Projekt.

Bearbeiten bis: **Freitag, 28.02.2025**

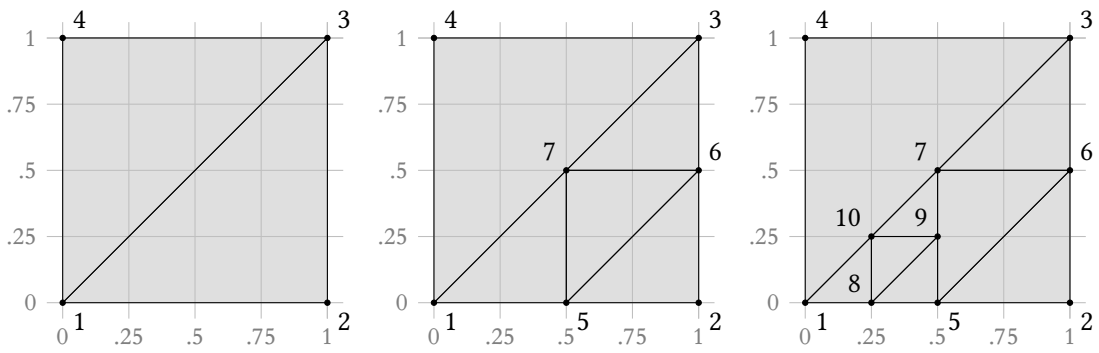
In den Programmierblättern haben wir die Finite-Elemente-Methode mit linearen Elementen auf konformen Dreiecksnetzen für elliptische PDEs implementiert. Dabei haben wir die konformen Dreiecksnetze jeweils durch (mehrmaliges) uniformes Verfeinern vorgegebener, grober Dreiecksnetze gewonnen. Wie wir aber aus der Vorlesung wissen, gibt es für die Poisson-Gleichung beweisbare A-posteriori-Fehlerschätzer, welche man benutzen kann, um das Dreiecksnetz nur an den Stellen zu verfeinern, wo der Fehler tatsächlich gross ist. Um sicherzustellen, dass die so lokal verfeinerten Dreiecksnetze *nicht entartet* bleiben, muss man beim Verfeinern der Dreiecksnetze aufpassen. Dafür sind verschiedene Methoden bekannt, zum Beispiel:

- die Benutzung von temporären *Transitionskanten*,
- ein Vorgehen namens *Newest Vertex Bisection*,
- das Erlauben von *hängenden Knoten*
- oder eine jeweilge, *automatische Neugenerierung* vom ganzen Dreiecksnetz.

Das Ziel dieses Projekts ist die Implementierung von Dreiecksnetzen mit hängenden Knoten in MATLAB, mit welchen es dann möglich ist, die Poisson-Gleichung adaptiv mit Hilfe von A-posteriori-Fehlerschätzern zu lösen. Für die Implementierung der Dreiecksnetze mit hängenden Knoten werden wir hierbei eine Darstellung von hierarchischen Dreiecksnetzen anwenden, welche massgeblich von Binär- und Quaternärbäumen inspiriert ist.

Darstellung und Verfeinerung von hierarchischen Dreiecksnetzen

Es gibt verschiedene Methoden, ein Dreiecksnetz in $\mathbb{R}^2$ darzustellen. In den Programmierblättern haben wir von einer klassischen, aber zugleich auch sehr kompakten Darstellung mittels Punkt- und Elementliste gebrauch gemacht. Dementgegen werden wir hier eine sehr ausführliche Darstellung betrachten, welche aus mehreren Listen besteht und von einem (groben) Dreiecksnetz ohne hängende Knoten startet und die Verfeinerungen hierarchisch darstellt. Die Benutzung einer hierarchischen Darstellung der Dreiecke und Kanten im Dreiecksnetz erlaubt es uns, beim Verfeinern jeweils nur Punkte, Kanten und Dreiecke hinzufügen zu müssen. Weiter ist es einerseits hilfreich für die Implementierung des Fehlerschätzers, die Kanten, aus welchen die Ränder der Dreiecke bestehen, dargestellt zu haben. Andererseits ist es hilfreich für die Handhabung der hängenden Knoten, die Hierarchien der Kanten zu haben. Um die Darstellung der Dreiecksnetze klarer kommunizieren zu können, betrachten wir die folgende Sequenz von Dreiecksnetzen:



Dabei ergibt sich das zweite Netz aus dem ersten durch Verfeinerung des Dreiecks $(1, 2, 3)$ und das dritte aus dem zweiten durch Verfeinerung des Dreiecks $(1, 5, 7)$. Wir setzen den oberen Rand als Dirichlet-Rand und die anderen als Neumann-Ränder an, das heisst die Punkte 1 und 2 sind auf dem Neumann- und 3 und 4 auf dem Dirichlet-Rand.

