ulm university universität
uulm

Universität Ulm
Fakultät für Mathematik und Wirtschaftswissenschaften

Effiziente Realisierung der bedingten Delaunay Triangulierung in Matlab und C

Wissenschaftliche Arbeit im Fach Mathematik

vorgelegt von
Anna Jostes
am 29. März 2012

Gutachter

Prof. Dr. S. Funken

Inhaltsverzeichnis

Einleitung

In der Numerik betrachtet man das Lösen mathematischer Problemstellungen mit Hilfe von Computern. Da ein Computer nicht exakt rechnen kann, müssen diese Probleme *diskretisiert* werden.
Häufig genügt eine Lösung des Problems auf einem endlichen Gebiet. Um dieses zu diskretisieren, werden Netze aus Polygonen über das Gebiet gelegt. Dabei hat es sich als sinnvoll erwiesen, ein Netz aus Dreiecken zu wählen. Wir sprechen in diesem Fall von einer *Triangulierung* des Gebietes. Eine solche Triangulierung kann unterschiedlichen Anforderungen genügen, etwa einer möglichst kleinen Gesamtanzahl von Dreiecken. Die von uns betrachtete *Delaunay-Triangulierung* hat das Ziel, Dreiecke mit sehr kleinen Winkeln innerhalb der Triangulierung zu vermeiden. Wir werden zusätzlich fordern, dass keine Kante der Triangulierung eine bestimmte Länge überschreitet und kein Winkel einen Minimalwinkel unterschreitet.

Ziel dieser Arbeit ist die Implementierung und Beschreibung eines Programmes, das ein solches Dreiecksnetz über ein durch einen Polygonzug begrenztes Gebiet legt.

Im ersten Kapitel beschäftigen wir uns mit dem Bestimmen einer Triangulierung zu einer gegebenen Punktemenge. Die hierbei entstehende Triangulierung erfüllt im Allgemeinen keine unserer Anforderungen und wird lediglich als Ausgangssituation für verbessernde Algorithmen verwendet.
Das zweite Kapitel beschreibt einen Algorithmus, mit dem eine bestehende Triangulierung ohne das Einfügen zusätzlicher Punkte optimiert werden kann.
Im dritten Kapitel verändern wir die bisher bestehende Triangulierung so, dass sie unseren Anforderungen genügt.
Jedes dieser Kapitel beginnt mit der Erarbeitung der benötigten Theorie. Anschließend werden wir auf die Implementierung der verschiedenen Funktionen eingehen. Dabei motivieren wir zunächst die einzelnen Funktionen. Schließlich wird der verwendete Quellcode angegeben und erläutert.
Im letzten Kapitel betrachten wir das Programm, welches alle besprochenen Funktionen sukzessive aufruft. Anschließend wird ein Fazit die Arbeit beenden.

Die fertige Implementierung des Programms findet sich auch auf der beigelegten CD.

1 Bedingte Triangulierung

Wir beschäftigen uns in diesem Kapitel zunächst mit dem Begriff der (bedingten) Triangulierung. Anschließend wird mit dem *planesweep*-Algorithmus ein Vorgehen beschrieben, wie eine solche Triangulierung bestimmt wird.

Als Quelle wurde [Ed2001] Kapitel 2 verwendet.

1.1 Theoretische Grundlagen

Wir werden zu Beginn einige Bezeichnungen und Notationen festlegen, die in der restlichen Ausarbeitung verwendet werden.

Definition 1.1 (Punkt, Kante, Dreieck) Gegeben sei eine Teilmenge P des $\mathbb{R}^2$. Ein Element aus P nennen wir *Punkt*.
Ein Element aus $P^2 := P \times P$ nennen wir *Kante*. Eine Kante wird nach ihrem Anfangs- und Endpunkt benannt: Die Kante AB verbindet die beiden Punkte A und B aus P.
Ein Element aus $P^3 := P \times P \times P$ nennen wir ein *Dreieck*. Ein Dreieck wird nach seinen Eckpunkten benannt: Das Dreieck ABC wird aufgespannt durch die paarweise verschiedenen Eckpunkte A, B und C aus P. Zu einem Dreieck ABC gehören auch die Kanten AB, BC und CA.

