ulm university universität
uulm

Institut für Numerische Mathematik

WiMa-Praktikum II

Adaptive Gitter-Verfeinerung

PROJEKTARBEIT

Christian Aßfalg
Pelin Ekmekci
Christian Kaps

16. Dezember 2015

Inhaltsverzeichnis

1 Einleitung

Bei der numerischen Lösung von partiellen Differentialgleichungen mit der Finite Elemente Methode wird das Gebiet auf dem die Lösung berechnet wird typischerweise durch Triangulierung diskretisiert. Je nach Feinheit dieser Diskretisierung verändert sich der Diskretisierungsfehler – bei zu grobem Gitter sind die Fehler zu groß. Um die Diskretisierungsfehler zu verringern muss das Gitter also verfeinert werden. Allerdings steigt mit zunehmender Feinheit der Diskretisierung auch der Rechenaufwand. Daher soll das Gitter idealerweise nur dort verfeinert werden, wo der Fehler groß ist, um einen Kompromiss aus Laufzeit und Fehler zu erhalten.

Zur Bestimmung des lokalen Fehlers wird ein Fehlerindikator für jedes Gitterelement bestimmt. Basierend auf diesem Fehler werden dann mittels einer Markierungsstrategie die zu verfeinernden Elemente ausgewählt.

Diese markierten Elemente werden dann verfeinert, wodurch jedoch sogenannte *hängende Knoten* entstehen können (siehe Abbildung 1). Hängende Knoten zerstören die Triangulation, da die die Nachbarelemente eines verfeinerten Elements dann aus 4 Kanten bestehen. Das typische Vorgehen einer Annäherung der Lösungsfunktion mit stückweise linearen Basisfunktionen ist in diesem Fall nicht mehr möglich: bei hängenden Knoten können die Basisfunktionen nicht mehr unabhängig voneinander gewählt werden.

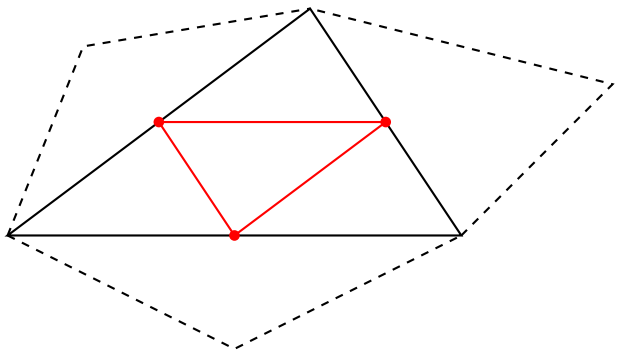


Abbildung 1: Entstehung hängender Knoten durch Rot-Verfeinerung

In unserem Fall wird das Problem der hängenden Knoten durch eine weitere Verfeinerung der Dreiecke mit hängenden Knoten gelöst, bis es keine hängenden Knoten mehr gibt. Eine Verfeinerungsstrategie, welche die Dreiecksstruktur des Gitters erhält, ist die sogenannte *RGB-Verfeinerung (Rot-, Grün-, Blau-Verfeinerung).*

Im Rahmen dieses Projektes wird die RGB-Verfeinerung in MATLAB implementiert (siehe Anhang A) und der Anschaulichkeit halber auf Grauwertbildern getestet (siehe Abschnitt 4).

