden Eigenwerte nach deren Realteil in aufsteigender Ordnung. Damit stehen die q Eigenwerte mit den kleinsten Realteilen in den Diagonalelementen `T(1,1)` bis `T(q,q)`.

• errorind

Zweck:

Fehlerschätzer für adaptive Gitterverfeinerung

Syntax:

```
errf = errorind(p, e, t, B, U, T, problem)
```

Beschreibung:

`errf = errorind(p, e, t, B, U, T, problem)` gibt eine Liste von geschätzten Fehlern zur aktuellen Triangulierung zurück. Die Komponente `errf(j)` korrespondiert dabei zum j–ten Dreieck `t(:,j)`. Die Eingabeparameter `p`, `e` und `t` beschreiben die Triangulierung. `B` ist die Randbedingungs–Matrix, `U` sind die bis auf die relative Genauigkeit `rtol` (siehe `main`) bestimmten Schurvektoren mit der zugehörigen Schurschen oberen Dreiecks–Matrix `T`. Der String `problem` enthält den Problemnamen.

• jacobi

Zweck:

Jacobi–Verfahren für Eigenwertprobleme

Syntax:

```
Delta_U = jacobi(S, M, U, T, ip)
```

Beschreibung:

`Delta_U = jacobi(S, M, U, T, ip)` gibt die Lösung des Gleichungssystems

$$\mathrm{diag}(S)\,\Delta U - \mathrm{diag}(M)\,\Delta U\,T = -R$$

mit $R = S\,U - M\,U\,T$ zurück. Dabei werden nur die der Indexliste `ip` entsprechenden Komponenten bestimmt, die übrigen werden Null gesetzt. Die Eingabeparameter `S` und `M` sind die System–Matrizen, `U` die aktuellen Schurvektoren mit der zugehörigen Schurschen oberen Dreiecks–Matrix `T`.

• main

Zweck:

Lösung des Eigenwertproblems mit einem adaptiven Mehrgitter–Verfahren

Syntax:

```
[p, e, t, U, T, err] = main(problem)
```

Beschreibung:

`[p, e, t, U, T, err] = main(problem)` bestimmt die q Eigenwerte mit kleinstem Realteil und die dazugehörigen Schurvektoren des Problems `problem` mit einem adaptiven Mehrgitter–Verfahren. Der Eingabeparameter `problem` ist eine Zeichenkette mit dem Namen des Problems. Zum Aufruf der Routine müssen ein MATLAB–`m`–File, welches die Funktion f enthält, und ein MATLAB–`mat`–File, das die Triangulierung beschreibt, mit dem gleichen Namen in den Unterverzeichnissen `fun` bzw. `geo` abgespeichert sein (siehe Grundsätzliches). Es sind folgende Eingaben erforderlich:

- `q`: Anzahl der zu bestimmenden Eigenwerte, Voreinstellung `q = 1`
- `lmax`: Anzahl der adaptiven Gitterverfeinerungen, Voreinstellung `lmax = 5`
- `rtol`: relative Toleranz zur Lösung der diskreten Systeme, Voreinstellung `rtol = 1e-5`. Die Mehrgitter–Iteration wird ausgeführt, solange gilt:

  ```
  any(res(S, M, U, T, scal) > rtol)
  ```

Die voreingestellten Werte werden jeweils durch eine leere Eingabe (nur Return drücken) angenommen. Die Rückgabeparameter `p`, `e` und `t` beschreiben die Endtriangulierung, die q-spaltige Matrix `U` enthält die bis auf die relative Genauigkeit `rtol` bestimmten Schurvektoren zur Endtriangulierung, `T` ist eine obere Dreicksmatrix mit q Zeilen und Spalten und den gesuchten Eigenwerten in der Diagonalen und `err` eine q-spaltige Matrix mit den skalierten Residuen pro Mehrgitter–Zyklus zur Endtriangulierung. Die grafische Darstellung der Triangulierung erfolgt mit der Routine `meshplot`, die der Schurvektoren mit der Routine `polyplot`.