Die Darstellung, die wir hier benutzen werden, basiert auf acht Listen, welche man als Matrizen oder Zeilenvektoren abspeichern kann.

- Die ersten zwei Listen sind durch die Punkte indiziert: die *Punkt-zu-Koordinaten*-Liste speichert die Koordinaten aller Punkte, während die *Punkt-zu-Typ*-Liste den Typ (im Inneren/auf dem Dirichlet-/Neumann-Rand) aller Punkte speichert.
- Die nächsten drei Listen sind durch die Kanten indiziert: die *Kante-zu-Punkt*-Liste speichert die Punkte aller Kanten, die *Kante-zu-Typ*-Liste den Typ (im Inneren/auf dem Dirichlet-/Neumann-Rand) aller Kanten, während die *Kante-zu-Kante*-Liste die Hierarchie der Kanten untereinander speichert.
- Die letzten drei Listen sind durch die Dreiecke indiziert: die *Dreieck-zu-Punkt* Liste speichert die Punkte aller Dreiecke, die *Dreieck-zu-Kante* Liste die Kanten aller Dreiecke, während die *Dreieck-zu-Dreieck* Liste die Hierarchie der Dreiecke untereinander speichert.

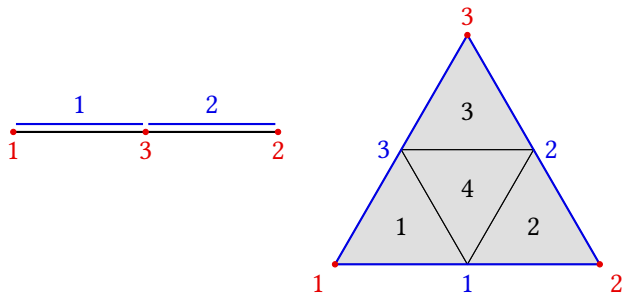
All diese Listen werden wir als Felder in einem MATLAB-`struct` speichern. Die Anzahl der Punkte, Kanten und Dreiecke bezeichnen wir mit n_P, n_S beziehungsweise n_D. Die genaue Beschreibung der Listen zur Implementierung lautet wie folgt:

- Die *Punkt-zu-Koordinaten*-Liste wird als Feld `p2c` gespeichert und ist eine Matrix der Grösse $2 \times n_P$. Sie enthält in der i-ten Spalte die Koordinaten des i-ten Punktes.
- Die *Punkt-zu-Typ*-Liste wird als Feld `p2t` gespeichert und ist ein Zeilenvektor der Länge n_P. Sie enthält an der i-ten Stelle den Typ des i-ten Punktes: 0 für einen Punkt auf dem Dirichlet-Rand, 1 für einen Punkt im Inneren und 2 für einen Punkt auf dem Neumann-Rand.
- Die *Kante-zu-Punkt*-Liste wird als Feld `k2p` gespeichert und ist eine Matrix der Grösse $3 \times n_S$. Sie enthält in der i-ten Spalte die Indizes der zur i-ten Kante assoziierten Punkte. Der erste assoziierte Punkt ist der Endpunkt der Kante mit dem kleineren Index, der zweite assoziierte Punkt der Endpunkt mit dem grösseren Index und der dritte assoziierte Punkt der Mittelpunkt. Falls der Mittelpunkt noch nicht in den Punktlisten existiert, steht an der dritten Stelle in der Spalte eine 0.
- Die *Kante-zu-Typ*-Liste wird als Feld `k2t` gespeichert und ist ein Zeilenvektor der Länge n_S. Sie enthält an der i-ten Stelle den Typ der i-ten Kante: 0 für eine Kante auf dem Dirichlet-Rand, 1 für eine Kante im Inneren und 2 für eine Kante auf dem Neumann-Rand.
- Die *Kante-zu-Kante*-Liste wird als Feld `k2k` gespeichert und ist eine Matrix der Grösse $3 \times n_S$. Sie enthält in der i-ten Spalte die Indizes der zur i-ten Kante assoziierten Kanten. Die erste assoziierte Kante ist diejenige, welche vom Endpunkt mit dem kleineren Index bis zum Mittelpunkt geht, und die zweite assoziierte Kante ist diejenige, welche vom Endpunkt mit dem grösseren Index bis zum Mittelpunkt geht. Die dritte assoziierte Kante ist jene, welche die Kante selber als erste oder zweite assoziierte Kante hat. Falls die assoziierte Kanten nicht oder noch nicht existieren, so enthält die Spalte statt deren Indizes jeweils eine 0.
- Die *Dreieck-zu-Punkt*-Liste wird als Feld `d2p` gespeichert und ist eine Matrix der Grösse $3 \times n_D$. Sie enthält in der i-ten Spalte die Indizes der zum i-ten Dreieck assoziierten Punkte.