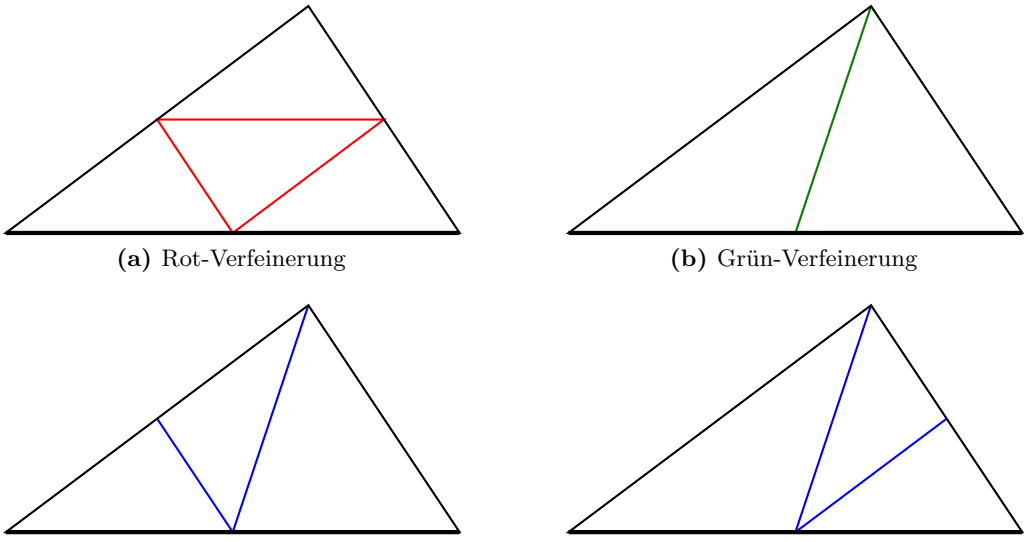
2 Algorithmus zur Verfeinerung

2.1 Markierungsstrategie

Bei der Anwendung der RGB-Verfeinerung auf Grauwertbilder dient die maximale Differenz der Grauwerte innerhalb eines Gitterelements als Fehlerindikator. Wir wollen in diesem Projekt verschiedene Strategien zur Markierung der zu verfeinernden Element betrachten:
Für eine Triangulierung $\mathcal{T}$ sei der Vektor $\eta = (\eta_T)$, der für das Element $T \in \mathcal{T}$ den Fehlerindikator η_T enthält, gegeben.

Strategien zum Markieren der zu verfeinernden Elemente:

- Strategie 1(Absolutes Kriterium):
  Verfeinere alle Dreiecke $T \in \mathcal{T}$, für die gilt: $\eta_T \geq \theta, \quad \theta \in \mathbb{R}$.
- Strategie 2 (Relatives Kriterium):
  Verfeinere alle Dreiecke $T \in \mathcal{T}$, für die gilt: $\eta_T \geq \theta \cdot \max_{T' \in \mathcal{T}} \eta_{T'}, \quad \theta \in [0, 1]$.
- Strategie 3 (Dörfler-Marking/Bulk-Chasing):
  Suche eine minimale Menge $\mathcal{M} \subset \mathcal{T}$, für die gilt: $\theta \sum_{T \in \mathcal{T}} \eta_T^2 < \sum_{T \in \mathcal{M}} \eta_T^2, \quad \theta \in [0, 1]$.
  Verfeinere dann alle $T \in \mathcal{M}$.



(a) Rot-Verfeinerung **(b)** Grün-Verfeinerung

(c) Blau-Verfeinerung (1) **(d)** Blau-Verfeinerung (2)

Abbildung 2: Verfeinerungsvarianten (längste Seite dicker gezeichnet)

2.2 RGB-Verfeinerung

Zur Beschreibung der RGB-Verfeinerung gibt es mehrere äquivalente Algorithmen (siehe [1] und [2]). Wir haben uns für die Variante aus [1] entschieden, da sie aus unserer Sicht einfacher zu implementieren ist:

1. Markiere die Kanten aller markierten Elemente.
2. Solange noch weitere Kanten markiert werden:
   a) Ist für ein Element eine Kante markiert, dann markiere auch die längste Kante.
3. Verfeinere jedes Element gemäß der Anzahl der jeweils markierten Kanten (siehe Abbildung 2):
   a) 3 markierte Kanten: Das Element wird rot-verfeinert.
   b) 2 markierte Kanten: Das Element wird blau-verfeinert. (Die Blau-Verfeinerung besitzt zwei Varianten, je nachdem welche Kanten markiert sind.)
   c) 1 markierte Kante: Das Element wird grün-verfeinert.

Die durch die Markierungsstrategie markierten Elemente werden somit rot-verfeinert und die Entstehung hängender Knoten wird durch die zusätzliche rot-, blau- bzw. grün-Verfeinerung nicht markierter Elemente verhindert (siehe Abbildung 3).

