Programmierblatt 2.

Abgabewoche: **28.10–1.11.2024**

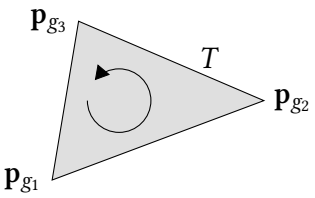
Nachdem auf dem ersten Blatt die Darstellung und Implementierung von Dreiecksnetzen erläutert worden ist, ist dieses zweite Blatt dem numerischen Lösen des Poisson-Problems

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

auf dem durch ein Dreiecksnetz $\mathcal{T}$ gegebenen Gebiet Ω 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 Finite Elemente berechnet werden können. Da das resultierende Gleichungssystem symmetrisch positiv definit ist, soll als Löser das CG-Verfahren mit und ohne diagonaler Präkonditionierung 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

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

und

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

sind (siehe Übungsblatt 4, Aufgaben 1 und 2). Dabei bezeichnen wir hier mit g_r den Index des r-ten Eckpunktes des Dreiecks T in der Punkteliste; diese Information ist genau in der Elementliste $\mathbf{F}$ gespeichert. Setzen wir

$$\mathbf{a} = \mathbf{p}_{g_2} - \mathbf{p}_{g_1}, \quad \mathbf{b} = \mathbf{p}_{g_3} - \mathbf{p}_{g_2}, \quad \mathbf{c} = \mathbf{p}_{g_1} - \mathbf{p}_{g_3} \quad \text{und} \quad d = \big| \det\big([\mathbf{a}, \mathbf{c}]\big) \big|,$$

so sind

$$\mathbf{M}_T = \frac{d}{24} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \quad \text{und} \quad \mathbf{A}_T = \frac{1}{2d} \begin{bmatrix} -\mathbf{b}^\top \mathbf{c} - \mathbf{a}^\top \mathbf{b} & \mathbf{b}^\top \mathbf{c} & \mathbf{a}^\top \mathbf{b} \\ \mathbf{b}^\top \mathbf{c} & -\mathbf{a}^\top \mathbf{c} - \mathbf{b}^\top \mathbf{c} & \mathbf{a}^\top \mathbf{c} \\ \mathbf{a}^\top \mathbf{b} & \mathbf{a}^\top \mathbf{c} & -\mathbf{a}^\top \mathbf{b} - \mathbf{a}^\top \mathbf{c} \end{bmatrix}.$$

Die Beziehung zwischen der Massen- und Steifigkeitsmatrix

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

und den lokalen Elementmatrizen ist dann durch

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

gegeben. Dabei sind

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

beziehungsweise analog

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

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[1] benutzt, womit sich dann der folgende Algorithmus ergibt:

Algorithmus Assemblierung der FEM-Matrizen

Input: Elementliste $\mathbf{F}$, Punktliste $\mathbf{P}$
Output: FEM-Matrizen $\mathbf{M}$ und $\mathbf{A}$ im sparse-Format

alloziere $\mathbf{V}_m$, $\mathbf{V}_a$, $\mathbf{I}$ und $\mathbf{J}$
for $k = 1 : \text{size}(\mathbf{F}, 2)$ **do**
bilde die lokalen Elementmatrizen $\mathbf{M}_T$ und $\mathbf{A}_T$ des k-ten Dreiecks T
gegeben durch $\mathbf{F}(:, k)$ 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 $\mathbf{I}$ und $\mathbf{J}$

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

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

end for
benutze den `MATLAB` `sparse`-Befehl um mit $\mathbf{I}(:)$, $\mathbf{J}(:)$ und $\mathbf{V}_m(:)$ die Matrix $\mathbf{M}$ und mit $\mathbf{I}(:)$, $\mathbf{J}(:)$ und $\mathbf{V}_a(:)$ die Matrix $\mathbf{A}$ 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.

[1] Lesen Sie die `MATLAB`-Hilfe, um zu verstehen, was der Befehl `A = sparse(I, J, V)` ergibt.

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

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

gegeben. Die Gewichte der nodalen Approximation von g sind daher einfach $g(\mathbf{p}_j)$, weshalb die nodale Approximation von g durch den Koeffizientenvektor $\mathbf{g} = \left[g(\mathbf{p}_j)\right]_{j=1}^{n_p}$ 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, welche den Auswertungspunkt $\mathbf{x}$ als einen Vektor nimmt. (Sprich: `gf = @(x) ...` und nicht `gf = @(x1, x2) ...`)

Aufgabe 3.