- Die *Dreieck-zu-Kante*-Liste wird als Feld `d2k` gespeichert und ist eine Matrix der Grösse $3 \times n_D$. Sie enthält in der i-ten Spalte die Indizes der zum i-ten Dreick assoziierten Kanten. Die Reihenfolge richtet sich nach der Reihenfolge der assoziierten Punkte; die erste assoziierte Kante ist die zwischen dem ersten und dem zweiten assoziierten Punkt, die zweite assoziierte Kante ist die zwischen dem zweiten und dem dritten assoziierten Punkt und die dritte assoziierte Kante ist die zwischen dem dritten und dem ersten assoziierten Punkt.

- Die *Dreieck-zu-Dreieck*-Liste wird als Feld `d2d` gespeichert und ist eine Matrix der Grösse $5 \times n_S$. Sie enthält in der i-ten Spalte die Indizes der zum i-ten Dreieck assoziierten Dreiecke. Das erste assoziierte Dreieck ist das, welches vom ersten assoziierten Punkt und den Mittelpunkten der ersten und dritten assoziierten Kanten aufgespannt wird, das zweite das, welches vom zweiten assoziierten Punkt und den Mittelpunkten der ersten und zweiten assoziierten Kanten aufgespannt wird, und das dritte das, welches vom dritten assoziierten Punkt und den Mittelpunkten der zweiten und dritten assoziierten Kanten aufgespannt wird. Das vierte assoziierte Dreieck ist das, welches von den Mittelpunkten aller drei assoziierten Kanten aufgespannt wird, und das fünfte assoziierte Dreieck ist jenes, welches das Dreieck selber als erst- bis viertassoziiertes Dreieck hat. Falls die assoziierten Dreiecke nicht oder noch nicht existieren, so enthält die Spalte statt den Indizes jeweils eine 0.

Graphisch ergeben sich für die Assoziierungen *Kante-zu-?* und *Dreieck-zu-?* folgende zwei Schemata:



Für die Assoziierungen *Kante-zu-Kante* und *Dreieck-zu-Dreieck* fehlt in den obigen Schemata jeweils der letzte Eintrag. Wenn wir die erst- und zweit-assoziierten Kanten einer Kante als ihre *Kinderkanten* bezeichnen, dann ist die Kante selber die Kinderkante ihrer dritt-assoziierten Kante. In dem Sinne ist die dritt-assoziierten Kante also die *Elterkante*. Das Analoge gilt, wenn man 'Kante' mit 'Dreieck', 'dritt' mit 'fünft' und 'erst und zweit' mit 'erst bis viert' ersetzt.

