## Projekt.

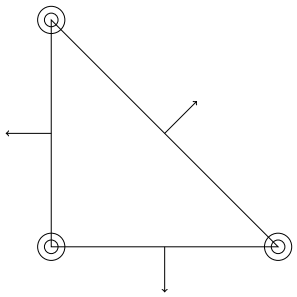
Bearbeiten bis: **Sonntag, 29.01.2023**

In den Programmieraufgaben haben wir bisher stückweise lineare Ansatzfunktionen auf Dreieckselementen betrachtet. Für einige Anwendungen ist es jedoch notwendig, dass die Ansatzfunktionen in $H^2(\Omega)$ liegen. Um das zu erreichen, müssen die Funktionen global mindestens einmal stetig differenzierbar sein. Solche Ansatzfunktionen sind beispielsweise durch die *Argyris*-Elemente gegeben. Dieses Projekt befasst sich mit der Konstruktion eines Argyris-Meshes, dem Umgang mit den zusätzlichen Freiheitsgraden und schliesslich dem Lösen der Laplace-Gleichung mit homogenen Dirichlet-Randdaten.

### Lokale Ansatzfunktionen

Zuerst wollen wir uns damit beschäftigen, die lokalen Ansatzfunktionen zu verstehen. Wie aus einer Übungsaufgabe bereits bekannt ist, existiert auf dem üblichen Referenzelement $\triangle$ genau ein Polynom vom Grad 5, welches folgenden vorgegebenen Bedingungen genügt:

- Funktionswerte in den Eckpunkten $(0,0)$, $(1,0)$ und $(0,1)$,
- Werte der ersten Ableitungen $\partial_x$ und $\partial_y$ in den Eckpunkten $(0,0)$, $(1,0)$ und $(0,1)$,
- Werte der zweiten Ableitungen $\partial^2_{xx}$, $\partial^2_{xy}$, sowie $\partial^2_{yy}$ in den Eckpunkten $(0,0)$, $(1,0)$ und $(0,1)$,
- Normalenableitungen in den Kantenmittelpunkten $(\frac{1}{2},0)$, $(\frac{1}{2},\frac{1}{2})$ und $(0,\frac{1}{2})$.



Folglich benötigen wir genau 21 lokale Basisfunktionen, welche für genau eine der vorgegebenen Bedingungen den Wert 1 und für die restlichen Bedingungen den Wert 0 annehmen. Diese Basisfunktionen sind unter `https://defelement.com/elements/argyris.html` tabelliert, nummeriert von 0 bis 20. Da MATLAB immer von 1 beginnend nummeriert, bezeichnen wir diese Ansatzfunktionen in diesem Projekt mit $\hat{\phi}_1, \dots, \hat{\phi}_{21}$. Der Einfachkeit halber drehen wir bei den Kantenfunktionen auch das Vorzeichen, falls erforderlich, um, da wir immer mit nach aussen gerichteten Normalen arbeiten möchten.

**Aufgabe 1.** Besuchen Sie die Seite `https://defelement.com/elements/argyris.html` und lesen Sie den Artikel. Überprüfen Sie stichprobenweise, dass die Funktionen den Anforderungen genügen.[1]

[1] Man soll bekanntlich nicht alles glauben, was man im Internet liest...

Diese lokalen Basisfunktionen benötigen wir für das Aufstellen der lokalen Masse- und Steifigkeitsmatrix. Eine Möglichkeit, sie auszuwerten, besteht darin, ihre Koeffizienten in der Monombasis abzuspeichern, sowie `function-handles` für die Monome anzulegen. Diese ordnen wir zuerst nach dem Grad und anschliessend lexikografisch, das heisst,