Definition 1.2 (Triangulierung) Unter einer Triangulierung T von P verstehen wir eine Teilmenge aller möglichen Dreiecke P^3 derart, dass gilt:

(i) Alle Punkte aus P sind Teil der Triangulierung, es gibt also zu jedem $A \in P$ mindestens ein Dreieck, dass A als Eckpunkt hat.

(ii) Sämtliche Kanten der Dreiecke aus T sind bis auf ihre Anfangs- und Endpunkte aus P disjunkt.

Ist ABC ein Dreieck der Triangulierung T, so sagen wir, dass auch die Kanten AB, BC und CA zur Triangulierung gehören.

Definition 1.3 (Nebenbedingung) Gegeben sei eine Teilmenge N der Kanten aus P^2. Sind alle Kanten aus N bis auf ihre Endpunkte disjunkt, so nennen wir N eine *Nebenbedingung*. Diese beinhaltet Kanten, die Teil der Triangulierung sein müssen. Die Menge N darf leer sein.

Definition 1.4 (bedingte Triangulierung) Sei T eine Triangulierung von P. Ist die Menge der Nebenbedingungen N nicht leer und ist jede Kante AB aus N Teil der Triangulierung T, so nennen wir T eine *bedingte Triangulierung* von P unter N.

Gegeben sei im Folgenden eine Punktemenge P sowie die möglicherweise leere Menge der Nebenbedingungen N. Gesucht ist eine Triangulierung T von P unter N. Um diese zu bestimmen, benötigen wir noch zwei weitere Definitionen.

Definition 1.5 (Sichtbarkeit) Zwei Punkt A und B aus P heißen *sichtbar* zueinander, wenn auf ihrer Verbindungskante AB kein weiterer Punkt aus P liegt und AB keine andere Kante schneidet.

Diese Definition soll durch ein kurzes Beispiel verdeutlicht werden.

Beispiel 1.6 Gegeben seien in der folgenden Abbildung 1.1 die Punkte A, B, C, X, Y und Z sowie die Kante YZ. Wir betrachten die Sichtbarkeit der Punktepaare (A, B), (B, C) und (C, A).

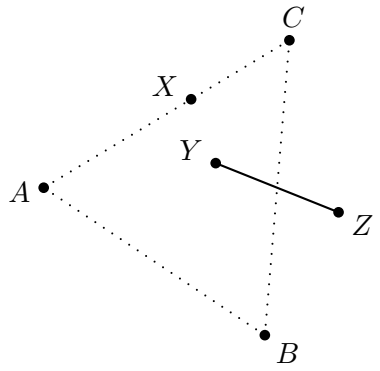


Abbildung 1.1: Beispiel Sichtbarkeit

- A und B sind sichtbar zueinander, da AB nicht unterbrochen ist
- A und C sind nicht sichtbar zueinander, da AC durch X unterbrochen wird
- B und C sind nicht sichtbar zueinander, da BC durch YZ geschnitten wird

Definition 1.7 (konvexe Hülle) Eine Menge M heißt *konvex*, wenn zu je zwei Punkten A und B aus M auch die Verbindungsstrecke AB ganz in der Menge liegt. Die *konvexe Hülle* von M ist der Schnitt aller konvexen Mengen, die M enthalten.
Sei nun M die kleinste Menge, die alle Punkte der Punktemenge P umfasst. Unter der *konvexen Hülle von P* verstehen wir den Polygonzug, der die konvexe Hülle von M umschließt.

Auch diese Definition werden wir an Hand eines kurzen Beispiels veranschaulichen.

Beispiel 1.8 Gegeben sei eine Punktemenge P, die alle Punkte aus Abbildung 1.1 umfasst. Die konvexe Hülle dieser Punktemenge ist in der folgenden Abbildung 1.2 dargestellt.