Für unser erstes Dreiecksnetz erhalten wir beispielsweise, nach Wahl einer Reihenfolge der Numerierung der Dreiecke und Kanten, folgende acht Listen:

$$\texttt{p2c} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad \texttt{k2p} = \begin{bmatrix} 1 & 2 & 1 & 3 & 1 \\ 2 & 3 & 3 & 4 & 4 \end{bmatrix} \qquad \texttt{d2p} = \begin{bmatrix} 1 & 1 \\ 2 & 3 \\ 3 & 4 \end{bmatrix}$$

$$\texttt{p2t} = \begin{bmatrix} 2 & 2 & 0 & 0 \end{bmatrix} \qquad \texttt{k2t} = \begin{bmatrix} 2 & 2 & 1 & 0 & 2 \end{bmatrix} \qquad \texttt{d2k} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \end{bmatrix}$$

$$\texttt{k2k} = \begin{bmatrix} & & & & \\ & & & & \\ & & & & \end{bmatrix} \qquad \texttt{d2d} = \begin{bmatrix} & \\ & \\ & \\ & \\ & \end{bmatrix}$$

Wir wollen nun annehmen, dass wir eine Liste von Dreiecken haben, welche zu verfeinern sind. Dies kann in Form eines logischen Zeilenvektors der Länge n_D sein, der jeweils den Wert `true` an den Indizes der zu verfeinernden Dreiecke hat, oder eines Vektors, der direkt aus den Indizes der zu verfeinernden Dreiecke besteht. Die acht Listen werden dann wie folgt aufdatiert:

- Wir iterieren über die zu verfeinernden Dreiecke. Für jedes zu verfeinernde Dreieck können wir dabei mit Hilfe der Liste `d2k` die Indizes der drei assoziierten Kanten bestimmen. Mit diesen Indizes und der Liste `k2p` können wir dann weiter bestimmen, ob der Mittelpunkt schon in unserem Gitter existiert, oder ob der Punkt neu aufgenommen werden muss. Die neu aufzunehmenden Punkt vermerken wir dabei sukzessive in `k2p` mit Indizes startend mit $n_P + 1$.
- Nach dieser Iteration haben wir nun berechnet, wie viele neue Mittelpunkte an die Punktlisten angehängt werden müssen und können die Listen `p2c` und `p2t` entsprechend verlängern. Gleichzeitig wissen wir auch, wie viele Dreiecke verfeinert werden und können auch die Listen `d2p`, `d2k` und `d2d` entsprechend verlängern, da jedes zu verfeinernde Dreieck vier Dreiecke anfügen wird.
- Um zu wissen, wie viele neue Kanten die Verfeinerung haben wird, benötigen wir die folgende Überlegung: Eine neue Kante kann nur auf zwei Arten generiert werden. Entweder die Kante geht in einem zu verfeinernden Dreieck von Mittelpunkt zu Mittelpunkt, wobei offenbar jedes zu verfeinernde Dreieck genau drei neue Kanten generiert, oder die Kante geht vom Mittelpunkt zum Eckpunkt. In dem Fall, wo die Kante vom Mittelpunkt zum Eckpunkt geht, wird sie nur dann generiert, wenn auch der Mittelpunkt neu angefügt werden muss. Jeder neue Mittelpunkt erzeugt folglich zwei neue Kanten. Da wir die Anzahl neuer Kanten kennen, können wir also auch die Listen `k2p`, `k2t` und `k2k` entsprechend verlängern.
- Nun kann man die verlängerten Listen anfangen, korrekt zu füllen. Hierzu iteriert man wieder über die zu verfeinernden Dreiecke. Für jedes zu verfeinernde Dreieck gehen wir dabei wie folgt vor: Mit Hilfe der Liste `d2k` bestimmen wir die Indizes der drei assoziierten Kanten. Mit diesen Indizes und der Liste `k2p` können wir dann weiter bestimmen, ob jeder der drei Mittelpunkte neu in `k2p` aufgenommen wurde. Für jeden neu aufgenommenen Mittelpunkt vervollständigen wir zuerst die entsprechenden Spalten der Listen `p2c` und `p2t`, fügen dann die zwei neuen Kanten in den Listen `k2p`, und `k2t` ein und vervollständigen die notwendigen Assoziierungen in `k2k`. Danach fügen wir die drei neuen Kanten, die jeweils von Mittelpunkt zu Mittelpunkt verlaufen, in den Listen `k2p`, und `k2t` ein. Schliesslich können wir die vier neuen Dreiecke in die Listen `d2p` und `d2k` einfügen sowie die notwendigen Assoziierungen in `d2d` vervollständigen.