$$
\begin{array}{llllll}
\psi_1(\mathbf{x}) = 1, & & & & & \\
\psi_2(\mathbf{x}) = x, & \psi_3(\mathbf{x}) = y, & & & & \\
\psi_4(\mathbf{x}) = x^2, & \psi_5(\mathbf{x}) = xy, & \psi_6(\mathbf{x}) = y^2, & & & \\
\psi_7(\mathbf{x}) = x^3, & \psi_8(\mathbf{x}) = x^2y, & \psi_9(\mathbf{x}) = xy^2, & \psi_{10}(\mathbf{x}) = y^3, & & \\
\psi_{11}(\mathbf{x}) = x^4, & \psi_{12}(\mathbf{x}) = x^3y, & \psi_{13}(\mathbf{x}) = x^2y^2, & \psi_{14}(\mathbf{x}) = xy^3, & \psi_{15}(\mathbf{x}) = y^4, & \\
\psi_{16}(\mathbf{x}) = x^5, & \psi_{17}(\mathbf{x}) = x^4y, & \psi_{18}(\mathbf{x}) = x^3y^2, & \psi_{19}(\mathbf{x}) = x^2y^3, & \psi_{20}(\mathbf{x}) = xy^4, & \psi_{21}(\mathbf{x}) = y^5.
\end{array}
$$

Zusätzlich zu den Funktionen brauchen wir die Koeffizienten der lokalen Basisfunktionen bezüglich dieser Monombasis. Diese können in einer Matrix $\mathbf{B} = [b_{i,j}]_{i,j=1}^{21}$ abgespeichert werden, wobei die $i$-te Zeile von $\mathbf{B}$ die Koeffizienten der $i$-ten Basisfunktion enthält:

$$
\hat{\phi}_i(\mathbf{x}) = \sum_{j=1}^{21} b_{i,j}\,\psi_j(\mathbf{x}). \tag{1}
$$

Weiter benötigen wir neben den Monomen auch alle ihre Ableitungen, welche aufgrund der Linearität ebenfalls analog zu (1) berechnet werden können. Die `function [monomials, B] = assemble_functions()` gibt genau dafür ein $(21 \times 6)$-Cell-Array `monomials` zurück, sowie eine $(21 \times 21)$-Matrix $\mathbf{B}$. In der $j$-ten Zeile von `monomials` sind dabei die Funktion $\psi_j$ sowie alle ihre ersten und zweiten Ableitungen als `function-handle` abgespeichert, und zwar in der Reihenfolge

$$
\psi_j, \quad \partial_x\psi_j, \quad \partial_y\psi_j, \quad \partial^2_{xx}\psi_j, \quad \partial^2_{xy}\psi_j, \quad \partial^2_{yy}\psi_j.
$$

Mit diesen Monomen und der Matrix $\mathbf{B}$ lassen sich bereits die lokalen Masse- und Steifigkeitsmatrizen berechnen. Wie wir sehen werden, benötigen wir später die vier Matrizen $\mathbf{M}_{\text{ref}}, \mathbf{A}^{xx}_{\text{ref}}, \mathbf{A}^{xy}_{\text{ref}}$ und $\mathbf{A}^{yy}_{\text{ref}}$, wobei deren Einträge durch

$$
\begin{aligned}
m_{i,j} &= \left\langle \hat{\phi}_j,\, \hat{\phi}_i \right\rangle_{\triangle}, & a^{xx}_{i,j} &= \left\langle \partial_x\hat{\phi}_j,\, \partial_x\hat{\phi}_i \right\rangle_{\triangle}, \\
a^{xy}_{i,j} &= \left\langle \partial_y\hat{\phi}_j,\, \partial_x\hat{\phi}_i \right\rangle_{\triangle}, & a^{yy}_{i,j} &= \left\langle \partial_y\hat{\phi}_j,\, \partial_y\hat{\phi}_i \right\rangle_{\triangle}
\end{aligned}
$$

gegeben sind. Diese vier Matrizen finden Sie in den vier Textdateien `M_ref.txt`, `Axx_ref.txt`, `Axy_ref.txt`, sowie `Ayy_ref.txt`, welche mit dem Befehl `readmatrix()` eingelesen werden können. Die Integrale wurden dabei numerisch exakt mit einer Gauss-Quadratur mit hinreichend hohem Exaktheitsgrad berechnet.