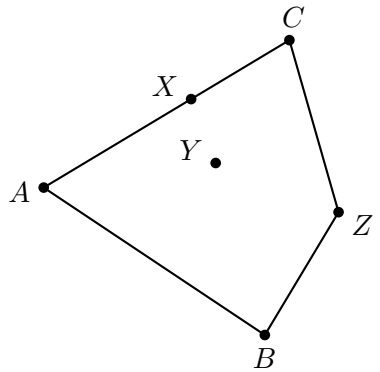


Abbildung 1.2: konvexe Hülle von P

Mit Hilfe dieser Definitionen können wir zu einer beliebigen Punktemenge P eine Triangulierung erzeugen. Wir gehen zunächst davon aus, dass die Menge N der Nebenbedingungen leer sei.

Idee 1.9 (*planesweep*-Algorithmus) Der Grundgedanke dieses Algorithmus liegt in der sogenannten *sweepline* - einer Parallelen zur y-Achse - welche sich von links nach rechts über alle Punkte hinweg bewegt. Überschreitet die *sweepline* einen Punkt, so wird dieser zur bestehenden Triangulierung hinzugefügt.

In den folgenden Abbildungen ist die senkrechte rote Kante stets die *sweepline*, die entsprechende waagerechte Kante gibt ihre Bewegungsrichtung an. Die blauen Kanten bilden die konvexe Hülle aller bisher eingefügten Punkte. Die schwarzen Kanten zeigen alle bereits zur Triangulierung gehörenden Kanten, welche nicht Teil der konvexen Hülle sind. Das linke Bild zeigt die Situation vor Einfügen des neuen Punktes, im rechten Bild ist der Punkt eingefügt.

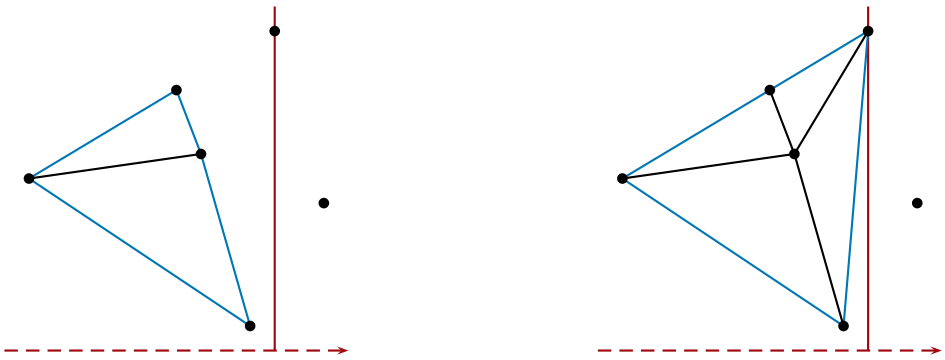


Abbildung 1.3: Momentaufnahme *planesweep* vor und nach Einfügen eines Punktes

Um einen neuen Punkt zur Triangulierung hinzuzufügen, benötigen wir in jedem Schritt die konvexe Hülle der aktuellen Triangulierung. Wir suchen alle Punkte dieser konvexen Hülle, welche zum neuen Punkt sichtbar sind. Anschließend fügen wir jeweils eine Kante zwischen dem neuen Punkt und den sichtbaren ein. Durch das Einfügen von zwei Kanten entsteht jeweils ein Dreieck. Liegen auf der aktuellen konvexen Hülle n Punkte, so werden in jedem Schritt mindestens ein, höchstens $n-1$ neue Dreiecke zur Triangulierung zugefügt.
Vor dem Einfügen des nächsten Punktes müssen wir die konvexe Hülle aktualisieren. Liegen zwischen dem ersten und dem letzten sichtbaren Punkt weitere Punkte in der konvexen Hülle, so werden diese entfernt. Anschließend wird der neue Punkt in der konvexen Hülle zwischen dem ersten und dem letzten sichtbaren Punkt eingefügt.
Dieses Vorgehen führen wir so lange fort, bis die *sweepline* alle Punkte durchlaufen hat. Die Menge der eingefügten Dreiecke bildet die Triangulierung.

Problem 1.10 (nur Punkte auf einer Geraden) Liegen die ersten Punkte aus P, die die *sweepline* überstreicht, wie in Abbildung 1.4 auf einer Geraden, so wird durch drei solcher Punkte kein Dreieck aufgespannt.

