Institut für Numerische und Angewandte Mathematik
FB Mathematik und Informatik der Universität Münster
Prof. Dr. Christian Engwer, MSc. Marcel Koch

07.12.2017

Übung zur Vorlesung

Wissenschaftliches Rechnen

WS 2017/18 — Blatt 9

Abgabe: 14.12.2017, 10:00 Uhr, Briefkasten 111
Code zusätzlich per e-mail an `marcel.koch@uni-muenster.de`

Aufgabe 1 (Massenerhaltung im FV-Kontext) (2 Punkte)

Zeigen Sie die „⇒“ Richtung folgender Aussage aus der Vorlesung. Aus der lokalen Massenerhaltung folgt globale Massenerhaltung genau dann, wenn die Flussberechnung von links und von rechts gleich ist, d.h. wenn

$$\nabla u(\gamma)|_{V_1} = \nabla u(\gamma)|_{V_2} \quad \text{für alle } V_1, V_2 \in \mathcal{V} \text{ mit einem gemeinsamen Face } \gamma.$$

Aufgabe 2 (Finite Elemente mit P_1 auf strukturierten Gitter) (4 Punkte)

Betrachten Sie das Poissonproblem in schwacher Formulierung mit Dirichlet Null Randwerten

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{dx} = \int_\Omega 1 \cdot v \mathrm{dx} \quad \forall v \in H_0^1, \tag{1}$$

auf dem Einheitsquadrat $\Omega = (0,1)^2$. Sie sollen nun die globale Steifigkeitsmatrix und rechte Seite einer P_1 Finite Elemente Diskretisierung analytisch berechnen.
Das Gebiet wird zunächst in Rechtecke mit Kantenlänge $h = 1/N$ zerlegt, wobei $N+1$ die Anzahl an Knoten in x- und y-Richtung ist. Die Rechtecke werden weiter entlang der $y = x$ Diagonalen in zwei Dreiecke unterteilt. Dies liefert eine strukturierte Triangulierung von Ω mit $(N+1)^2$ Freiheitsgraden.

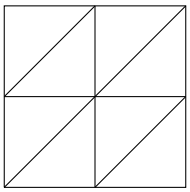

Abbildung 1: Triangulierungen mit $N = 2$

Berechnen Sie explizit die Einträge der Steifigkeitsmatrix un der rechnten Seite im inneren des Gebiets, d.h.

$$A = \left(a(\varphi_j, \varphi_i)\right), \quad b = (1, \varphi_i)_{L^2} \text{ für } i, j = 1, \ldots, N^2,$$

wobei φ_i, $i = 0, \ldots, N^2$ die nodale Basis des Ansatzraums ist, d.h. $\varphi_i(x_j) = \delta_{ij}$. Setzen Sie die Einträge für Randknoten so, dass die Matrix positiv definit bleibt.

In den kommenden zwei Wochen werden Sie in den Programmieraufgaben ein Finite Elemente Programm schreiben, das in der Lage ist die Poissongleichung

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= g \quad \text{auf } \partial\Omega \end{aligned}$$

für verschiedene rechte Seiten, Randwerte und Gebiete zu lösen. Auf diesem Aufgabenblatt werden Sie ein simples unstrukturiertes Dreiecksgitter implementieren, so wie sich mit dem Gebrauch von dünnbesetzten Matrizen vertraut machen.
Das nächste Aufgabenblatt behandelt die Berechnung der lokalen Steifigkeitsmatrizen und der lokalen rechten Seite und deren Speicherung in den globalen Datenstrukturen. Zusätzlich werden die Dirichlet-Knoten behandelt.

Aufgabe 3 (Implementierung eines Dreiecksgitters) (5 Punkte)

Es gibt verschiedenste Möglichkeiten eine Triangulierung in C++ zu implementieren. Wir beschränken uns hier auf die einfachste. Dazu benötigen wir nur die Koordinaten der Knoten im Gitter und für jedes Dreieck die Indizes der Knoten die das Dreieck erzeugen. Insgesamt lässt sich diese Gitterformat in Textform wie folgt darstellen:

```
NNODES number_of_nodes //=N
x_0 y_0
x_1 y_1
.
.
.
x_{N-1} y_{N-1}
NELEMENTS number_of_elements //=E
i_0 j_0 k_0
i_1 j_1 k_1
.
.
.
i_{E-1} j_{E-1} k_{E-1}
```

Die Triangulierung in Abbildung 1 entrspräche folgender Darstellung:

```
NNODES 9
0 0
0.5 0
1 0
0 0.5
0.5 0.5
1 0.5
0 1
0.5 1
1 1
NELEMENTS 8
0 1 4
```

```
  1 2 5
14 3 0 4
  4 1 5
16 3 4 7
  4 5 8
18 6 3 7
  7 4 8
```

(a) Implementieren Sie eine Klasse `P1Triangle` die ein Gitterelement darstellt. Die Klasse soll mindestens folgende Methoden zur Verfügung stellen:

- `std::size_t local2global(std::size_t local)` gibt zu einem lokalen Knoten die globale Nummerierung zurück,
- `std::size_t size()` gibt die Anzahl an lokalen Knoten zurück,
- `double jacDet()` gibt die für ein Dreieck E die Determinate der Jacobimatrix der Transformation $T : \hat{E} \mapsto E$ zurück,
- `Matrix_Type jacInvT()` gibt die transponierte Inverse Jacobimatrix der oben genannten Transformation T zurück.

