Wissenschaftliches Rechnen I (V2E3)

Wintersemester 2009/2010
Priv.-Doz. Dr. Marc Alexander Schweitzer
Benjamin Berkels, Orestis Vantzos

universitätbonn

Übungsblatt 3.

Abgabe am **Dienstag, 10.11.2009**.

Aufgabe 1. Es seien u und u_h Minimierer von

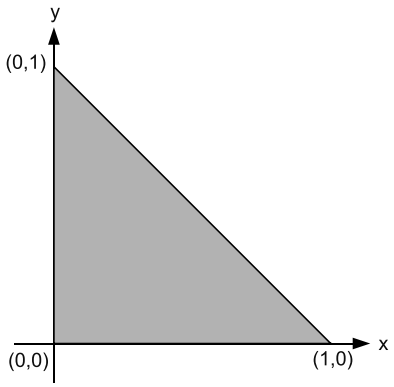
$$J(v) := \frac{1}{2}a(v,v) - \langle l, v\rangle$$

über V beziehungsweise $S_h \subset V$. Zeigen Sie, dass u_h dann auch

$$R(v) := a(u-v, u-v)$$

über S_h minimiert.

Aufgabe 2. Für ein gegebenes Dreieck ist die Menge der kubischen Polynome, deren Einschränkungen auf die Kanten dieses Dreiecks quadratisch sind, ein Raum der Dimension 7. Geben Sie eine Basis dieses Raumes auf dem Einheitsdreieck an.



Das Einheitsdreieck

Programmieraufgabe 1. (Implementierung einer einfachen Triangulierung/Uniforme Verfeinerung)

Die drei wesentlichen Schritte einer Finite Element Berechnung sind:

1. Gittergenerierung bzw adaptive Verfeinerung einer Triangulierung,
2. Diskretisierung der schwachen Formulierung,
3. Lösen des linearen Gleichungssystems.

Die Grundlage bildet also die Gittergenerierung, d.h. wir nehmen für die Diskretisierung an, dass ein zulässiges Gitter/eine zulässige Triangulierung für Ω vorliegt.
Im Rahmen der Diskretisierung müssen wir schwache Formen der Gestalt

$$a(u,v) = \int_\Omega (A\nabla u \nabla v + a_0 uv)\, dx = \int_\Omega fv\, dx - \int_{\Gamma_N} g_N v\, ds$$

für unsere Basisfunktionen φ und gegebene Daten $f \in L^2(\Omega)$ und $g_N \in L^2(\Gamma_N)$ mit $\Gamma_N \subset \partial\Omega$ auswerten können. Dies muss die Datenstruktur für eine Triangulierung einfach ermöglichen.
Hier wollen wir uns nun mit der Frage beschäftigen, wie man eine Triangulierung im Rechner überhaupt darstellen kann und einige Eigenschaften bzw Qualitätsmerkmale von Triangulierungen untersuchen. Die hierfür gewählte Datenstruktur ist sicherlich nur eine von vielen möglichen und unter vielen Gesichtspunkten nicht optimal. Sie wurde gewählt, um eine möglichst einfache Struktur zu haben, die sich in dieser Form sogar in Matlab realisieren läßt. Man beachte, dass wir hierfür alle Indexoperationen bei 1 starten lassen, und dies an zwei Stellen im folgenden explizit vorausgesetzt wird!

Knotenmenge `Nodes` Die grundlegendste Information unserer Triangulierung ist die Knotenmenge. Hier speichern wir zum einen die Koordinaten (x_i, y_i) unserer Knoten für $i = 1, \ldots,$ `Nv` in einem Feld `Nodes` der Dimension `Nv` $\times 2$. Darüber hinaus speichern wir noch die Information, ob ein Knoten frei ist oder dort essentielle Randwerte gefordert werden. Dazu verwenden wir die Felder `FNodePtrs`, `CNodePtrs`, `NodePtrs` die so aufgebaut sind, dass `NodePtrs` die Länge `Nv` = `Nf` + `Nc` hat, wobei `Nc` die Länge von `CNodePtrs` bezeichnet und `Nf` die von `FNodePtrs`. In `FNodePtrs` speichern wir die Positionen im Feld `Nodes` der entsprechenden freien Knoten, analog für `CNodePtrs`. In `NodePtrs` speichern wir nun die Positionen aller Knoten in den entsprechenden Teilfeldern `FNodePtrs` oder `CNodePtrs`. Dazu verwenden wir folgende Unterscheidung: wir speichern die Positionen in `FNodePtrs` als positive Werte und die Positionen in `CNodePtrs` als negative Werte.