Um dies zu umgehen, fügen wir zwei zusätzliche Hilfspunkte ein. Für diese beiden Punkte wählen wir die minimale x-Koordinate und subtrahieren 1, um links von allen anderen Punkten zu liegen. Die y-Koordinaten wählen wir `2*y_max-y_min` und `2*y_min-y_max` mit `y_min` dem kleinsten, `y_max` dem größten vorkommenden y-Wert. Dadurch liegen die ersten drei Punkte auf jeden Fall nicht mehr auf einer Geraden. Die Wahl der Punkte sichert uns eine problemlose Triangulierung

zu. In den folgenden Abbildungen (und auch in den späteren Beispielen) sind diese Punkte aus optischen Gründen stets deutlich näher zusammen.
Diese zusätzlichen Punkte werden in den kommenden Abbildungen zur besseren Unterscheidung stets als rote Kreise dargestellt. Die grünen Kanten sind die neu eingefügten.

Abbildung 1.4: Punkte auf einer Geraden

Wir betrachten noch einmal die Triangulierung aus Abbildung 1.3 nach dem Einfügen des aktuellen Punktes im Vergleich mit und ohne die Hilfspunkte. Dadurch erhalten wir für die späteren Abbildungen eine bessere Vergleichsmöglichkeit.

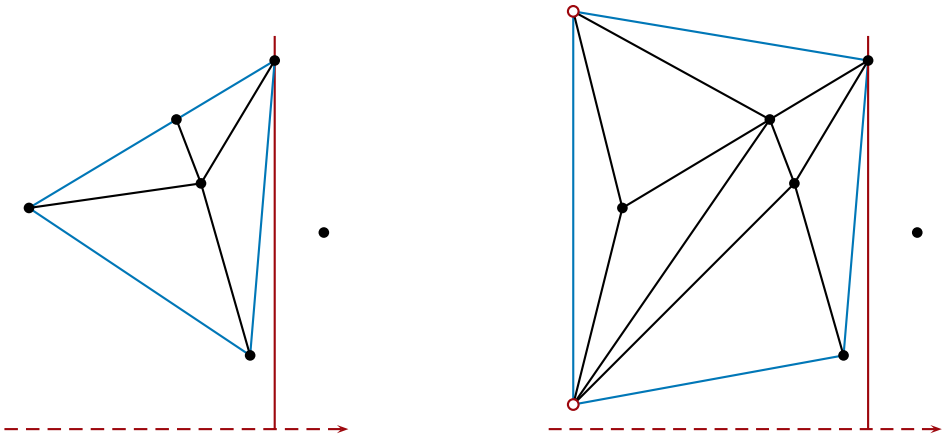


Abbildung 1.5: *planesweep*-Momentaufnahme mit und ohne Zusatzpunkte

Sei nun die Menge der Nebenbedingungen nicht mehr leer. Dann gibt es Kanten aus P^2, die auf jeden Fall Teil der Triangulierung sein müssen. Wir werden sehen, dass sich aus dieser Einschränkung im Vorgehen des *planesweep*-Algorithmus keine Änderung ergibt. Dazu betrachten wir in Abbildung 1.6 eine ähnliche Momentaufnahme wie in Abbildung 1.5, allerdings sei eine von der *sweepline* geschnittene Kante als Nebenbedingung vorgegeben.
Im gleichen Schritt wie im rechten Bild von Abbildung 1.5 wird nun ein Dreieck weniger zugefügt. Die konvexe Hülle der aktuellen Triangulierung ist nach Einfügen des neuen Punktes offensichtlich nicht mehr konvex im Sinne von Definition 1.7.
Dieses Verhalten wird auch der in Idee 1.9 beschriebene Algorithmus zeigen: Eine Nebenbedingungskante kann die Sichtbarkeit von zwei Punkten beeinflussen. Dadurch kann keine Kante entstehen, die die Nebenbedingungskante schneiden würde.

Demnach müssen wir den *planesweep*-Algorithmus nicht anpassen. Es ist lediglich notwendig, die Kanten der Nebenbedingung zu berücksichtigen, wenn über die Sichtbarkeit von zwei Punkten zueinander entschieden wird.

