

## Programmierblatt 2.

Bearbeiten bis: **Sonntag, 03.11.2019**

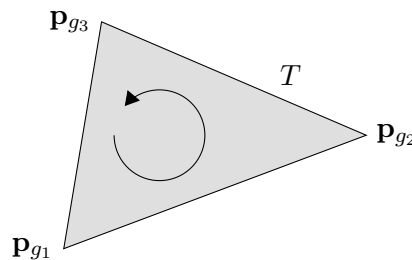
Nachdem auf dem ersten Blatt die Darstellung und Implementierung von Dreiecksnetzen erläutert worden ist, ist dieses zweite Blatt dem numerischen Lösen des Poisson-Problems

$$\begin{aligned} -\Delta u(\mathbf{x}) &= f(\mathbf{x}) && \text{für } \mathbf{x} \in \Omega, \\ u(\mathbf{x}) &= 0 && \text{für } \mathbf{x} \in \Gamma_D, \\ \langle \nabla u(\mathbf{x}), \mathbf{n}(\mathbf{x}) \rangle &= 0 && \text{für } \mathbf{x} \in \Gamma_N, \end{aligned}$$

auf dem durch ein Dreiecksnetz  $\mathcal{T}$  gegebenen Gebiet  $\Omega$  gewidmet. Dabei geschieht die Partition des Randes in  $\partial\Omega = \Gamma_D \cup \Gamma_N$  gemäss den Randbedingungsflags aus der Dreiecksnetzbeschreibung.

Die numerische Umsetzung soll dabei mit linearen Finiten Elementen geschehen. Hierzu müssen grundsätzlich die Steifigkeitsmatrix und die Massenmatrix assembliert und die nodale Interpolation einer Funktion in den Raum der linearen Finiten Elementen berechnet werden können. Da das resultierende Gleichungssystem symmetrisch positiv definit ist, soll als Löser das CG-Verfahren zum Zug kommen.

### Assemblierung der FEM-Matrizen



Die Assemblierung der Massen- und Steifigkeitsmatrix erfolgt mittels der lokalen Elementmatrizen. Diese sind für ein gegebenes Dreieck  $T \in \mathcal{T}$  gegeben durch

$$\mathbf{M}_T = \left[ m_{r,s}^T \right]_{r,s=1}^3 \quad \text{und} \quad \mathbf{A}_T = \left[ a_{r,s}^T \right]_{r,s=1}^3,$$

wobei

$$m_{r,s}^T := m_T(\varphi_{g_s}, \varphi_{g_r}) := \int_T \varphi_{g_s}(\mathbf{x}) \varphi_{g_r}(\mathbf{x}) \, d\mathbf{x} = \int_T \lambda_s(\mathbf{x}) \lambda_r(\mathbf{x}) \, d\mathbf{x}$$

und

$$a_{r,s}^T := a_T(\varphi_{g_s}, \varphi_{g_r}) := \int_T \langle \nabla \varphi_{g_s}(\mathbf{x}), \nabla \varphi_{g_r}(\mathbf{x}) \rangle \, d\mathbf{x} = \int_T \langle \nabla \lambda_s(\mathbf{x}), \nabla \lambda_r(\mathbf{x}) \rangle \, d\mathbf{x}$$

sind (siehe Übungsblatt 5, Aufgaben 1 und 2). Dabei bezeichnen wir hier mit  $g_r$  den Index des  $r$ -ten Eckpunktes des Dreiecks  $T$  in der Punktliste; diese Information ist genau in der Elementliste  $\mathbf{F}$  gespeichert.