Im Anschluss an die RGB-Verfeinerung wird die Fehlerschätzung mit dem verfeinerten Netz durchgeführt und gegebenenfalls werden entsprechend der Markierungsstrategie erneut Elemente zur Rot-Verfeinerung markiert. Dies wird iterativ so lange wiederholt, bis keine weiteren Elemene mehr markiert werden, oder eine maximale Anzahl an Iterationen erreicht ist.

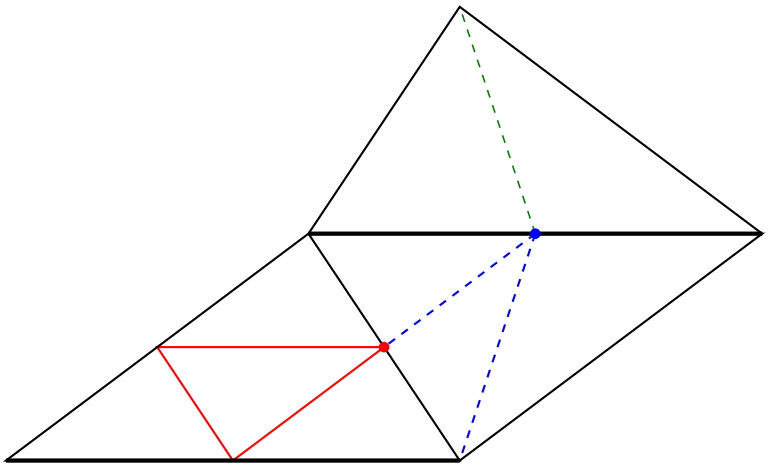


Abbildung 3: Anwendung der RGB-Verfeinerung (längste Seite dicker gezeichnet)

3 Umsetzung in Matlab

Im folgenden wird beschrieben, wie die RGB-Verfeinerung in MATLAB umgesetzt wurde. Die zugehörigen Quelltexte befinden sich im Anhang A.

3.1 Datenstruktur

Zur Beschreibung des zweidimensionalen Netzes wird folgende Datenstruktur verwendet:

- `coordinates` $\in \mathbb{R}^{N_c \times 2}$ enthält die Koordinaten der Netzpunkte
- `elements` $\in \mathbb{N}^{N_e \times 3}$ enthält die drei Eckpunte eines Elementes (Dreiecks) als Index in `coordinates`
- `boundary` $\in \mathbb{N}^{N_b \times 2}$ enthält die Anfangs- und Endpunkte einer Randkante als Index in `coordinates`

Hierbei bezeichnet

- N_c die Anzahl an Eckpunkten.
- N_e die Anzahl an Elementen (Dreiecken).
- N_b die Anzahl an Kanten auf dem Rand des Gebietes.

Diese drei Matrizen beschreiben das Netz vollständig. Um jedoch schneller die passenden Elemente zu finden werden folgende Hilfsmatritzen erstellt:

- `edge2nodes` $\in \mathbb{R}^{N_b \times 2}$ Welche Eckpunkte gehören zu einer Kante?
- `element2edges` $\in \mathbb{R}^{N_e \times 2}$ Welche Kanten gehören zu einem Element?
- `boundary2edges` $\in \mathbb{R}^{N_b \times 2}$ Welche Kanten gehören zum Rand?

Diese Matrizen stellen eine Nummerierung der Elemente und der Kanten dar. Die Hilfsmatrizen werden mit der Funktion `provideGeometricData` (siehe Listing 4) erzeugt.

3.2 Verfeinerung

Die Umsetzung der drei Markierungskriterien aus Abschnitt 2.1 kann den Listings 1-3 entnommen werden. Die Markierungsfunktionen erhalten jeweils den Fehler η_T und den Parameter θ. Der Rückgabewert ist ein Vektor, der die Indizes der zu verfeinernden Elemente enthält. Der Funktionsaufruf für das absolute Markierungskriterium lautet dann: `marked=markElements_absolute(eta,theta)`

Die RGB-Verfeinerung wird dann durch die Funktion `[coordinates, elements, boundary] = refineRGB(coordinates, elements, boundary, marked)` (siehe Listing 5) durchgeführt, welche die Datenstruktur und den Vektor der zur Rot-Verfeinerung markierten Elemente erhält und die vollständige Datenstruktur des verfeinerten Gitters zurück liefert.

