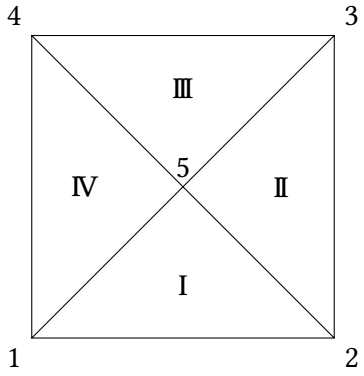
Programmierblatt 1.

Bearbeiten bis: **Sonntag, 09.10.2022**

Das Ziel dieser Serie von Programmierblättern ist es, die Implementierung der Finite-Elemente-Methode mit linearen Elementen auf konformen Dreiecksnetzen für elliptische PDEs (in MATLAB) zu vollführen. Da die Finite-Elemente-Methode massgeblich auf der Darstellung eines Gebietes als Netz aufbaut, ist dieses erste Blatt der Darstellung und Implementierung eines Dreiecksnetzes gewidmet.

Darstellung von Dreiecksnetzen

Um die grundlegenden Konzepte, die hier vorgestellt werden, besser zu verstehen, wollen wir zunächst eine einfache Geometrie mit zugehörigem, grobem Dreiecksnetz betrachten. Gegeben sei das Einheitsquadrat, das in vier kongruente Dreiecke I–IV unterteilt ist.



Ein Dreiecksnetz kann kanonisch durch eine *Punktliste* und eine *Elementliste* dargestellt werden. Die Punkteliste enthält dabei die Koordinaten aller n_P Eckpunkte des Dreiecksnetzes,

$$\mathbf{p}_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}, \dots, \mathbf{p}_{n_P} = \begin{bmatrix} x_{n_P} \\ y_{n_P} \end{bmatrix} \in \mathbb{R}^2.$$

Die Punktliste ist in MATLAB in folgender Form als Matrix abzuspeichern

$$\mathbf{P} = \begin{bmatrix} x_1 & x_2 & \cdots & x_{n_P} \\ y_1 & y_2 & \cdots & y_{n_P} \end{bmatrix} \in \mathbb{R}^{2\times n_P}.$$

Die Elementliste enthält nun zu jedem Dreieck des Dreicksnetzes die Indizes der drei Eckpunkte bezüglich der Punkteliste. Für das k-te Dreieck in dem Dreicksnetz seien $i_1^{(k)}$, $i_2^{(k)}$ und $i_3^{(k)}$ die Indizes seiner Ecken, d.h. $\mathbf{p}_{i_1^{(k)}}$, $\mathbf{p}_{i_2^{(k)}}$ und $\mathbf{p}_{i_3^{(k)}}$ sind die Eckpunkte. Dann sind diese Indizes in MATLAB spaltenweise abzuspeichern, womit sich eine dreizeilige Matrix der Form

$$\mathbf{F} = \begin{bmatrix} i_1^{(1)} & i_1^{(2)} & \cdots & i_1^{(n_F)} \\ i_2^{(1)} & i_2^{(2)} & \cdots & i_2^{(n_F)} \\ i_3^{(1)} & i_3^{(2)} & \cdots & i_3^{(n_F)} \end{bmatrix} \in \{1, 2, \dots, n_P\}^{3\times n_F} \subset \mathbb{N}^{3\times n_F}$$

ergibt, wo n_F die Anzahl der Dreiecke ist.

Beispielsweise lauten für das Einheitsquadrat die Punkt- und Elementliste:

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 1 & 0.5 \end{bmatrix} \quad \text{und} \quad \mathbf{F} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 5 & 5 & 5 & 5 \end{bmatrix}.$$

Zur Beschreibung der Randbedingungen definieren wir zusätzlich noch einen Zeilenvektor

$$\mathbf{B} = [b_i]_{i=1}^{n_P} \in \{0, 1, 2\}^{1 \times n_P} \subset \mathbb{N}^{1 \times n_P},$$

der ein *Flag* für die Randbedingungen enthält. Ist der i-te Punkt der Punktliste, $\mathbf{p}_i$, ein innerer Punkt, so ist $b_i = 1$, bei einem Randpunkt mit Dirichlet-Randbedingung sei $b_i = 0$ und bei einem Punkt mit Neumann-Randbedingung setzen wir $b_i = 2$.

Aufgabe 1.