Die Beziehung zwischen der Massen- und Steifigkeitsmatrix

$$\mathbf{M} = [m_{i,j}]_{i,j=1}^{n_p} \quad \text{und} \quad \mathbf{A} = [a_{i,j}]_{i,j=1}^{n_p}$$

und den lokalen Elementmatrizen ist dann durch

$$m_{i,j} := m(\varphi_j, \varphi_i) = \sum_{T \in \mathcal{T}} m_T(\varphi_j, \varphi_i) \quad \text{und} \quad a_{i,j} := a(\varphi_j, \varphi_i) = \sum_{T \in \mathcal{T}} a_T(\varphi_j, \varphi_i)$$

gegeben. Dabei sind

$$m_T(\varphi_j, \varphi_i) = \begin{cases} m_{r,s}^T, & \text{falls es } r, s \in \{1, 2, 3\} \text{ mit } g_r = i \text{ und } g_s = j \text{ gibt,} \\ 0, & \text{sonst,} \end{cases}$$

beziehungsweise analog

$$a_T(\varphi_j, \varphi_i) = \begin{cases} a_{r,s}^T, & \text{falls es } r, s \in \{1, 2, 3\} \text{ mit } g_r = i \text{ und } g_s = j \text{ gibt,} \\ 0, & \text{sonst.} \end{cases}$$

Zum Abspeichern der Massen- und Steifigkeitsmatrix soll das **MATLAB sparse**-Format verwendet werden. Für die Assemblierung einer Matrix im **sparse**-Format in **MATLAB** wird der **sparse**-Befehl<sup>1</sup> benutzt, womit sich dann der folgende Algorithmus ergibt:

---

**Algorithmus** Assemblierung der FEM-Matrizen

---

*Input:* Elementliste **F**, Punktliste **P**

*Output:* FEM-Matrizen **M** und **A** im **sparse**-Format

alloziere **V<sub>m</sub>**, **V<sub>a</sub>**, **I** und **J**

**for**  $k = 1 : \text{size}(\mathbf{F}, 2)$  **do**

    bilde die lokalen Elementmatrizen **M<sub>T</sub>** und **A<sub>T</sub>** des  $k$ -ten Dreiecks  $T$   
    gegeben durch **F**(:,  $k$ ) und setze

$$\mathbf{V}_m(:, 3k - 2 : 3k) := \mathbf{M}_T \quad \text{und} \quad \mathbf{V}_a(:, 3k - 2 : 3k) := \mathbf{A}_T$$

    setze die zugehörigen, globalen Indizes in **I** und **J**

$$\mathbf{I}(:, 3k - 2 : 3k) := \begin{bmatrix} \mathbf{F}(1, k) & \mathbf{F}(1, k) & \mathbf{F}(1, k) \\ \mathbf{F}(2, k) & \mathbf{F}(2, k) & \mathbf{F}(2, k) \\ \mathbf{F}(3, k) & \mathbf{F}(3, k) & \mathbf{F}(3, k) \end{bmatrix}$$

$$\mathbf{J}(:, 3k - 2 : 3k) := \begin{bmatrix} \mathbf{F}(1, k) & \mathbf{F}(2, k) & \mathbf{F}(3, k) \\ \mathbf{F}(1, k) & \mathbf{F}(2, k) & \mathbf{F}(3, k) \\ \mathbf{F}(1, k) & \mathbf{F}(2, k) & \mathbf{F}(3, k) \end{bmatrix}$$

**end for**

    benutze den **MATLAB sparse**-Befehl um mit **I**(:), **J**(:) und **V<sub>m</sub>**(:) die Matrix **M**  
    und mit **I**(:), **J**(:) und **V<sub>a</sub>**(:) die Matrix **A** zu assemblieren

---

## Aufgabe 1.

Schreiben Sie eine Funktion

```
function [M, A] = assemble_mass_and_stiffness(mesh),
```

welche die Massen- und Steifigkeitsmatrix im **sparse**-Format für das übergebene Dreiecksnetz **mesh** berechnet.

## Berechnung der nodalen Interpolation einer Funktion

Um die rechte Seite der diskretisierten Gleichung aber auch den Fehler zwischen der numerischen Lösung und einer analytisch gegebenen Lösung approximativ berechnen zu

<sup>1</sup>Lesen Sie die **MATLAB**-Hilfe, um zu verstehen was der Befehl **A = sparse(I, J, V)** ergibt.

können, ist es hilfreich die nodale Approximation einer Funktion  $g: \Omega \rightarrow \mathbb{R}$  bestimmen zu können. Die nodale Approximation von  $g$  ist offensichtlich durch

$$g(\mathbf{x}) \approx \sum_{j=1}^{n_p} g(\mathbf{p}_j) \varphi_j(\mathbf{x})$$

gegeben. Die Gewichte der nodalen Approximation von  $g$  sind daher einfach  $g(\mathbf{p}_j)$ , womit die nodale Approximation von  $g$  also durch den Koeffizientenvektor  $\mathbf{g} = [g(\mathbf{p}_j)]_{j=1}^{n_p}$  beschrieben ist.