## Freiheitsgrade I: Im Inneren des Gebietes

### Vernetzung

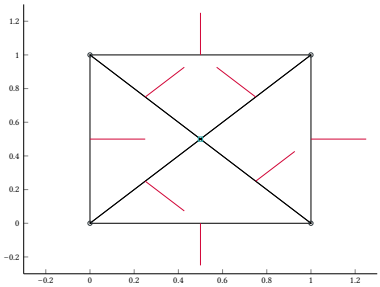
Wir wollen uns nun näher mit der Vernetzung eines Gebietes befassen. Gegeben sei dazu ein Dreiecksnetz zu einem Gebiet, wie es aus dem Programmierblatt 1 bereits bekannt ist. Wir benötigen nun zusätzlich alle Kantenmittelpunkte sowie einen zugehörigen Normalenvektor. Schliesslich sind die lokalen Basisfunktionen noch korrekt zu verkleben.

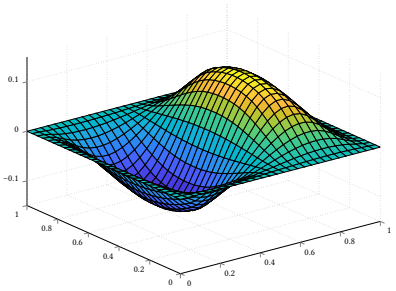
Gemäss dem ersten Programmierblatt wissen wir bereits, wie wir die Kantenmittelpunkte aus einem gegebenen Gitter extrahieren können. Anders als bei der Netzverfeinerung hängen wir sie hier aber nicht der Punkteliste an, sondern speichern sie in einer neuen Variable $\mathbf{Q} \in \mathbb{R}^{2\times n_E}$ in einem MATLAB-struct `dof_handler` ab. Hierbei bezeichne $n_E$ die Anzahl der Kanten des Gitters. Im gleichen Zug können auch die Normalen über die Kanten bestimmt werden, welche wir auch im `dof_handler` in einer Matrix $\mathbf{N} \in \mathbb{R}^{2\times n_E}$ abspeichern.

Analog zur Gitterverfeinerung benötigen wir im `dof_handler` auch die Indizes der Kantenmittelpunkte in der Matrix $\mathbf{Q}$, und demzufolge auch die der Normalen in $\mathbf{N}$. Diese speichern wir nun in einer zusätzlichen Matrix $\mathbf{K} \in \mathbb{R}^{3\times n_F}$ ab. In einer weiteren Matrix $\mathbf{O} \in \{-1, 1\}^{3\times n_F}$ sollen zusätzlich die Orientierungen der Normalen abgespeichert werden. Hierbei steht $+1$ für eine nach aussen gerichtete Normale, $-1$ für eine nach innen gerichtete.

Wichtig ist dabei, dass man sich die Mittelpunkte und Normalen in einer festgesetzten Anordnung abspeichert und diese Reihenfolge nie verändert. Sinnvoll sind beispielsweise folgende Anordnungen:

- Der $(i, j)$-te Eintrag bezieht sich auf die Kante gegenüber von $\mathbf{F}(i, j)$.
- Der $(i, j)$-te Eintrag bezieht sich auf die Kante von $\mathbf{F}(i, j)$ nach $\mathbf{F}(1 + \mathrm{mod}(i, 3),\ j)$.





Auf dem oben abgebildeten Gitter können diese Matrizen folgendermassen aussehen, wenn man die zweite Anordnung der Mittelpunkte wählt. Die Punkte und die Elemente seien dazu genau gleich wie auf Programmierblatt 1 nummeriert.

$$\mathbf{Q} = \begin{bmatrix} 0.5 & 0.25 & 0 & 0.75 & 1 & 0.75 & 0.5 & 0.25 \\ 0 & 0.25 & 0.5 & 0.25 & 0.5 & 0.75 & 1 & 0.75 \end{bmatrix}$$

$$\mathbf{N} = \begin{bmatrix} 0 & \sqrt{2}/2 & 1 & \sqrt{2}/2 & 1 & -\sqrt{2}/2 & 0 & \sqrt{2}/2 \\ -1 & -\sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 & \sqrt{2}/2 & 1 & \sqrt{2}/2 \end{bmatrix}$$

$$\mathbf{K} = \begin{bmatrix} 1 & 5 & 7 & 3 \\ 4 & 6 & 8 & 2 \\ 2 & 4 & 6 & 8 \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 1 & 0.5 \end{bmatrix}$$

$$\mathbf{O} = \begin{bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix} \qquad \mathbf{F} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 5 & 5 & 5 & 5 \end{bmatrix}$$