4 Ergebnisse

Um zu überprüfen, ob die Gitterverfeinerung korrekt implementiert wurde, wird die Gitterverfeinerung auf ein Bild des Ulmer Münsters angewendet. Dieses wird zunächst in ein Graustufenbild mit 256 Graustufen umgewandelt (siehe Abbildung 4). Als Fehler η_T wird der maximale Unterschied der Grauwerte innerhalb eines Gitterelements berechnet.

Abbildung 4: `muenster.jpg` farbig und in 256 Graustufen

Auf das Anfangsgitter (siehe Abbildung 5) wird dann die RGB-Verfeinerung angewendet, im folgenden Beispiel zunächst mit Dörfler-Markierung ($\theta = 0.8$).

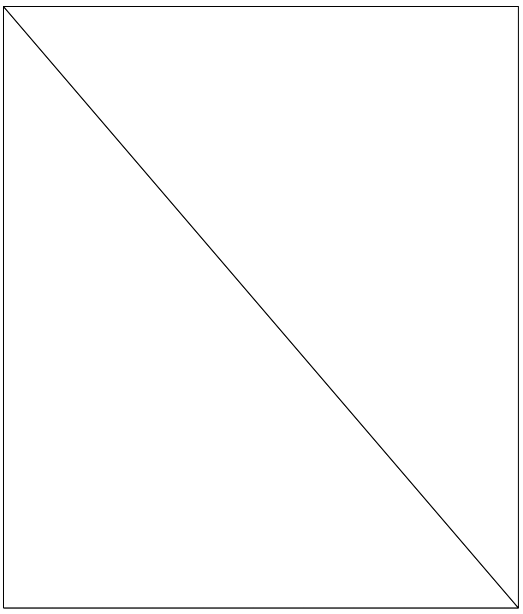
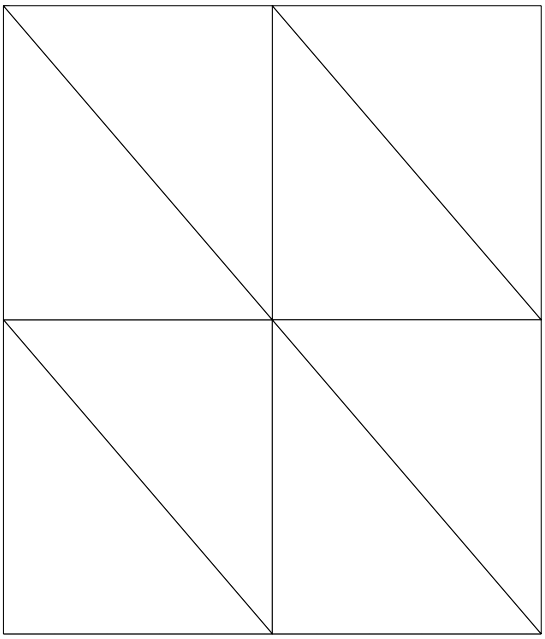
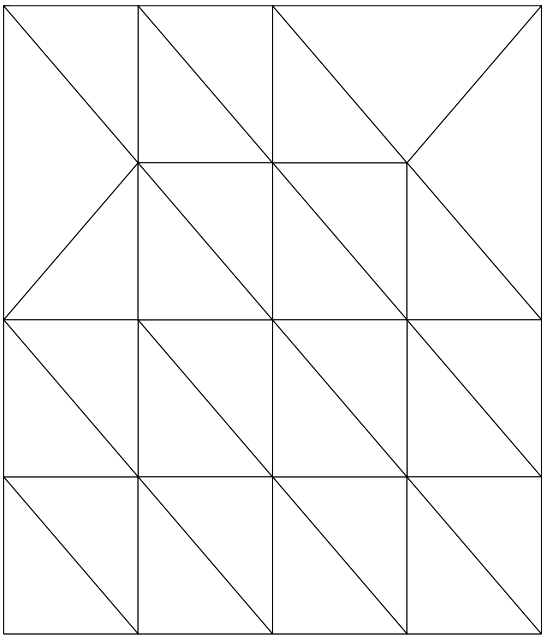