### Aufgabe 2.

Schreiben Sie eine Funktion

```
function g = compute_nodal_interpolation(gf, mesh),
```

welche den Koeffizientenvektor  $\mathbf{g}$  der nodalen Approximation von  $g$  für das übergebene Dreiecksnetz `mesh` berechnet. Die Funktion  $g$  wird dabei als `function handle gf` übergeben.

### Aufgabe 3.

Schreiben Sie eine Funktion

```
function mesh_function_plot(g, mesh),
```

welche die durch den Koeffizientenvektor  $\mathbf{g}$  und dem übergebene Dreiecksnetz `mesh` definierte Funktion visualisiert.

## Diskretisierung der rechten Seite

Zur Diskretisierung der rechten Seite müssen die Integrale

$$\int_{\Omega} f(\mathbf{x}) \varphi_i(\mathbf{x}) \, d\mathbf{x}$$

für alle Basisfunktionen  $\varphi_i$  gebildet werden. Bezeichne  $\mathbf{f} = [f_j]_{j=1}^{n_p}$  der zu der nodalen Approximation von  $f$  zugehörige Koeffizientenvektor, so gilt

$$\int_{\Omega} f(\mathbf{x}) \varphi_i(\mathbf{x}) \, d\mathbf{x} \approx \int_{\Omega} \sum_{j=1}^{n_p} f_j \varphi_j(\mathbf{x}) \varphi_i(\mathbf{x}) \, d\mathbf{x} = \sum_{j=1}^{n_p} \int_{\Omega} \varphi_j(\mathbf{x}) \varphi_i(\mathbf{x}) \, d\mathbf{x} f_j = [\mathbf{Mf}]_i.$$

Die rechte Seite der diskretisierten Gleichung ist daher einfach durch  $\mathbf{Mf}$  gegeben.

## CG-Verfahren

Das CG-Verfahren ist ein iteratives Verfahren zur Lösung grosser linearer Gleichungssysteme  $\mathbf{A}\mathbf{u} = \mathbf{b}$  mit symmetrischer und positive definiten Systemmatrix  $\mathbf{A}$ . Es lautet:

---

**Algorithmus** CG-Verfahren

---

*Input:* Systemmatrix  $\mathbf{A}$ , rechte Seite  $\mathbf{b}$ , Fehler-Toleranz  $\varepsilon_{\text{tol}}$

*Output:* Lösungsvektor  $\mathbf{u}$

```
 $k = 0$   
 $\mathbf{d}_0 = \mathbf{r}_0 = \mathbf{b}$   
 $\varepsilon_0 = \|\mathbf{r}_0\|_2^2$   
while  $\sqrt{\varepsilon_k} > \varepsilon_{\text{tol}}$  do  
     $\mathbf{z} = \mathbf{A}\mathbf{d}_k$   
     $\alpha_k = \varepsilon_k / (\mathbf{d}_k^\top \mathbf{z})$   
     $\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha_k \mathbf{d}_k$   
     $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{z}$   
     $\varepsilon_{k+1} = \|\mathbf{r}_{k+1}\|_2^2$   
     $\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \varepsilon_{k+1} \mathbf{d}_k / \varepsilon_k$   
     $k = k + 1$   
end while
```

---

**Aufgabe 4.**

Implementieren Sie das CG-Verfahren als Funktion

```
function [u, iter] = cg_solver(A, b, eps_tol),
```

welche neben der Lösung  $\mathbf{u}$  auch die Anzahl benötigter Iterationen in der Variablen `iter` zurückgibt.