**Aufgabe 2**. Schreiben Sie eine Funktion

```
function [mesh, dof_handler] = get_dof_handler(mesh),
```

welche für ein gegebenes Dreiecksnetz einen `dof_handler` zurückgibt. Stellen Sie darin auch sicher, dass die Elemente des Netzes positiv orientiert sind (vgl. Programmierblatt 2), indem Sie, falls nötig, Einträge in `mesh.F` vertauschen. Schreiben Sie weiter eine Funktion

```
function mesh_plot_argyris(mesh, dof_handler),
```

welche ein Argyris-Mesh ähnlich zur obigen, linken Abbildung darstellt.

Mit diesen Informationen können wir nun die globalen Freiheitsgrade berechnen. Für einen Gitterpunkt $\mathbf{p}$ im Inneren benötigen wir die folgenden Werte, um eine Funktion $f$ darzustellen:

$$f(\mathbf{p}), \quad \partial_x f(\mathbf{p}), \quad \partial_y f(\mathbf{p}), \quad \partial^2_{xx} f(\mathbf{p}), \quad \partial^2_{xy} f(\mathbf{p}), \quad \partial^2_{yy} f(\mathbf{p}).$$

Folglich benötigen wir für jeden inneren Punkt auch sechs Werte, welche den globalen Freiheitsgraden entsprechen. Für jede Kante benötigen wir zusätzlich die Normalenableitung der Funktion $f$ am Kantenmittelpunkt entlang der Orientierung des zugehörigen Normalenvektors. Die Freiheitsgrade speichern wir in einem Vektor $\mathbf{f}$ ab, mit

$$\begin{bmatrix} \mathbf{f}_{6i-5} \\ \vdots \\ \mathbf{f}_{6i} \end{bmatrix} = \begin{bmatrix} f(\mathbf{p}_i) \\ \vdots \\ \partial^2_{yy} f(\mathbf{p}_i) \end{bmatrix}, \qquad i = 1, \dots, n_P, \qquad \mathbf{f}_{6n_P+i} = \partial_{\mathbf{n}} f(\mathbf{q}_i), \qquad i = 1, \dots, n_E,$$

wobei $\mathbf{q}_i$ der Mittelpunkt der $i$-ten Kante bezeichne.

**Aufgabe 3**. Schreiben Sie eine Funktion

```
function g = compute_interpolation(mesh, dof_handler, ...
        f, fx, fy, fxx, fxy, fyy),
```

welche für ein Gitter und eine Funktion $f$ den zugehörigen Koeffizientenvektor berechnet.

### Globale Basisfunktionen

Wie im stückweise linearen Fall müssen auch hier die lokalen Basisfunktionen in sinnvoller Weise zu globalen Basisfunktionen verklebt werden. Insbesondere sind zu jedem Gitterpunkt $\mathbf{p}_i$ die Funktionen $\varphi_{6i-5}, \dots, \varphi_{6i}$ erforderlich, welche

$$\begin{bmatrix} 1 & \partial_x & \partial_y & \partial^2_{xx} & \partial^2_{xy} & \partial^2_{yy} \end{bmatrix} \begin{bmatrix} \varphi_{6i-5}(\mathbf{p}) \\ \varphi_{6i-4}(\mathbf{p}) \\ \varphi_{6i-3}(\mathbf{p}) \\ \varphi_{6i-2}(\mathbf{p}) \\ \varphi_{6i-1}(\mathbf{p}) \\ \varphi_{6i}(\mathbf{p}) \end{bmatrix} = \mathbf{I}_6$$

erfüllen. Analog dazu soll zu jeder Kante $\mathbf{e}_j$ eine Funktion $\varphi_{6n_P+j}$ gehören, so dass die Normalenableitung im Kantenmittelpunkt gleich 1 ist, sie aber zu allen anderen Freiheitsgraden nichts beiträgt.