Abbildung 5: Anfangsgitter

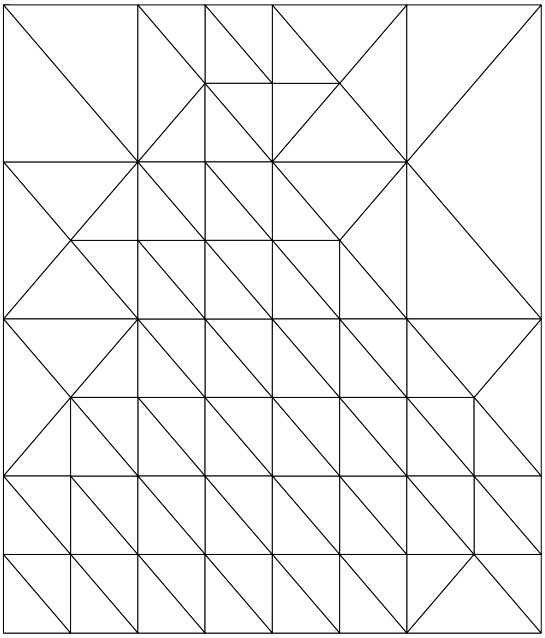
Iteration 1



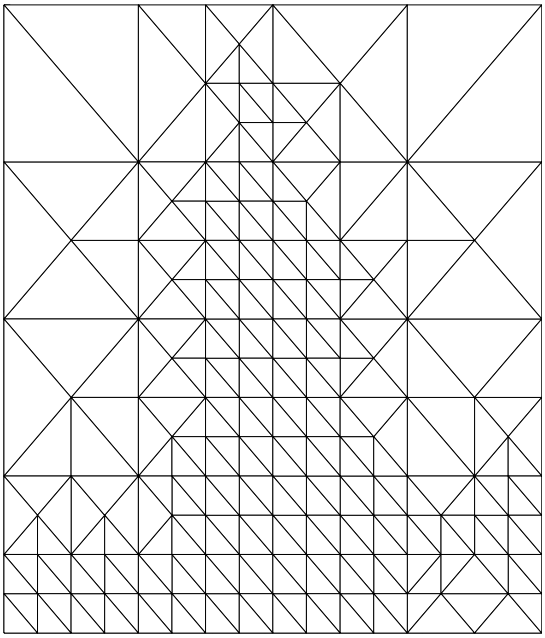
Iteration 2



Iteration 3



Iteration 4



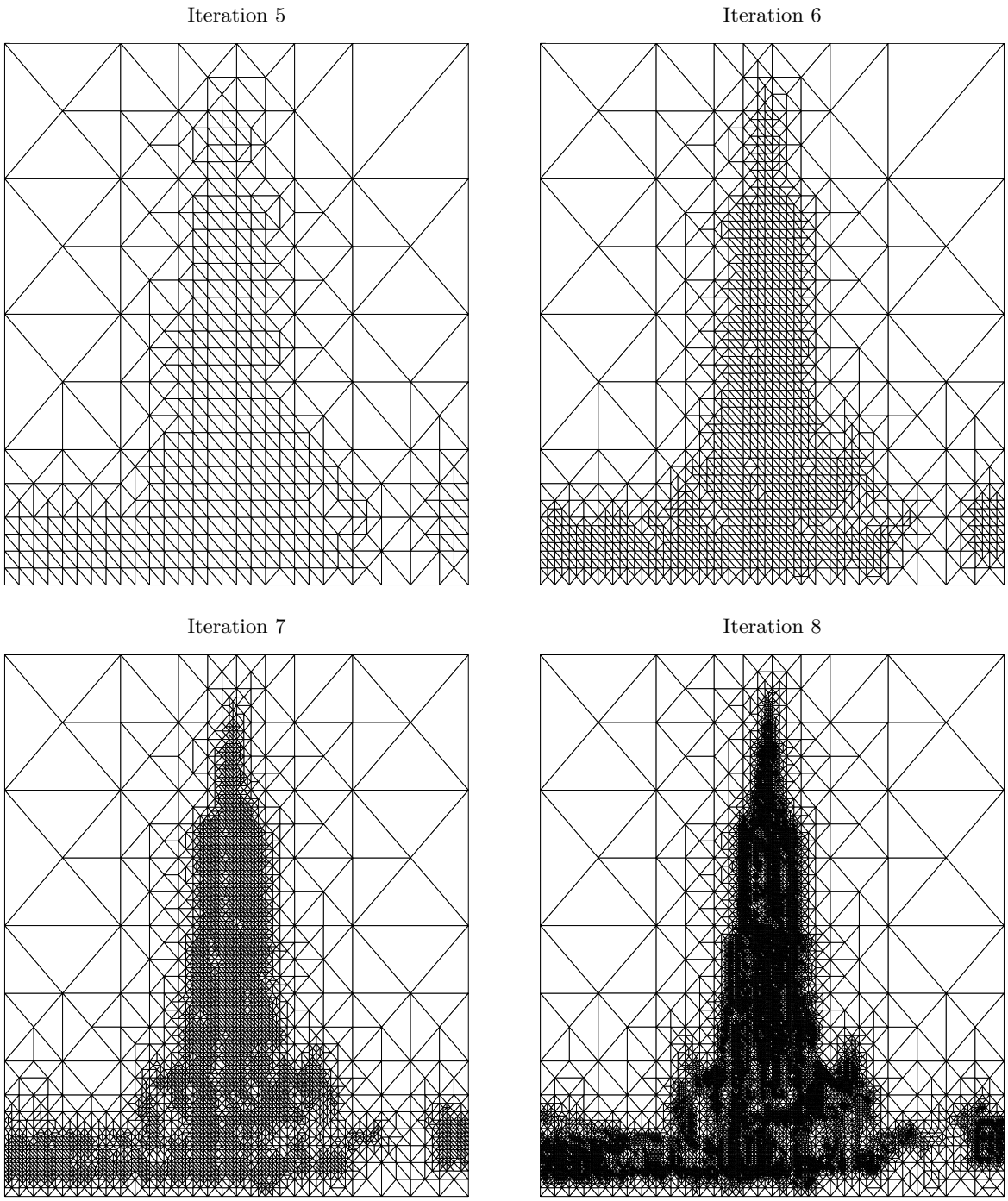
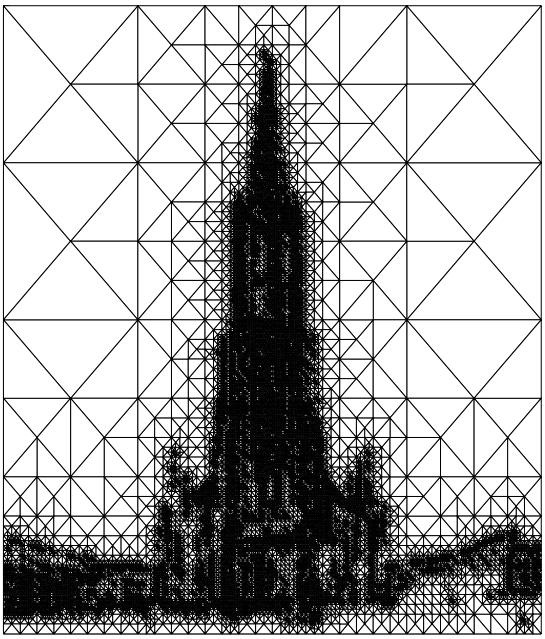