**Handhabung der Randbedingungen**

Mit dem bis hier beschriebenen Vorgehen kann man nun direkt die diskretisierte Gleichung für das Poisson-Problem als

$$\mathbf{A}\mathbf{u} = \mathbf{b} \quad \text{mit} \quad \mathbf{b} := \mathbf{M}\mathbf{f}$$

aufstellen, wenn man annimmt, dass man keine Dirichlet-Randbedingungen hätte, d.h. wenn  $\Gamma_D = \emptyset$  gelten würde.

Wegen den vorhandene Dirichlet-Randbedingungen müssen aber die Unbekannten in  $\mathbf{u}$  und  $\mathbf{A}$  sowie die Gleichungen in  $\mathbf{A}$  und  $\mathbf{b}$  gestrichen werden, die zu Eckpunkten mit Dirichlet-Randbedingung assoziiert sind. Mit dem Ausdruck

```
idxs = mesh.B ~= 0;
```

erhält man in **MATLAB** einen logischen Vektor der genau an denen Stellen ein `'false'` enthält die den Eckpunkten mit Dirichlet-Randbedingung entsprechen. Folglich lautet dann das zu lösende Gleichungssystem mit logischer Indizierung

$$\mathbf{A}(\text{idxs}, \text{idxs})\mathbf{u}(\text{idxs}) = \mathbf{b}(\text{idxs}),$$

wobei man vorher den Vektor  $\mathbf{u}$  mit dem Nullvektor initialisieren kann, damit dann die Werte bei den Eckpunkten mit Dirichlet-Randbedingung auch richtig sind.

**Aufgabe 5.**

Schreiben Sie eine Funktion

```
function [u, iter] = solve_poisson_problem(ff, mesh, eps_tol),
```

welche die numerische Lösung des Poisson-Problems für die als `function handle`  $ff$  übergebene Funktion  $f$  auf dem übergebene Dreiecksnetz `mesh` löst. Verwenden Sie zur Lösung des linearen Gleichungssystems Ihres CG-Verfahren mit der Fehler-Toleranz `eps_tol`.

### Aufgabe 6.

Schreiben Sie ein MATLAB-Skript, welches das Poisson-Problem mit der rechten Seite,

$$f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \frac{13}{4}\pi^2 \sin(\pi x) \cos\left(\frac{3}{2}\pi y\right),$$

auf den Dreiecksnetzen löst, die Sie durch ein- bis sechsmaligen Verfeinern des durch `mesh_generation_square` generierten Dreiecksnetz erhalten. Wählen Sie dafür `eps_tol = 10-8` Für jede Verfeinerungsstufe berechnen Sie den  $L^2$ - und  $H^1$ -Fehler der numerischen Lösung  $u$  gegenüber der nodalen Approximation der exakten Lösung

$$u\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \sin(\pi x) \cos\left(\frac{3}{2}\pi y\right).$$

Anschliessend erzeugen Sie einen `semilogy`-Plot in dem Sie die Anzahl Verfeinerungen gegen den Fehler auftragen. Ebenso erzeugen Sie einen zweiten `semilogy`-Plot in dem Sie die Anzahl Verfeinerungen gegen die benötigte Anzahl Iterationen im CG-Verfahren auftragen. Was fällt auf?

Hinweis. Für eine durch einen Koeffizientenvektor  $\mathbf{v}$  gegebene Finite-Element-Funktion  $v$  gilt:

$$\|v\|_{L^2(\Omega)} = \sqrt{\mathbf{v}^T \mathbf{M} \mathbf{v}} \quad \text{und} \quad \|v\|_{H^1(\Omega)} = \sqrt{\mathbf{v}^T \mathbf{A} \mathbf{v} + \mathbf{v}^T \mathbf{M} \mathbf{v}}.$$

### Aufgabe 7.

Schreiben Sie ein MATLAB-Skript, welches das Poisson-Problem mit rechter Seite  $f(\mathbf{x}) = 100$  auf dem dreifach oder vierfach verfeinerten "fem"-Dreiecksnetz löst und visualisiert.