Wenn wir in unserem ersten Dreiecksnetz das Dreieck $(1, 2, 3)$ verfeinern, so verändert die erste Iteration über die zu verfeinernden Dreiecke die Liste `k2p` wie folgt:

$$\texttt{k2p} = \begin{bmatrix} 1 & 2 & 1 & 3 & 1 \\ 2 & 3 & 3 & 4 & 4 \\ 5 & 6 & 7 & & \end{bmatrix}.$$

Wir wissen dann, dass ein Dreieck verfeinert wird und drei neue Mittelpunkte generiert werden. Daher werden auch neun neue Kanten generiert.

Mit der zweiten Iteration über die zu verfeinernden Dreiecke ergeben sich dann für das Beispiel die folgenden acht Listen:

$$\texttt{p2c} = \begin{bmatrix} 0 & 1 & 1 & 0 & \frac{1}{2} & 1 & \frac{1}{2} \\ 0 & 0 & 1 & 1 & 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

$$\texttt{p2t} = \begin{bmatrix} 2 & 2 & 0 & 0 & 2 & 2 & 1 \end{bmatrix}$$

$$\texttt{d2p} = \begin{bmatrix} 1 & 1 & 1 & 2 & 3 & 6 \\ 2 & 3 & 5 & 6 & 7 & 7 \\ 3 & 4 & 7 & 5 & 6 & 5 \end{bmatrix}$$

$$\texttt{d2k} = \begin{bmatrix} 1 & 3 & 6 & 8 & 11 & 13 \\ 2 & 4 & 14 & 12 & 13 & 14 \\ 3 & 5 & 10 & 7 & 9 & 12 \end{bmatrix}$$

$$\texttt{k2p} = \begin{bmatrix} 1 & 2 & 1 & 3 & 1 & 1 & 2 & 2 & 3 & 1 & 3 & 5 & 6 & 5 \\ 2 & 3 & 3 & 4 & 4 & 5 & 5 & 6 & 6 & 7 & 7 & 6 & 7 & 7 \\ 5 & 6 & 7 & & & & & & & & & & & \end{bmatrix}$$

$$\texttt{k2t} = \begin{bmatrix} 2 & 2 & 1 & 0 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\texttt{k2k} = \begin{bmatrix} 6 & 8 & 10 & & & & & & & & & & & \\ 7 & 9 & 11 & & & & & & & & & & & \\ & & & & & 1 & 1 & 2 & 2 & 3 & 3 & & & \end{bmatrix}$$

$$\texttt{d2d} = \begin{bmatrix} 3 & & & & & \\ 4 & & & & & \\ 5 & & & & & \\ 6 & & & & & \\ & & 1 & 1 & 1 & 1 \end{bmatrix}$$

In diesen Listen kann man in `k2k` und `d2d` anfangen, die hierarchische Baum-Struktur zu sehen. Spezifisch können wir in `k2k` sehen, dass die ersten fünf und letzen drei Kanten jeweils Wurzelkanten sind, da sie keine Elternkante haben, während die sechste bis elfte Kante die ersten drei Kanten als Elternkante besitzen. Dabei fällt auf, dass durch Verfeinerungen neue Wurzelkanten generiert werden. Dementgegen ist die Situation in `d2d` einfacher, da immer nur die initial vorhandenen zwei Wurzeldreiecke vorhanden sein können. Da wir allerdings mehr als eine Wurzel haben können, würden `k2k` und `d2d`, jeweils seperat betrachtet, in der Informatik als *Wälder* bezeichnet werden.

Das effektiv resultierende *aktive* Dreiecksnetz aus dieser Darstellung enthält man auch, indem man die hierarchische Baum-Struktur aus der Liste`d2d` verwendet: Das zugehörige *aktive* Dreiecksnetz ergibt sich aus allen *Blattdreiecken*; hierbei ist ein Blattdreieck ein Dreieck, welches keine Kinderdreiecke besitzt. In unserem Beispiel sind alle ausser dem ersten Dreieck Blattdreiecke.

Aufgabe 1.

Schreiben Sie eine Funktion

```
function hmesh = hmesh_initialise(mesh),
```

welches ein Dreiecksnetz, `mesh`, der Form der Programmierblätter nimmt und das entsprechende initiale hierarchische Dreiecksnetz als MATLAB `struct`, `hmesh`, bestehend aus den acht Listen, zurückgibt.

Aufgabe 2.