Für die Konstruktion dieser Funktionen können wir elementweise vorgehen, indem auf jedem Element $T$ aus 21 transportierten, lokalen Basisfunktionen $\phi_k$ die Restriktionen $\varphi_j|_T$ gebastelt werden. Befinden wir uns beispielsweise auf dem Element $\mathbf{F}_i$, suchen wir die Koeffizienten $c^{(i)}_{k,j}$, so dass

$$\varphi_j\big|_{\mathbf{F}_i} = \sum_{k=1}^{21} c^{(i)}_{k,j} \phi_k, \qquad j \in \bigcup_{\ell=1}^{3} \{6F_{\ell,i} - 5, \dots 6F_{\ell,i}\} \cup \{6n_P + M_{1,i}, 6n_P + M_{2,i}, 6n_P + M_{3,i}\}.$$

Während das Verkleben der Basisfunktionen für Punktauswertungen trivial ist, fliesst für Basisfunktionen, welche sich auf Ableitungen beziehen, auch der Transport in die Koeffizienten mit ein. Dies wurde glücklicherweise aber schon auf Blatt 5, Aufgabe 4 besprochen.

## Assemblierung der globalen Elementmatrizen

Da wir jetzt wissen, wie die globalen Basisfunktionen aussehen, können wir die globalen Masse- und Steifigkeitsmatrizen assemblieren. Dies geschieht ähnlich wie im linearen Fall:

**Algorithmus** Assemblierung der Argyris-FEM-Matrizen

*Input:* Gitter `mesh` mit zugehörigem `dof_handler`
*Output:* Argyris-FEM-Matrizen $\mathbf{M}$ und $\mathbf{A}$ im sparse-Format

alloziere $\mathbf{V}_m$, $\mathbf{V}_a$, $\mathbf{I}$ und $\mathbf{J}$
**for** $k = 1, \dots, n_F$ **do**
initiiere $\mathbf{p}_j := \mathbf{P}(:, \mathbf{F}_{j,k})$ für $j = 1, 2, 3$
initiiere $\mathbf{a} := \mathbf{p}_2 - \mathbf{p}_1$ und $\mathbf{b} := \mathbf{p}_3 - \mathbf{p}_1$
initiiere $d := \det([\mathbf{a}, \mathbf{b}])$
setze

$$\mathbf{v}_k = \Big[6\mathbf{F}_{1,k} - 5 : 6\mathbf{F}_{1,k}, \quad 6\mathbf{F}_{2,k} - 5 : 6\mathbf{F}_{2,k}, \quad \dots \\ 6\mathbf{F}_{3,k} - 5 : 6\mathbf{F}_{3,k}, \quad 6n_P + \mathbf{K}_{:,k}^{\mathsf{T}}\Big]^{\mathsf{T}}$$

setze $\mathbf{I}_k := \texttt{repmat(}\mathbf{v}_k\texttt{, 1, 21)}$ und $\mathbf{J}_k := \mathbf{I}_k^{\mathsf{T}}$,
assembliere die Matrix $\mathbf{L}$ so, dass

$$\mathbf{L}\begin{bmatrix} c_{1,v_1}^{(k)} & c_{1,v_2}^{(k)} & \dots & c_{1,v_{21}}^{(k)} \\ c_{2,v_1}^{(k)} & c_{1,v_2}^{(k)} & \dots & c_{2,v_{21}}^{(k)} \\ \vdots & \vdots & & \vdots \\ c_{21,v_1}^{(k)} & c_{21,v_2}^{(k)} & \dots & c_{21,v_{21}}^{(k)} \end{bmatrix} = \texttt{eye(21)}$$

setze $\mathbf{T} := \mathbf{L}^{-\mathsf{T}}$ und

$$\mathbf{M}_k := \mathbf{T}\mathbf{M}_{\text{ref}}\mathbf{T}^{\mathsf{T}} \cdot d$$
$$\mathbf{A}_k := \mathbf{T}\big(\|\mathbf{b}\|^2\mathbf{A}_{xx,\text{ref}} - \mathbf{a}^{\mathsf{T}}\mathbf{b}(\mathbf{A}_{xy,\text{ref}} + \mathbf{A}_{xy,\text{ref}}^{\mathsf{T}}) + \|\mathbf{a}\|^2\mathbf{A}_{yy,\text{ref}}\big)\mathbf{T}^{\mathsf{T}}/d,$$