Abbildung 6: RGB-Verfeinerung mit Dörfler-Marking ($\theta = 0.8$)

absolutes Kriterium ($\theta = 120$)



relatives Kriterium ($\theta = 0.8$)

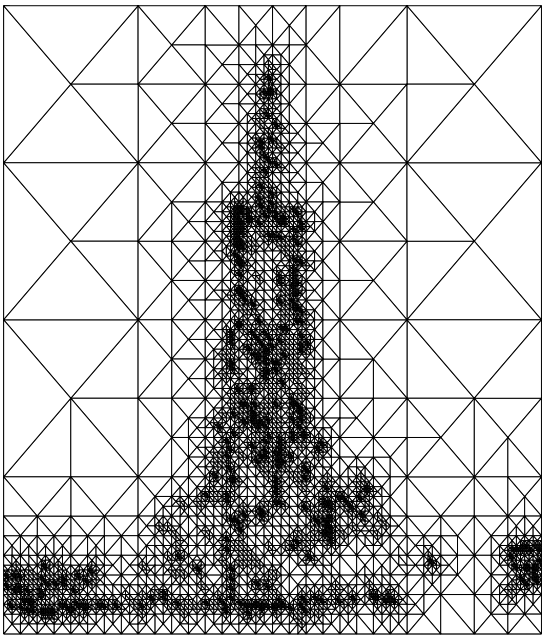


Abbildung 7: RGB-Verfeinerung mit weiteren Markierungsstrategien nach 8 Iterationen