Schreiben Sie eine Funktion

```
function mesh = mesh_generation_square(),
```

welche das oben definierte Dreiecksnetz des Einheitsquadrats und die entsprechenden Randbedingungen in der Variablen `mesh` zurückgibt. Bei den Ecken 1 und 2 sollen dabei Neumann-Randbedingungen gesetzt sein, bei 3 und 4 Dirichlet-Randbedingungen. Dabei soll `mesh` hier und auch nachfolgend jeweils ein `MATLAB`-struct sein, welches **P**, **F** und **B** in den Feldern `mesh.P`, `mesh.F` und `mesh.B` gespeichert hat.

Implementieren sie weiter eine Funktion

```
function mesh_plot(mesh),
```

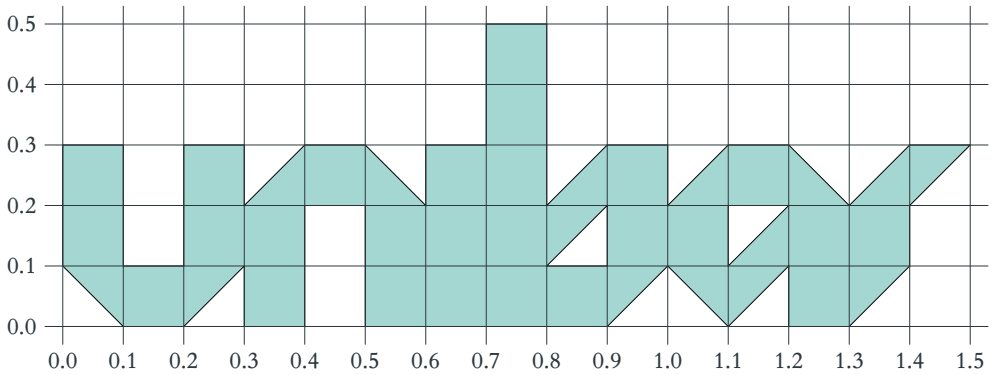
die ein Dreiecksnetz visualisiert und die Randknoten farblich markiert, wobei für Dirchlet- und Neumann-Randbedingungen je ein eigenes Markersymbol verwendet wird.

Aufgabe 2.

Schreiben Sie eine Funktion

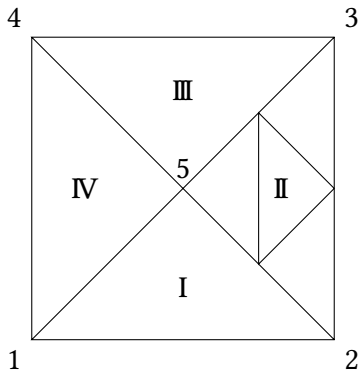
```
function mesh = mesh_generation_unibas(),
```

das ein grobes Dreiecksnetz inklusive Randbedingungen für die unten dargestellte Geometrie in der Variablen `mesh` zurückgibt. Bei allen Ecken, die am Rand des Gebietes liegen, sollen dabei Dirichlet-Randbedingungen gesetzt sein.



Verfeinerung von Dreiecksnetzen

Nachdem nun bekannt ist, wie Dreiecksnetze dargestellt werden können, wollen wir diese nun in gleichbleibender Darstellung verfeinern. Dies geschieht durch das Halbieren aller Kanten und das anschliessende Verbinden der neuen Punkte zu Dreiecken, wie in der folgenden Abbildung für das Element Ⅱ des Einheitsquadrates gezeigt ist:



Für die algorithmische Umsetzung wird zunächst eine *Kantenliste* **E** implementiert. Diese enthält zu einem gegebenen Punkt $\mathbf{p}_i$ der Punktliste in der i-ten Spalte alle Indizes der Endpunkte adjazenter Kanten, deren Index grösser als i ist. Durch diese Konvention bei der Sortierung der Kanten, wird die Kantenliste eindeutig. Insbesondere muss bei der Erstellung der Kantenliste darauf geachtet werden, ob eine Kante bereits vorhanden ist, oder nicht

Neben der Kantenliste hat man dabei zusätzlich noch einen Zeilenvektor **K**, der in der i-ten Spalte angibt, wie viele Kanten in der i-ten Spalte der Kantenliste vorkommen. Weiter ist es für die Handhabung der Randbedingungen von Vorteil, wen man zu der Kantenliste auch eine Liste **I** hat, welche angibt, ob die Kante an der gleichen Stelle in der Kantenliste im Inneren oder am Rand ist. Liegt die Kante im Inneren, soll der Wert in **I** gleich 1 und ansonsten 0 sein. Wenn eine Kante im Inneren liegt, so wird sie in zwei Dreiecken vorkommen.