• meshplot

Zweck:

Grafische Darstellung der Triangulierung

Syntax:

`meshplot(p, e, t)`

Beschreibung:

`meshplot(p, e, t)` stellt die durch `p`, `e` und `t` spezifizierte Triangulierung grafisch dar. Dabei entspricht eine gelbe Randkante der Dirichlet–Randbedingung, eine blaue Randkante der Neumann–Randbedingung.

• mgm

Zweck:

Rekursive Mehrgitter–Routine

Syntax:

`[U, T] = mgm(S, M, U, T, l, my, ny1, ny2, scal)`

Beschreibung:

`[U, T] = mgm(S, M, U, T, l, my, ny1, ny2, scal)` bestimmt rekursiv Grobgitter–Korrekturen zur aktuellen Iterierten auf dem feinsten Gitter. Die Eingabeparameter `S` und `M` sind die restringierten System–Matrizen, `U` die Grobgitter–Korrekturen an die momentanen Schur–Vektoren mit zugehöriger Schurscher oberer Dreiecks–Matrix `T` und `l` die Nummer des nächst gröberen Gitters. Die natürliche Zahl `my` spezifiziert die Art des Mehrgitter–Zyklus:

`my = 1;` V-Zyklus

`my = 2;` W-Zyklus

Die Zahlen `ny1` bzw. `ny2` geben die Anzahl der Vor– bzw. Nachglättungen an, `scal` ist die Skalierungsgröße.

• polybound

Zweck:

Bestimmung der Randbedingungs–Matrix

Syntax:

```
B = polybound(p, e)
```

Beschreibung:

`B = polybound(p, e)` ermittelt die Randbedingungs–Matrix entsprechend der im Eingabeparameter `e` spezifizierten Randbedingungen. Als Freiheitsgrade werden dabei nur innere Punkte und Randpunkte, die nicht auf einer Dirichlet–Kante liegen, angenommen.

• polyplot

Zweck:

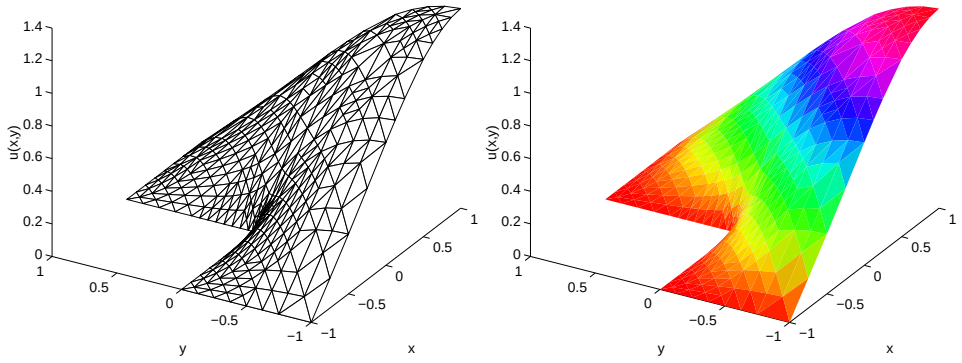
Grafische Darstellung von Funktionen über polygonal begrenzten Gebieten

Syntax:

`polyplot(p, t, u, s)`
`polyplot(p, t, u)`

Beschreibung:

`polyplot(p, t, u, s)` stellt die durch die Funktionswerte im Eingabeparameter `u` bestimmte stückweise lineare Funktion über der durch `p` und `t` spezifizierten Triangulierung dar. Der Eingabeparameter `s` ist eine Zeichenkette, die den `polyplot`-Stil bestimmt. Es sind die Eingaben `s = 'mesh'` und `s = 'surf'` möglich:



`s = 'mesh'` `s = 'surf'`

Die mit `main` bestimmten (u. U. komplexen) Schurvektoren werden am einfachsten durch ihre Betragsquadrate dargestellt:

```
polyplot(p, t, U(:,j).*conj(U(:,j)), s);
```

wobei $1 \leq$ `j` $\leq q$.
`polyplot(p, t, u)` verwendet die Voreinstellung `s = 'mesh'`.