Kantenmenge `Edges` Die Menge der Kanten wird in einem Feld der Dimension `Ne` $\times$ 2 mit dem Namen `Edges` gespeichert, wobei eine Kante beschrieben wird über ihre Endpunkte, d.h. zwei Elemente der Knotenmenge bzw. über deren Positionsindices im Feld `Nodes`.

Elementmenge `Elements` Die Menge der Elemente der Triangulierung, d.h. die Dreiecke, werden in einem Feld `Elements` der Dimension `Nt` $\times 3$ gespeichert. Dazu werden pro Dreieck eben die Indices der entsprechenden drei Kanten in dem Feld `Edges` gespeichert, um die äußeren Normalen an das Dreieck leicht bestimmen zu können, müssen wir die Orientierungen der Kante mitspeichern, d.h. wir multiplizieren jeden Index mit ± 1.
Darüber hinaus ist es hilfreich zu jeder Kante direkt auf die Dreiecke zugreifen zu können, die diese Kante gemeinsam haben. Dazu speichern wir diese Information in einem Feld der Dimension `Ne` $\times$ 2, wobei wir eben die Positionen der entsprechenden Dreiecke in `Elements` speichern, falls eine Kante eine Randkante ist, dann gibt es nur ein Dreieck, dass die Kante enthält und wir speichern als zweiten Wert 0 falls die Kante eine Dirichletrandbedingung hat, bzw. den Wert $-a$ wobei a die Position der Kante im Feld `FBndyEdges` ist.

Neumannrand `FBndyEdges` Die Menge der freien Randkanten wird im Feld `FBndyEdges` der Länge `Nb` gespeichert, d.h. die Positionen dieser Kanten im Feld `Edges`.

Randapproximation Da nicht alle Gebiete die wir betrachten wollen polygonal berandet sind, müssen wir den wirklich Rand des Gebiets approximieren. Dies wird natürlich durch die Kanten der Triangulierung erfolgen. Um nun aber bei einer Verfeinerung des Gitters auch eine bessere Randapproximation zu erhalten ist es notwendig explizit auf den wahren Rand zugreifen zu können. In unserer einfachen Implementierung setzen wir voraus, das wir den Rand mittels einer Kurve uniform parametrisiert gegeben haben und zudem auch die Inverse dieser Abbildung vorliegt. Dann können wir zu jeder

Randkante nicht nur Ihren Mittelpunkt bestimmen, sondern auch den Mittelpunkt des Stückes der Randkurve, das von den Eckpunkten der Kante begrenzt wird.
Diese Funktionalität wird von einer Funktion `BndyFcn` bereitgestellt, die zu gegebenen zwei Punkten auf dem Rand $k-1$ Punkte bestimmt, die im Inneren des entsprechenden Kurvensegments liegen.
Da nicht notwendigerweise alle Kanten "gekrümmt" sind, speichern wir in einem Feld `EdgeCFlags` der Länge `Ne` noch die Boolschen Werte 0 oder 1 falls die Kante einen gekrümmten Rand approximiert.
Schließlich speichern wir noch einen globalen Wert `Degree`, der den Polynomgrad unserer Element angibt.

Gittergenerierung durch Verfeinerung

1. Gegeben sein eine Triangulierung `T`.
2. Initalisiere Triangularierung `S`. Kopiere `Nodes`, `NodePtrs`, `FNodePtrs` und `CNodePtrs` von `T` nach `S`, sowie `NodeLevel` und `NodeParents` falls sie in `T` existieren.
3. Iteriere über alle Dreiecke `t` von `T`.
   (a) Iteriere über alle Kanten `e` von Dreieck `t`.
      i. Falls die Kante nicht schon unterteilt wurde: Bestimme den Kantenmittelpunkt und zerlege die Kante in zwei neue Kanten bestimmt durch die Kantenendpunkte und den Kantenmittelpunkt.
   (b) Erzeuge vier Dreiecke `s` die in `t` enthalten sind in `S` durch verbinden der Kantenmittelpunkte. Füge diese Informationen in `Edges`, `Elements`, `EdgeEls`, `EdgeCFlags`, `FBndyEdges`
   (c) Iteriere über alle Kanten `e` von Dreieck `t`.
      i. Bestimme ob der Kantenmittelpunkt von `e` ein freier Knoten oder essentieller Randknoten ist. Füge diese Information in `Nodes`, `NodePtrs`, `FNodePtrs` und `CNodePtrs` von `S` ein.