setze

$$\mathbf{I}(441k - 440 : 441k) := \mathbf{I}_k(:), \qquad \mathbf{V}_m(441k - 440 : 441k) := \mathbf{M}_k(:),$$
$$\mathbf{J}(441k - 440 : 441k) := \mathbf{J}_k(:), \qquad \mathbf{V}_a(441k - 440 : 441k) := \mathbf{A}_k(:).$$

**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 4**. Schreiben Sie eine Funktion

```
function [M, A] = assemble_mass_and_stiffness_global(mesh, ...
        dof_handler, M_ref, Axx_ref, Axy_ref, Ayy_ref),
```

welche die Masse- und Steifigkeitsmatrix gemäss dem Algorithmus aufstellt.

Hinweis. *Stellen Sie dabei sicher, dass die Kantenfunktionen den richtigen globalen Freiheitsgraden zugeordnet werden. Falls die Kanten anders angeordnet sind, enthält die Matrix* $\mathbf{L}$ *auch andere Koeffizienten! Beachten Sie hier auch, dass sich die Nummerierung in den lokalen Masse- und Steifigkeitsmatrizen auf die Reihenfolge auf* `defelement` *bezieht, nicht auf unsere Anordnung der Kanten.*

Diese Matrizen sollen nun am Einheitsquadrat getestet werden. Da per Konstruktion alle Polynome bis zum Grad 5 exakt dargestellt werden können, müssen diese mit der Massematrix ebenfalls exakt integriert werden, und zwar unabhängig vom Verfeinerungslevel. Sind also $p$, $q$ Polynome vom Grad $\leq 5$, muss

$$\int_{\square} p(\mathbf{x})q(\mathbf{x})\,\mathrm{d}\mathbf{x} \approx \mathbf{p}^{\mathsf{T}}\mathbf{M}\mathbf{q}$$

gelten, wobei $\mathbf{p}, \mathbf{q}$ die Koeffizientenvektoren der Polynome $p$, respektive $q$ seien, und $\mathbf{M}$ die globale Massematrix bezeichne. Ebenso muss gelten, dass

$$\int_{\square} \langle \nabla p, \nabla q \rangle \, \mathrm{d}\mathbf{x} \approx \mathbf{p}^{\mathsf{T}} \mathbf{A} \mathbf{q}.$$

**Aufgabe 5**. Schreiben Sie ein Skript

`test_mass_and_stiffness.m`,

welches die Masse- und Steifigkeitsmatrix auf dem einmal verfeinerten Gitter des Einheitsquadrats testet. Für den Test sollen die Polynome aus dem cell-Array `monomials` mit Hilfe der Funktion `integral_square_gauss` aus der Beilage mit $m = 6$ integriert werden. Ihre Fehler sollten die Grössenordnung $10^{-14}$ haben.

## Freiheitsgrade II: Am Gebietsrand

Bis jetzt haben wir jeden Punkt behandelt, als läge er im Gebietsinneren. Da sich die Art und Zahl der Freiheitsgrade eines Randpunktes von der eines Punktes im Inneren unterscheidet, muss dies noch angepasst werden. Der Einfachkeit halber wollen wir uns auf homogene Dirichlet-Randdaten beschränken. Weiter wollen wir annehmen, dass zwischen zwei Eckpunkten immer ein Punkt auf dem Rand liegt, welcher aber keine Ecke ist. Weiter soll vorausgesetzt sein, dass sich die Eckpunkte am Anfang der Punkteliste befinden.[2]