Schreiben Sie eine Funktion

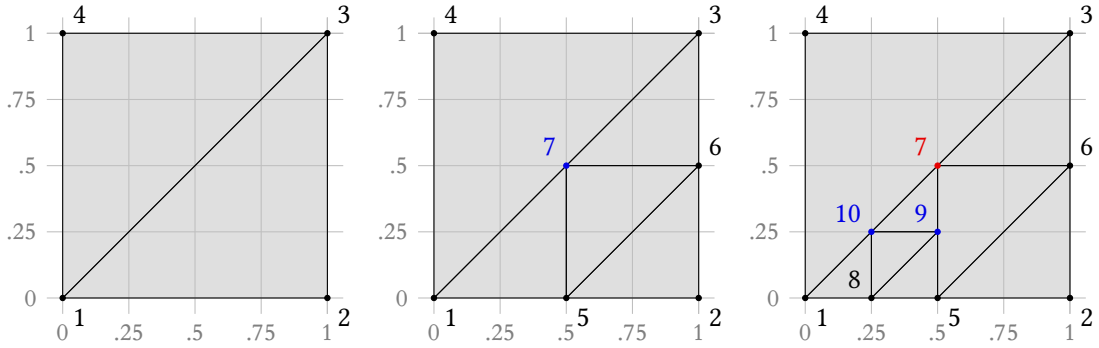
```
function hmesh = hmesh_refine_marked(hmesh, marked),
```

welches ein hierarchisches Dreiecksnetz, `hmesh`, sowie einen logischen Zeilenvektor, `marked`, der jeweils den Wert ‘wahr’ an den zu verfeinernden Dreiecke besitzt, als Input nimmt. Die zu verfeinernden Dreiecke müssen dabei Blattdreiecke sein (ansonsten kann man sie ignorieren, da sie schon verfeinert sind). Die Funktion gibt das entsprechend verfeinerte hierarchische Dreiecksnetz wieder als `hmesh` zurück.

Hängende Knoten und einfach hängende Knoten

In den beschriebenen *aktiven* Dreiecksnetzen können (und sollen auch) *hängende* Knoten erscheinen, das heisst Punkte im aktiven Dreiecksnetz, welche sich auf einer inneren Kante des aktiven Dreiecksnetzes befinden. Wenn wir auf dem aktiven Dreiecksnetz eine stetige, stückweise lineare Funktion definieren wollen, dann dürfen wir an diesen Punkten keinen Wert vorgeben, sondern dieser muss sich aus den Werten anderer Punkte ergeben.

Zum Beispiel ist die Situation in unserer Sequenz von hierarchischen Dreiecksnetzen wie folgt:



Im ersten dargestellten Dreiecksnetz hat das aktive Dreicksnetz keine hängenden Knoten. Im zweiten dargestellten Dreiecksnetz hat das aktive Dreicksnetz den hängenden Knoten 7, dessen Wert sich als Mittel der Werte in den Knoten 1 und 3 ergibt. Im dritten dargestellten Dreiecksnetz hat das aktive Dreicksnetz die hängenden Knoten 7, 9 und 10. Der Wert im Knoten 7 ist wieder als Mittelwert der Werte in den Knoten 1 und 3 gegeben, der Wert im Knoten 9 als Mittelwert der Werte in den Knoten 5 und 7 und der Wert im Knoten 10 als Mittelwert der Werte in den Knoten 1 und 7. Wie wir sehen, kann somit der Wert bei einem hängende Knoten vom Wert bei einem anderen hängenden Knoten abhängen. Ein hängender Knoten der nur von nicht-hängenden Knoten abhängt, nennnen wir *einfach* hängend.

Wenn wir an die linearen Finite Elemente denken, welche wir auf dem aktiven Dreiecksnetz eines hierarchischen Dreiecksnetzes anwenden wollen, ist es klar, dass ein einfach hängender Knoten einfach aus dem Gleichungssystem zu eliminieren ist (siehe Übungsblatt 8, Aufgabe 1). Dementgegen kann die Eliminiation eines hängende Knoten, welcher nicht einfach hängend ist, grundsätzlich beliebig schwierig sein. Wir wollen daher unsere hierarchischen Dreiecksnetze jeweils so weiter verfeinern, dass im aktiven Dreiecksnetz nur einfach hängende Knoten vorkommen.