Man beachte, dass im Fall einer "gekrümmten" Randkante nicht der Mittelpunkte der Kante sondern der entsprechenden Randkurve zu verwenden ist.

Visualisierung Wir stellen der Einfachheit halber zwei Matlab-Routinen zur Verfügung

- ReadTriMesh.m: Mittels dieser Routine kann ein Gitter in obiger Datenstruktur unter Matlab eingelesen werden.
- ShowTriMesh.m: Mittels dieser Routine kann man ein Gitter in obiger Datenstruktur einfach unter Matlab darstellen.

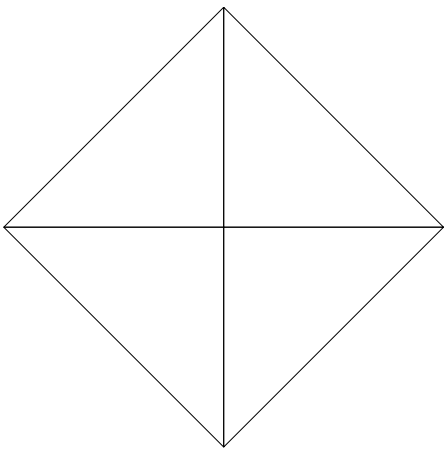
Beispiel

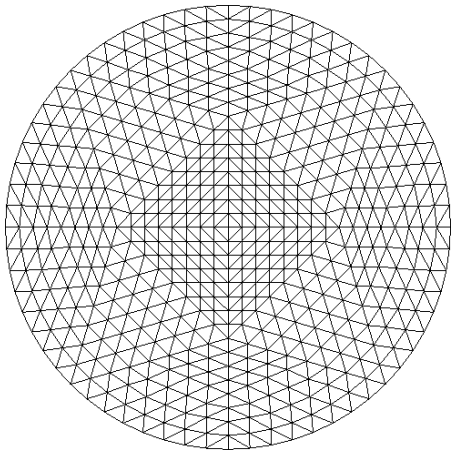
$$\texttt{FBndyEdges} = [], \quad \texttt{Degree} = 1, \quad \texttt{FNodePtrs} = [1]$$

$$\texttt{Nodes} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \texttt{CNodePtrs} = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

$$\texttt{NodePtrs} = \begin{bmatrix} 1 \\ -1 \\ -2 \\ -3 \\ -4 \end{bmatrix}, \quad \texttt{Edges} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 1 \\ 3 & 4 \\ 4 & 1 \\ 4 & 5 \\ 5 & 1 \\ 5 & 2 \end{bmatrix}$$

$$\texttt{EdgeEls} = \begin{bmatrix} 1 & 4 \\ 1 & 0 \\ 1 & 2 \\ 2 & 0 \\ 2 & 3 \\ 3 & 0 \\ 3 & 4 \\ 4 & 0 \end{bmatrix}, \quad \texttt{EdgeCFlags} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \quad \texttt{Elements} = \begin{bmatrix} 1 & 2 & 3 \\ -3 & 4 & 5 \\ -5 & 6 & 7 \\ -7 & 8 & -1 \end{bmatrix}$$





Teilaufgaben

1. Implementieren Sie die obige Datenstruktur und den beschriebenen Verfeinerungsalgorithmus.
2. Verfeinern Sie das Gitter aus obigem Beispiel mindestens drei mal (setze dazu `EdgeCFlags`=0 für alle Kanten).
3. Implementieren Sie die Funktion `BndyFcn` für den Einheitskreis.
4. Verfeinern Sie das Gitter aus obigem Beispiel mindestens drei mal mit der `BndyFcn` des Einheitskreises.

Abgabe der Programmieraufgabe 17.11.2009