Schreiben Sie eine Funktion

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

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

Hinweis. *Der MATLAB-Befehl* `patch` *ist hilfreich um Dreicksnetze zu plotten. Am Ende dieses Dokuments sehen Sie, wie eine mögliche Visulisation aussehen kann.*

Diskretisierung der rechten Seite

Zur Diskretisierung der rechten Seite müssen die Integrale

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

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

$$\int_\Omega f(\mathbf{x})\phi_i(\mathbf{x})\,\mathrm{d}\mathbf{x} \approx \int_\Omega \sum_{j=1}^{n_p} f_j\phi_j(\mathbf{x})\phi_i(\mathbf{x})\,\mathrm{d}\mathbf{x} = \sum_{j=1}^{n_p} \int_\Omega \phi_j(\mathbf{x})\phi_i(\mathbf{x})\,\mathrm{d}\mathbf{x} f_j = [\mathbf{M}\mathbf{f}]_i.$$

Man kann also die rechte Seite der diskretisierten Gleichung einfach als $\mathbf{M}\mathbf{f}$ wählen.

CG-Verfahren

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

Algorithmus CG-Verfahren

Input: Systemmatrix $\mathbf{A}$, rechte Seite $\mathbf{b}$, Fehler-Toleranz ε_{tol}
Output: Lösungsvektor $\mathbf{u}$

Setze $\mathbf{u}_0 := \mathbf{0}$, $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{u}_0$, $\mathbf{p}_{-1} := \mathbf{0}$ und $s_0 := 1$ und $k := 0$
while $\sqrt{\mathbf{r}_k^\top \mathbf{r}_k} > \varepsilon_{\text{tol}}$ **do**
  Setze $t_k := \mathbf{r}_k^\top \mathbf{r}_k$
  Setze $\mathbf{p}_k := \mathbf{r}_k + (t_k/s_k)\mathbf{p}_{k-1}$
  Setze $\mathbf{q}_k := \mathbf{A}\mathbf{p}_k$
  Setze $\alpha_k = t_k/(\mathbf{p}_k^\top \mathbf{q})$
  Setze $\mathbf{u}_{k+1} := \mathbf{u}_k + \alpha_k \mathbf{p}_k$
  Setze $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{q}_k$
  Setze $s_{k+1} := t_k$
  Setze $k := k + 1$
end while

Da die Anzahl der benötigten Iterationen wesentlich von der Konditionszahl der Matrix $\mathbf{A}$ abhängt, kann man versuchen, ein äquivalentes lineares Gleichungssystem $\mathbf{C}^{-1}\mathbf{Au} = \mathbf{C}^{-1}\mathbf{b}$ für eine symmetrische und positiv definite Matrix $\mathbf{C}$ zu lösen. Dabei erhofft man sich, dass $\text{cond}(\mathbf{C}^{-1}\mathbf{A}) < \text{cond}(\mathbf{A})$ gilt. Häufig benötigt man bereits weniger Iterationen, wenn man als Vorkonditionierer die Diagonale von $\mathbf{A}$ wählt, insbesondere kann die Diagonalmatrix auch schnell invertiert werden. Dies führt auf den folgenden Algorithmus:

Algorithmus Diagonal vorkonditioniertes CG-Verfahren

Input: Systemmatrix $\mathbf{A}$, rechte Seite $\mathbf{b}$, Fehler-Toleranz ε_{tol}
Output: Lösungsvektor $\mathbf{u}$

Setze $\mathbf{C}^{-1} := (\text{diag}\,\mathbf{A})^{-1}$, $\mathbf{u}_0 := \mathbf{0}$, $\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{u}_0$, $\mathbf{p}_{-1} := \mathbf{0}$ und $s_0 := 1$ und $k := 0$
while $\sqrt{\mathbf{r}_k^\top \mathbf{r}_k} > \varepsilon_{\text{tol}}$ **do**
  Setze $\mathbf{z}_k := \mathbf{C}^{-1}\mathbf{r}_k$
  Setze $t_k := \mathbf{r}_k^\top \mathbf{z}_k$
  Setze $\mathbf{p}_k := \mathbf{z}_k + (t_k/s_k)\mathbf{p}_{k-1}$
  Setze $\mathbf{q}_k := \mathbf{A}\mathbf{p}_k$
  Setze $\alpha_k = t_k/(\mathbf{p}_k^\top \mathbf{q})$
  Setze $\mathbf{u}_{k+1} := \mathbf{u}_k + \alpha_k \mathbf{p}_k$
  Setze $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{q}_k$
  Setze $s_{k+1} := t_k$
  Setze $k := k + 1$
end while

Aufgabe 4.

Implementieren Sie das CG-Verfahren und das diagonal vorkonditionierte CG-Verfahren als Funktionen

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

welche neben der Lösung u auch die Anzahl benötigter Iterationen in der Variablen `iter` zurückgeben.

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 assoziert sind. Mit dem Ausdruck

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

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

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

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 Funktionen

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

welche die numerische Lösung des Poisson-Problems für die als `function handle ff` übergebene Funktion f auf dem übergebenen Dreiecksnetz `mesh` berechnen. Verwenden Sie zur Lösung des linearen Gleichungssystems das CG-Verfahren respektive das diagonal vorkonditionierte CG-Verfahren mit der Fehler-Toleranz `eps_tol`.

Aufgabe 6.