Zum Beispiel haben die drei Listen für das Einheitsquadrat von oben die Form

$$\mathbf{E} = \begin{bmatrix} 2 & 5 & 5 & 5 \\ 5 & 3 & 4 & \\ 4 & & & \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} 3 & 2 & 2 & 1 & 0 \end{bmatrix} \quad \text{und} \quad \mathbf{I} = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & \\ 0 & & & \end{bmatrix}.$$

Die Erstellung der Kantenliste erfolgt nach folgendem Algorithmus:

Algorithmus Erstellen der Kantenliste

Input: Elementliste **F**
Output: Kantenliste **E**, Kantenlistenlängen **K**, Kantenlistenflags **I**

```
for i = 1, ..., n_F do
    for j = 1, 2, 3 do
        if Kante [F(j, i), F(1 + mod(j, 3), i)] noch nicht vorhanden then
            füge Kante zu E hinzu
        else
            die Kante ist im Inneren, setze an der richtigen Stelle in I eine 1
        end if
    end for
end for
```

Mit Hilfe der Kantenliste lässt sich eine vorgegebene Triangulierung leicht verfeinern. Hierzu werden zunächst alle Kanten der Kantenliste halbiert, die neuen Punkte an die Punkteliste angehängt und die entsprechenden Randbedingungen gesetzt:

Algorithmus Erstellen der neuen Punktliste und der Indizesliste

Input: Punkteliste **P**, Randbedingungen **B**, Kantenliste **E**, Kantenlistenlängen **K**, Kantenlistenflags **I**

Output: Punkteliste **PP**, Indizes neuer Punkte **L**, Randbedingungen **BB**

setze **PP** := **P**
setze **BB** := **B**
for $i = 1, \ldots,$ `length`(**E**) **do**
  for $j = 1, \ldots,$ **K**(i) **do**
    berechne den Mittelpunkt der Kante $\big[i,\ \mathbf{E}(j,i)\big]$, hänge ihn an **PP** an
    und setze **L**(j,i) := `length`(**PP**)
    if **I**$(j,i) = 1$ **then**
      setze **BB**(**L**(j,i)) := 1
    else
      setze **BB**(**L**(j,i)) := $\max\big\{\mathbf{B}(i), \mathbf{B}(\mathbf{E}(j,i))\big\}$
    end if
  end for
end for

Wesentlich im obigen Algorithmus ist, dass die Indizes der neuen Punkte in der Liste **L** abgespeichert werden, da sie für die Aktualisierung der Elementliste benötigt werden. Daneben ist zu bemerken, dass am Rand zwischen einem Randpunkt mit Dirichlet- und einem mit Neumann-Randbedingungen ein Punkt mit Neumann-Randbedingungen gesetzt wird. Die neue Elementliste ergibt sich aus folgendem Algorithmus:

Algorithmus Erstellen der neuen Elementliste

Input: Elementliste **F**, Kantenliste **E**, Indizes neuer Punkte **L**

Output: Elementliste **FF**

for $i = 1, \ldots,$ `length`(**F**) **do**
  for $j = 1, \ldots, 3$ **do**
    finde den Index (k_1, k_2) der Kante $\big[\mathbf{F}(j,i), \mathbf{F}\big(1 + \mathrm{mod}(j,3),\ i\big)\big]$ in **E**
    setze $m(j)$:= **L**(k_1, k_2)
  end for
  generiere neue Elemente und füge diese zu **FF** hinzu, d.h.

$$\mathbf{FF}(:, 4i-3 : 4i) := \begin{bmatrix} \mathbf{F}(1,i) & m(1) & m(3) & m(2) \\ m(1) & \mathbf{F}(2,i) & m(2) & m(3) \\ m(3) & m(2) & \mathbf{F}(3,i) & m(1) \end{bmatrix}$$

end for

Aufgabe 3.

Implementieren Sie eine Funktion

```
function meshf = mesh_refine(mesh),
```

die eine vorgegebene Triangulierung in `mesh` verfeinert und das Ergebnis in `meshf` abspeichert. Testen Sie diese Funktion mit den vorhandenen Grobgittern aus den Aufgaben 1 und 2.