• polyrefinemesh

Zweck:

Verfeinerung einer Triangulierung eines polygonal begrenzten Gebietes

Syntax:

```
[p1, e1, t1, P] = polyrefinemesh(p, e, t, it)
[p1, e1, t1, P, p, e, t, mp] = polyrefinemesh(p, e, t)
```

Beschreibung:

`[p1, e1, t1, P] = polyrefinemesh(p, e, t, it)` gibt eine Verfeinerung der durch `p`, `e` und `t` bestimmten Triangulierung zurück. Dabei werden die der Indexliste `it` entsprechenden Dreiecke durch Kantenhalbierung in vier Teildreiecke unterteilt. Die an diese angrenzenden Dreiecke werden danach mit einer Bisektionsmethode verfeinert, um eventuell „hängende“ Knoten zu beseitigen. Eine genaue Beschreibung des Algorithmus findet sich in [3]. Die Ausgabeparameter `p1`, `e1` und `t1` beschreiben die neue Triangulierung, die Matrix `P` ist die Interpolationsmatrix für den Transfer zwischen alter und neuer Triangulierung. `[p1, e1, t1, P, p, e, t, mp] = polyrefinemesh(p, e, t)` gibt eine uniforme Verfeinerung der durch `p`, `e` und `t` bestimmten Triangulierung zurück, d. h. alle Dreiecke werden in vier Teildreiecke unterteilt. Die Matrix `mp` enthält die Indizes der Kantenmittelpunkte der Dreiecke der alten Triangulierung. Dabei gibt die Komponente `mp(1,j)` den Index des Mittelpunktes der Kante (`t(1,j)`,`t(2,j)`), `mp(2,j)` den der Kante (`t(1,j)`,`t(3,j)`) und `mp(3,j)` den der Kante (`t(2,j)`,`t(3,j)`) an.

• rectmesh

Zweck:

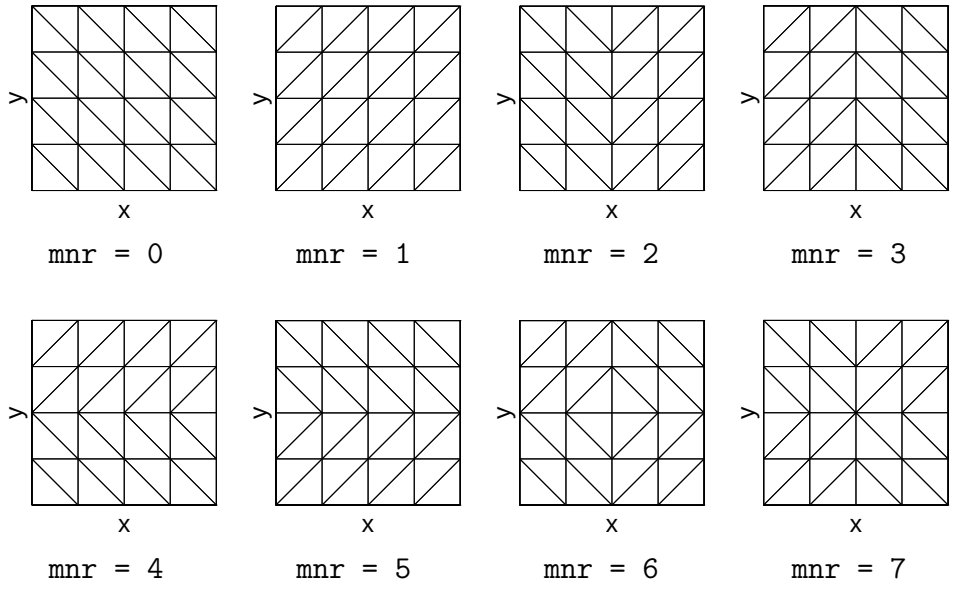
Erzeugung einer Start–Triangulierung für Rechteck–Gebiete

Syntax:

```
[p, e, t] = rectmesh(x, y, mnr)
[p, e, t] = rectmesh(x, y)
```

Beschreibung:

`[p, e, t] = rectmesh(x, y, mnr)` erzeugt für Rechteck–Gebiete eine Triangulierung. Der Vektor `x` gibt dabei die Unterteilung der horizontalen Randkanten, der Vektor `y` die der vertikalen Randkanten an. Der Parameter `mnr` kann Werte zwischen 0 und 7 annehmen:



wobei hier `x = [-5 -2.5 0 2.5 5]` und `y = [-5 -2.5 0 2.5 5]`.
Die Voreinstellung für die Randbedingung ist die Dirichlet–Randbedingung.
`[p, e, t] = rectmesh(x, y)` verwendet als Voreinstellung `mnr = 0`.

- **res**

Zweck:

Bestimmung des Residuums

Syntax:

`r = res(S, M, U, T, scal)`

Beschreibung:

`r = res(S, M, U, T, scal)` gibt die skalierten Residuen gemäß

$$r_j = \frac{\|R_j\|_{M^{-1}}}{\text{scal}}, \quad 1 \leq j \leq q,$$

mit $R = SU - MUT$ zurück. Die Inverse der Massenmatrix M wird dabei durch die Inverse der Diagonalmatrix $\text{diag}(M)$ approximiert. Die Eingabeparameter `S` und `M` sind die System–Matrizen, `U` die aktuellen Schurvektoren mit der zugehörigen Schurschen oberen Dreiecks–Matrix `T`. `scal` ist die aktuelle Skalierungsgröße.

• smoother

Zweck:

Glättungs–Routine für das Mehrgitter–Verfahren

Syntax:

`[U, T, m] = smoother(S, M, U, T, ll, ny, scal)`

Beschreibung:

`[U, T, m] = smoother(S, M, U, T, ll, ny, scal)` führt `ny` Glättungsschritte für das Mehrgitter–Verfahren mit einer cg–artigen Projektionsmethode durch, die ausführlich in [2] beschrieben ist. Die Eingabeparameter `S` und `M` sind die System–Matrizen, `U` die aktuellen Schurvektoren mit der zugehörigen Schurschen oberen Dreiecks–Matrix `T`. Die Variable `ll` kann die Werte 0 oder 1 annehmen:

`ll = 0;` Glättung auf dem feinsten Gitter

`ll = 1;` Glättung auf einem gröberen Gitter

`scal` ist die Skalierungsgröße. Der Funktionswert `m` gibt die Dimension des zuletzt benutzten Unterraumes V gemäß $\dim(V) = q+m$ an, wobei q die Anzahl der gesuchten Eigenwerte ist.

• solvese

Zweck:

Lösung der Sylvester–Gleichung

Syntax:

```
X = solvese(T1, T2, Y)
```

Beschreibung:

`X = solvese(T1, T2, Y)` löst die Sylvester–Gleichung

$$X\,T_1 - T_2\,X = Y,$$

wobei `T1` und `T2` quadratische obere Dreiecksmatrizen gleicher Größe sind. `Y` ist eine beliebige quadratische Matrix passender Größe. Die eindeutige Lösbarkeit dieses Gleichungssystems ist gesichert, solange kein Diagonalelement von `T1` in der Diagonalen von `T2` und umgekehrt enthalten ist.

Literatur

[1] P. Deuflhard, T. Friese, F. Schmidt, R. März und H.–P. Nolting. Effiziente Eigenmodenberechnung für den Entwurf integriert–optischer Chips. In K.–H. Hoffmann, W. Jäger, T. Lohmann und H. Schunck. *Mathematik–Schlüsseltechnologie für die Zukunft*, S. 267–280. Springer–Verlag Berlin Heidelberg New York, 1996.

[2] T. Friese. Eine nichtlineare Mehrgitter–Methode für das Eigenwertproblem linearer, nicht–selbstadjungierter Operatoren. Vorarbeiten zur Dissertation, voraussichtlicher Abschluß: Sommer 1997.

[3] Partial Differential Equation Toolbox User's Guide. The MathWorks, Inc., 1995.