Wenn wir nun annehmen, dass wir ein hierarchisches Dreiecksnetz haben, welches nur einfach hängende Knoten hat, und eine Liste von Dreiecken haben, welche zu verfeinern sind, dann müssen wir beim Verfeinern nur aufpassen, dass wir keinen hängenden Knoten erzeugen, der nicht einfach hängend ist. Um dies sicherzustellen zu können, reicht es, die Liste von zu verfeinernden Dreiecke wie folgt zu erweitern:

- Wir iterieren zuerst einmal über alle aktiven Dreiecke. Für jede Kante eines aktiven Dreiecks überprüfen wir dabei, ob sie Kinderkanten hat. Falls die Kante Kinderkanten besitzt, so assoziieren wir den Index des Dreiecks mit diesen Kinderkanten.

- Dann iterieren wir über alle zu verfeinernden Dreiecke. Für jede Kante des Dreiecks, welche einen assoziierten Dreiecksindex hat, wird dieses assoziierte Dreieck ebenfalls zur Verfeinerung markiert und dann rekursiv überprüft, ob es Kanten hat, welche einen assoziierten Dreiecksindex besitzen, die ebenfalls zur Verfeinerung markiert werden müssen.

Aufgabe 3.

Schreiben Sie eine Funktion

```
function hangs = hmesh_compute_hanging(hmesh),
```

welche ein hierarchisches Dreiecksnetz, `hmesh`, nimmt und einen logischen Zeilenvektor, `hangs`, zurückgibt, der jeweils den Wert 'wahr' an den Stellen enthält, welche Punkte entsprechen, die hängende Knoten sind.

Aufgabe 4.

Schreiben Sie eine Funktion

```
function marked = hmesh_complete_marking(hmesh, marked),
```

welche ein hierarchisches Dreiecksnetz, `hmesh`, welches nur einfach hängende Knoten hat, sowie einen logischen Zeilenvektor, `marked`, der jeweils den Wert 'wahr' an den zu verfeinernden Dreiecken besitzt, als Input nimmt. Als Output wird der logischen Zeilenvektor, `marked`, ausgegeben, der so erweitert wurde, dass die Verfeinerung des Dreiecksnetzes wieder nur einfach hängende Knoten besitzen wird.

Lineare Finite Elemente

Mit dem aktiven Dreiecksnetz eines hierarchischen Dreiecksnetzes mit nur einfach hängenden Knoten ist es nun nicht mehr schwer, analog wie auf Programmierblatt 2 die linearen Finite Elemente zu implementieren. Dabei betrachtet man zuerst alle Punkte des Netzes als Freiheitsgrade. Danach geht man wie folgt vor, um die hängenden Knoten aus der Steifigkeits- und Massenmatrix zu eliminieren:

- Man enumeriert zuerst nur die nicht-hängenden Knoten.
- Dann erstellt man eine dünnbesetzte Matrix $\mathbf{C}$, welche die Enumerierung der nicht-hängenden Knoten auf die aller Knoten abbildet inklusive der algebraischen Einschränkung bei den hängenden Knoten.
- Indem man die Steifigkeits- und Massenmatrix mit hängenden Knoten von links mit $\mathbf{C}^T$ und von rechts mit $\mathbf{C}$ multipliziert, werden die hängenden Knoten eliminiert.

A-posteriori-Fehlerschätzer

Hat man die approximative Lösung der Poisson-Gleichung auf dem aktiven Dreiecksnetz eines hierarchischen Dreiecksnetzes mit nur einfach hängenden Knoten, so ist es nun einfach, den aus der Vorlesung bekannten A-posteriori-Fehlerschätzer zu implementieren.

- Zuerst muss man nur über die aktiven Dreiecke iterieren und die flächenbezogenen Residuen berechnen. Gleichzeitig speichert man zu jeder Kante des Dreiecks die nach innen gerichtete Normalableitung der Lösung.
- Danach iteriert man über alle Kanten der aktiven Dreiecke und berechnet deren kantenbezogenen Sprung.
- Indem man erneut über die aktiven Dreiecke iteriert, kann man für jedes aktive Dreieck den totalen lokalen Fehlerschätzer berechnen.
- Schliesslich wählt man einen festen Prozentsatz der grössten lokalen Fehlerschätzer und markiert deren Dreiecke für eine Verfeinerung.