Befinden wir uns also in einer Ecke $\mathbf{p}$ des polygonal berandeten Gebietes, muss die Lösung auf beiden Kanten, welche die Ecke mit einem weiteren Randpunkt verbindet, dem Nullpolynom entsprechen. Wegen der Eindeutigkeit der Interpolationsaufgabe muss also sowohl der Wert der Lösung $u$ am Punkt $\mathbf{p}$ als auch die ersten und zweiten Ableitungen in Richtung der Kanten verschwinden. Der einzige, übrig gebliebene Freiheitsgrad ist somit die gemischte zweite Ableitung

$$\partial^2_{\mathbf{e}_k \mathbf{e}_\ell} u(\mathbf{p}) = \mathbf{e}_k^{\mathsf{T}} (\mathbf{H}u)(\mathbf{p}) \mathbf{e}_\ell,$$

wobei $\mathbf{H}u$ die Hesse-Matrix von $u$ und $\mathbf{e}_k$ und $\mathbf{e}_\ell$ die zwei Kanten bezeichnen.

Etwas anders präsentiert sich die Situation an einem Punkt $\mathbf{p}$ am Gebietsrand, welcher aber kein Eckpunkt ist. In diesem Falle haben wir bei $\mathbf{p}$ sowohl eine wohldefinierte Tangente $\mathbf{t}$ als auch eine wohldefinierte Normale $\mathbf{n}$. Da die Lösung $u$ am Rand verschwinden muss, muss der Wert von $u$ in $\mathbf{p}$ wieder verschwinden, genau so wie die erste und zweite Ableitung in der Tangentialrichtung. Übrig bleiben also

$$\partial_{\mathbf{n}} u(\mathbf{p}), \quad \partial^2_{\mathbf{nn}} u(\mathbf{p}), \quad \partial^2_{\mathbf{tn}} u(\mathbf{p}).$$

In diesem Fall sehen auch die globalen Basisfunktionen leicht anders aus. Es lässt sich herleiten, dass der zum Knoten gehörige Block der Transformationsmatrix $\mathbf{L}$ in beiden Fällen gleich von den zwei Vektoren $\mathbf{t}$ und $\mathbf{n}$ abhängt, welche entweder Tangente und Normale an $\mathbf{p}_i$ oder die beiden Kanten $\mathbf{e}_k$ und $\mathbf{e}_\ell$. Diese Matrizen können mit der Funktion `assemble_boundary_block` aufgestellt werden.

**Aufgabe 6**. Ergänzen Sie Ihr mesh um eine Variable $\mathbf{V} \in \{0, 1\}^{n_P}$, welche an der $i$-ten Stelle genau dann den Wert 1 enthält, wenn der Punkte $\mathbf{p}_i$ ein Eckpunkt ist. Stellen Sie sicher, dass dies bei der Gitterverfeinerung angepasst wird. Ergänzen Sie weiter ihren `dof_handler` um eine Variable $\mathbf{R} \in \mathbb{R}^{4 \times n_P}$, wobei

$$\mathbf{R}(1:2,\ i) = \text{eine Kante an } \mathbf{p}_i, \qquad \mathbf{R}(3:4,\ i) = \text{andere Kante an } \mathbf{p}_i,$$

[2] Dies lässt sich durch einmaliges Verfeinern und Umnummerieren des Ursprungsgebietes erreichen.

falls $\mathbf{p}_i$ ein Eckpunkt ist, sowie

$$\mathbf{R}(1:2,\ i) = \text{Tangente an } \mathbf{p}_i, \qquad \mathbf{R}(3:4,\ i) = \text{Normale an } \mathbf{p}_i,$$

falls $\mathbf{p}_i$ sonst ein Randpunkt ist. Vervollständigen Sie dafür den Skelettcode und integrieren Sie ihn in ihre `get_dof_handler`-Funktion.

Adaptieren Sie Ihre Funktion `compute_interpolation`, indem Sie für Eck- und Randpunkte die richtigen Freiheitsgrade abspeichern. Passen Sie Ihre Funktion `assemble_mass_and_stiffness_global` dahingehend an, dass Sie im Falle von Randpunkten die richtigen Werte zu den richtigen Freiheitsgraden assemblieren. Speichern Sie die Freiheitsgrade für Randknoten[3] in der folgenden Reihenfolge ab:

$$f(\mathbf{p}), \quad \partial_{\mathrm{t}} f(\mathbf{p}), \quad \partial_{\mathrm{n}} f(\mathbf{p}), \quad \partial^2_{\mathrm{tt}} f(\mathbf{p}), \quad \partial^2_{\mathrm{tn}} f(\mathbf{p}), \quad \partial^2_{\mathrm{nn}} f(\mathbf{p}).$$