Anhand der Abbildung 6 lässt sich die Funktionsweise der RGB-Verfeinerung gut nachvollziehen. Bei der ersten Iteration werden die beiden Elemente des Ausgangsgitters jeweils rot-verfeinert. Bereits bei der zweiten Iteration werden nicht mehr alle Elemente zur Rot-Verfeinerung markiert. Hier treten zusätzlich eine Blau- und eine Grün-Verfeinerung auf. Nach 8 Iterationen ist der Münsterturm deutlich zu erkennen. Der Himmel hat einen gleichmäßigen Grauwertverlauf und muss daher weniger verfeinert werden als das Ulmer Münster, bei dem sehr hohe Kontraste auftreten. Folglich ist das Gitter im Bereich des Himmels sehr grob und im Bereich des Münsters sehr fein.

Wie erwartet entstehen keine hängenden Knoten. Die Anzahl der entstehenden Elemente hängt von der Markierungsstrategie und der Wahl der jeweiligen Parameter ab (siehe Tabelle 1). Während das absolute Kriterium terminiert, sobald der größte absolute Fehler kleiner wird als θ, laufen die relativen Kriterien so lange, bis der Fehler, also die Abweichung der Grauwerte jedes Elements, null ist. Daher sollte man die Anzahl der Iteration beschränken.

Markierungsstrategie	Anzahl der Gitterelemente
absolut ($\theta = 120$)	25 372
relativ ($\theta = 0.8$)	9 088
Dörfler-Marking ($\theta = 0.8$)	22 918

Tabelle 1: Anzahl der erzeugten Gitterelemente nach 8 Iterationen

Literatur

[1] Funken, Stefan ; Bantle, Andreas: *Numerik partieller Differentialgleichungen – Übungsblatt 7*. Wintersemester 13/14 https://www.uni-ulm.de/fileadmin/website_uni_ulm/mawi.inst.070/ws13_14/NumerikPDEs/Blatt7.pdf

[2] Lebiedz, Dirk ; Radic, Mladjan: *WiMa-Praktikum II: Adaptive Gitter-Verfeinerung*. Wintersemester 15/16 http://www.uni-ulm.de/fileadmin/website_uni_ulm/mawi.inst.070/ws15_16/WiMa-Praktikum/Project5.pdf

A Code

Listing 1: markElements_absolute.m

```
function marked = markElements_absolute(eta,theta)
    marked=find(eta>=theta);
end
```

Listing 2: markElements_max.m

```
function marked = markElements_max(eta,theta)
    marked=find(eta>=theta*max(eta));
end
```

Listing 3: markElement_Doerfler.m

```
function marked = markElements_Doerfler(eta,theta)
    [eta2,I]=sort(eta.^2,'descend');
    C=cumsum(eta2);
    x=theta*C(end);
    m=find(C>x,1,'first');
    marked=I(1:m);
end
```

Listing 4: provideGeometricData.m

```
function [edge2nodes,element2edges,boundary2edges] ...
    = provideGeometricData(elements,boundary)
    edge2nodes=unique(sort([elements(:,[1,2]);elements(:,[2,3]);...
        elements(:,[1,3])],2),'rows');
    [~,ib1]=ismember(sort(elements(:,[1,2]),2),edge2nodes,'rows');
    [~,ib2]=ismember(sort(elements(:,[2,3]),2),edge2nodes,'rows');
    [~,ib3]=ismember(sort(elements(:,[1,3]),2),edge2nodes,'rows');
    element2edges=[ib1,ib2,ib3];
    [~,boundary2edges]=ismember(sort(boundary,2),edge2nodes,'rows');
end
```