(b) Implementieren Sie eine Klasse `P1Grid` die die Knoten und Elemente im Gitter verwaltet. Die Klasse soll mindestens folgende Attribute und Methoden zur Verfügung stellen:

- jeweils einen Container für die Knoten im Gitter und die Elemente im Gitter,
- einen Konstruktor der einen `string` übernimmt und damit eine Datei im obigen Format einließt und daraus das Gitter erstellt,
- `std::size_t size_dofs()` gibt die Anzahl an Freiheitsgraden zurück,
- `T elements()` liefert einen Lesezugriff auf die Elemente im Gitter, den Typen passen sie an Ihre gewählten Container an.

(c) Testen Sie Ihre Implementierung, in dem Sie die auf die auf der Homepage zur verfügung gestellten Gitter einlesen und deren Fläche berechnen.

Die Knoten des Gitters können Sie durch den bereits für das N-Körper Problem verwendeten Typen realisieren. Dieser ist auch auf der Homepage verfügbar.

Aufgabe 4 (Verwendung von Sparse-Matrizen) (5 Punkte)

Die bei der diskretisierung des Poissonproblems (1) entstehenden Matrizen haben nur sehr wenige Einträge ungleich Null. Bei einem strukturierten Gitter wie in Aufgabe 2 beschrieben sind nur maximal sieben Einträge einer Zeile ungleich Null. Es macht daher Sinn sich nur die nicht Null-Einträge abzuspeichern. Dies kann z.B. durch Bandmatrizen realisiert werden. Für unstrukturierte Gitter wird ein allgemeineres Format benötigt, oft greift man dabei auf das *compressed row storage (CRS)* oder das *compressed column storage (CCS)* Format zurück. Bei diesen Formaten speichert man folgende Informationen:

- `values`: ein Array mit den Werten der nicht Null-Einträge,

- `row_indices`: ein Array mit den Zeilenindzes der nicht Null-Einträge, d.h. `row_indices[i]` gibt an in welcher Zeile sich der Wert `values[i]` befindet,
- `col_start`: ein Array das für jede Spalte der Matrix den Index ihres ersten nicht Null-Eintrags, bezogen auf die `values` und `row_indices` Arrays, enthält. Das Array einthält einen zusätzlichen Eintrag, die Anzahl aller nich Null-Einträge.

Um das CCS Format zu verdeutlichen, betrachten wir folgendes Beispiel:

$$M = \begin{pmatrix} 0 & 3 & 0 & 0 & 0 \\ 22 & 0 & 0 & 0 & 17 \\ 7 & 5 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 8 \end{pmatrix}$$

`values`	22	7	3	5	14	1	17	8
`row_indices`	1	2	0	2	4	2	1	4
`col_start`	0	2	4	5	6	8		

Ihre Aufgabe ist es nun die in Aufgabe 2 aufgestellte Matrix als CCS Matrix zu implementieren. Dazu werden Sie das CCS Format der linearen Algebra Bibliothek *Eigen*[1] benutzen. Eigen ist eine Header Bibliothekt, d.h. Sie müssen Eigen nicht auf Ihrem System installieren, es reicht wenn Sie dem Kompiler den Pfad zu den Header Dateien angeben mit der `-I /path/to/eigen/include` Option. Sie können die dazu die Makefile von der Homepage benutzen. Um die CCS Matrix mit den aus Aufgabe 2 bekannten Werten zufüllen, verwenden Sie die `setFromTriplets` Funktion der CCS Matrix. An diese Funktion übergeben Sie den `begin()` und `end()` Iterator eines Vektors von `Eigen::Triplets`. Diese Triplets bestehen aus drei Komponenten, den Zeilenindex, den Spaltenindex und den Wert eines nicht Null-Eintrags.

- In der Aufgabe 2 habe Sie nur Dirichlet Null Randwerte betrachtet. Hier betrachten wir das Problem

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{dx} = \int_\Omega 1 \cdot v \mathrm{dx} \quad \forall v \in H_0^1,$$
$$u|_{\partial\Omega} = g,$$

  mit $g(x) = \frac{1}{4}|x|^2$. Die Funktion g entspricht in diesem Fall der exakten Lösung. Stellen Sie die globale Steifigkeitsmatrix A und die globale rechte Seite b auf. Passen Sie die Randwerte in b entsprechend der Funktion g an.

- Lösen Sie das System $Ax = b$ mit einem in Eigen verfügbaren Löser[2] ihrer Wahl, für verschiedene Schrittweiten h. Stellen Sie den maximalen Fehler in abhängigkeit von h dar.

[1] http://eigen.tuxfamily.org
[2] http://eigen.tuxfamily.org/dox/group__TopicSparseSystems.html