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

$$f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \frac{25}{4}\pi^2 \cos(2\pi x)\cos\left(\frac{3}{2}\pi y\right)$$

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

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

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 Anzal Iterationen im CG-Verfahren auftragen. Lösen Sie die Aufgabe mit dem CG-Verfahren und dem diagonal vorkonditionierten CG-Verfahren. Was fällt auf?

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

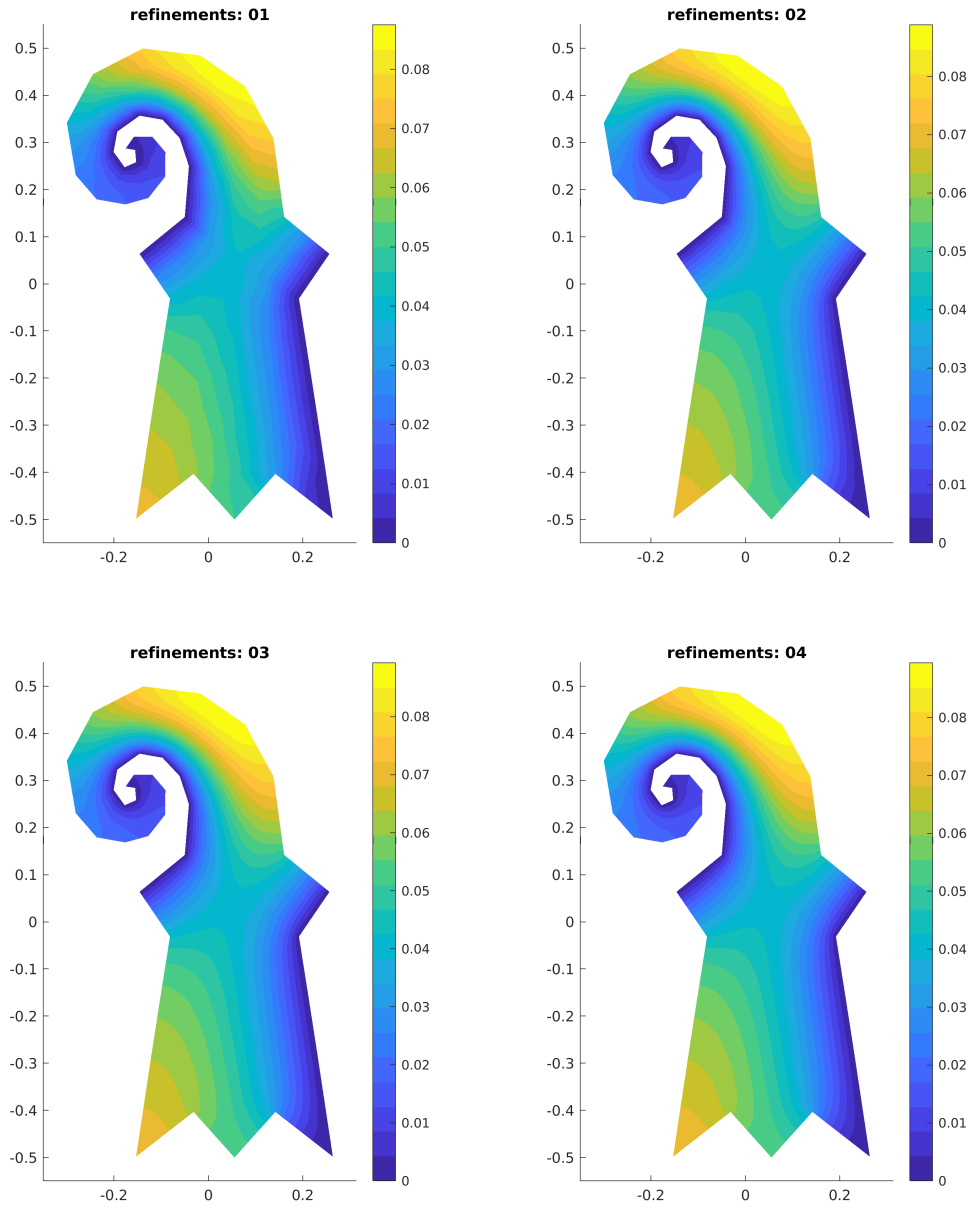
$$\|v\|_{L^2(\Omega)} = \sqrt{\mathbf{v}^\mathsf{T}\mathbf{M}\mathbf{v}} \quad \textit{und} \quad \|v\|_{H^1(\Omega)} = \sqrt{\mathbf{v}^\mathsf{T}\mathbf{A}\mathbf{v} + \mathbf{v}^\mathsf{T}\mathbf{M}\mathbf{v}}.$$

Aufgabe 7.

Schreiben Sie ein `MATLAB`-Skript, welches das Poisson-Problem mit rechter Seite

$$f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = 1 + 6\left(y + \frac{1}{2}\right)^3$$

auf dem einfach bis dreifach oder vierfach verfeinerten "basel"-Dreiecksnetz[2] löst und visualisiert:



[2]Den Code für die Definition des groben "basel"-Dreiecksnetzes, `mesh_generation_basel.m` finden Sie auf der Website der Vorlesung.