Listing 5: refineRGB.m

```
function [coordinates,elements,boundary] ...
    = refineRGB(coordinates,elements,boundary,marked)

[edge2nodes,element2edges,boundary2edges] = ...
    provideGeometricData(elements,boundary);
[elements,element2edges] = ...
    sortElements2Edges(coordinates,elements,element2edges,edge2nodes);
Nc=size(coordinates,1);
Ned=size(edge2nodes,1);
Ne=size(elements,1);

% Kanten markieren
markedEdges=unique(element2edges(marked,:));
c=0;
while(c~=length(markedEdges))
    c=length(markedEdges);
```

```
    tmp=ismember(element2edges,markedEdges);
    tmp=tmp(:,2) | tmp(:,3);
    markedEdges=union(markedEdges,element2edges(tmp,1));%laengste
       Kante markieren
end

% Mittelpunkte von markierten Kanten ausrechnen
MP=coordinates(edge2nodes(markedEdges,1),:)/2+ ...
    coordinates(edge2nodes(markedEdges,2),:)/2;
coordinates=[coordinates;MP]; %Mittelpunktskoordinaten hinten an
   coordinates dranhaengen
edge2MP=zeros(Ned,1);
edge2MP(markedEdges)=Nc+1:size(coordinates,1);% Fuer jede markierte
   Kante Index auf deren Mittelpunkt

% Boundary anpassen
boundary=sort(boundary,2);
tmp=ismember(boundary2edges,markedEdges);
for i=find(tmp)'
    edge=boundary2edges(i);
    boundary(end+1,:)=[edge2MP(edge),boundary(i,2)];
    boundary(i,2)=edge2MP(edge);
end

% Verfeinerung
tmp=ismember(element2edges,markedEdges); %Ermittelt fuer jedes
   Element welche seiner Kanten markiert sind
tmp2=2*tmp(:,2)+3*tmp(:,3);%Welche ausser der laengsten Kante ist
   noch markiert.(wird nur fuer BLAU benoetigt)

for i=1:Ne % Alle Elemente durchlaufen
    switch nnz(tmp(i,:)) %Anzahl markierter Kanten des i-ten Elements
        case 3 %ROT
            P=elements(i,:);
            Pm=edge2MP(element2edges(i,:));
            elements(i,:)=Pm;
            elements(end+3,:)=[P(1),Pm(1),Pm(3)];
            elements(end-1,:)=[P(2),Pm(1),Pm(2)];
            elements(end-2,:)=[P(3),Pm(2),Pm(3)];
        case 2 %BLAU
            P=elements(i,:);
            ib=tmp2(i);
            Pm=edge2MP(element2edges(i,[1,ib]));

            elements(i,:)=[P(3),Pm(1),Pm(2)];
            switch ib
                case 2
                    elements(end+2,:)=[P(1),Pm(1),P(3)];
                    elements(end-1,:)=[P(2),Pm(1),Pm(2)];
                case 3
                    elements(end+2,:)=[P(1),Pm(1),Pm(2)];
                    elements(end-1,:)=[P(2),Pm(1),P(3)];
            end
        case 1 %GRUEN
```

```
            P=elements(i,:);
            Pm=edge2MP(element2edges(i,1)); %Mittelpunkt der
               laengsten Kante
            elements(i,:)=[P(1),Pm,P(3)];
            elements(end+1,:)=[P(2),Pm,P(3)];
    end
end
end

%Laenge jeder Kante ermitteln
function len = edgeLength(edges2nodes, coordinates)
    diff=coordinates(edges2nodes(:,1),:)- ...
         coordinates(edges2nodes(:,2),:);
    len=sum(diff.^2,2); %Quadrat der zeilenweisen euklidischen Norm
end

%elements2edges und elements so sortieren, dass die laengste
%Kante jedes Elements vorne ist.
function [elements,element2edges] = ...
    sortElements2Edges(coordinates,elements,element2edges,edge2nodes)

len=edgeLength(edge2nodes, coordinates);
for i=1:size(element2edges,1)
    [~,j]=max(len(element2edges(i,:)));
    element2edges(i,:)=circshift(element2edges(i,:),[0,-j+1]);
    elements(i,:)=circshift(elements(i,:),[0,-j+1]);
end
end
```