Die Tangenten und Normalen an den Dirichlet-Randpunkten sowie die Kanten an den Eckpunkten können dabei gemäss folgendem Algorithmus bestimmt werden:

---

**Algorithmus** Extrahieren der Tangenten und Normalen

---

*Input:* Gitter `mesh`
*Output:* Tangenten und Normalen $\mathbf{R}$

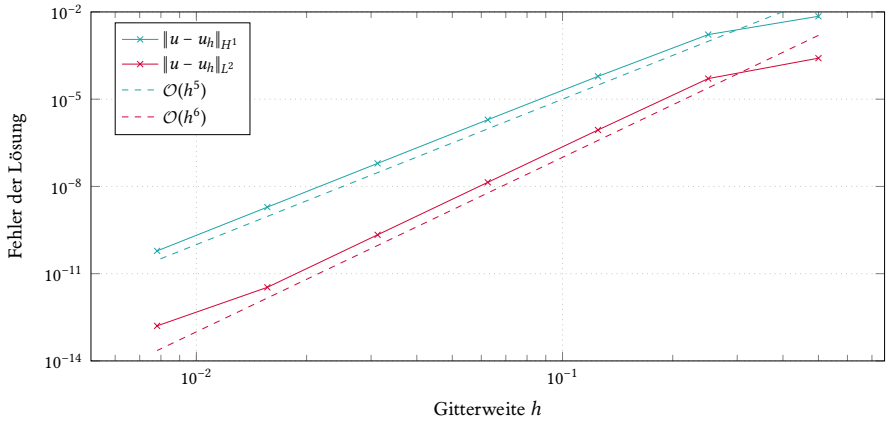
alloziere $\mathbf{R}$
**for** $k = 1, \dots, n_P$ **do**
  **if** $\mathbf{p}_i$ ist ein Eckpunkt **then**
    finde die beiden Randpunkte $\tilde{\mathbf{p}}_1$ und $\tilde{\mathbf{p}}_2$, welche mit $\mathbf{p}$ eine Kante teilen
    setze $\mathbf{R}(1:2,\ i) := \tilde{\mathbf{p}}_1 - \mathbf{p}_i$, $\mathbf{R}(3:4,\ i) := \tilde{\mathbf{p}}_2 - \mathbf{p}_i$
  **else if** $\mathbf{p}_i$ ist ein Dirichlet-Randpunkt **then**
    finde einen Nachbarrandpunkt $\tilde{\mathbf{p}}$ und setze $\mathbf{R}(1:2,\ i) := \tilde{\mathbf{p}} - \mathbf{p}_i$
    setze $\mathbf{R}(3:4,\ i) := \left[p_i^{(2)} - \tilde{p}^{(2)},\ \tilde{p}^{(1)} - p_i^{(1)}\right]^{\mathsf{T}} / \|\tilde{\mathbf{p}} - \mathbf{p}_i\|$
    falls die Normale nach innen zeigt, so drehe sie um
  **end if**
**end for**

---

**Aufgabe 7**. Testen Sie Ihre Funktionen auf dem ein- bis siebenmal verfeinerten Einheitsquadrat mit der Funktion

$$u(x, y) = \sin(2\pi x)\sin(2\pi y),$$

und vergleichen Sie den $L^2$- und den $H^1$-Fehler zur exakten Lösung in einem `loglog`-Plot. Für die Lösung des Gleichungssystems dürfen Sie den \-Löser benutzen. Zeichnen Sie zusätzlich die Vergleichsgeraden $h^5$ und $h^6$ ein. Stellen Sie mittels einem *logical array* `idxs` sicher, dass Sie dabei die Dirichlet-Daten aus dem Gleichungssystem entfernen.




[3] für Eckknoten entsprechend mit den Kanten $\mathbf{t} := \mathbf{e}_k$, $\mathbf{n} := \mathbf{e}_\ell$