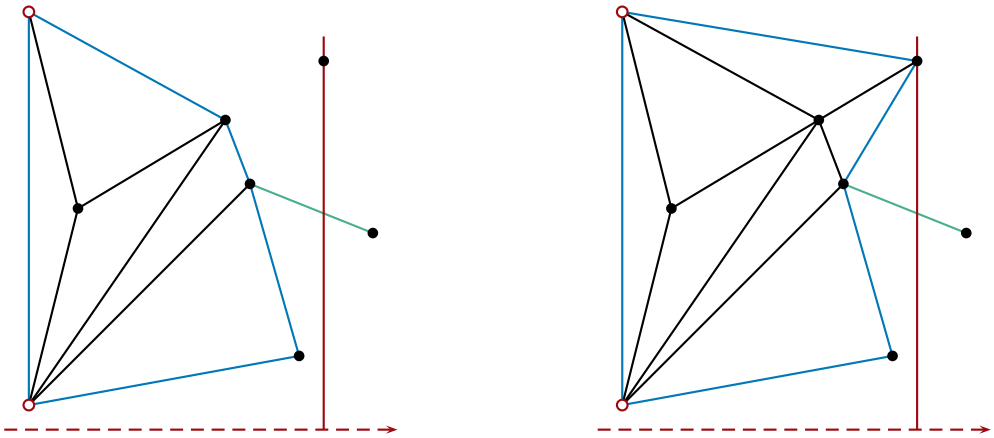


Abbildung 1.6: Momentaufnahme *planesweep* vor und nach Einfügen eines Punktes

Wir formulieren nun erstmals Optimalitätskriterien für die Triangulierung T. Hierfür benötigen wir zuerst noch folgende Definition.

Definition 1.11 (Netzweite) Die *Netzweite* einer Triangulierung T ist die Länge der längsten vorkommenden Kante.

Damit stellen wir nun unsere Anforderungen an die von uns erzeugte Triangulierung.

Anforderung 1.12 (Optimalität) Wir suchen eine Triangulierung T von P unter N, so dass

(i) die Netzweite eine vorgegebene Länge von $h_{\max}$ nicht überschreitet und

(ii) der kleinste vorkommende Winkel mindestens eine vorgegebene Größe $\alpha_{\min}$ hat.

Idee 1.13 (Kürzen der Nebenbedingungskanten) Der minimale Winkel kann beim Erzeugen der Triangulierung durch den *planesweep* nicht beachtet werden. Allerdings ist es sinnvoll, bereits vor dem Erzeugen der Triangulierung alle Kanten der Nebenbedingung N durch äquidistantes Einfügen neuer Punkte so zu unterteilen, dass die maximale Kantenlänge $h_{\max}$ nicht überschritten wird.

Da das fertige Programm ein von einem Polygonzug umschlossenes Gebiet triangulieren soll, wird durch dieses Vorgehen insbesondere auf dem ganzen Rand die maximale Netzweite von Anfang an berücksichtigt.

Beispiel 1.14 Abbildung 1.7 zeigt die Punkte aus dem bisher verwendeten Beispiel. Zusätzlich sind zwei Kanten gegeben, in der Abbildung grün dargestellt. Diese Kanten stellen die Nebenbedingung dar.Die rote Strecke zeige die Länge $h_{\max}$ an.

Wir betrachten anschließend in Abbildung 1.8 das Einfügen des gleichen Punktes wie in den Abbildungen 1.5 und 1.6 durch den *planesweep*-Algorithmus. In dieser Abbildung sehen wir auch ein Problem des verwendeten Algorithmus.

Problem 1.15 Die vom *planesweep* erzeugte Triangulierung beinhaltet häufig Dreiecke mit sehr spitzen Innenwinkeln.

Offenbar erfüllen bisher nur die geteilten Kanten der Nebenbedingung eine unserer Optimalitätsforderungen. Dieses Problem müssen wir vorerst hinnehmen. Eine Lösung wird in den Kapiteln 2 und 3 beschrieben.

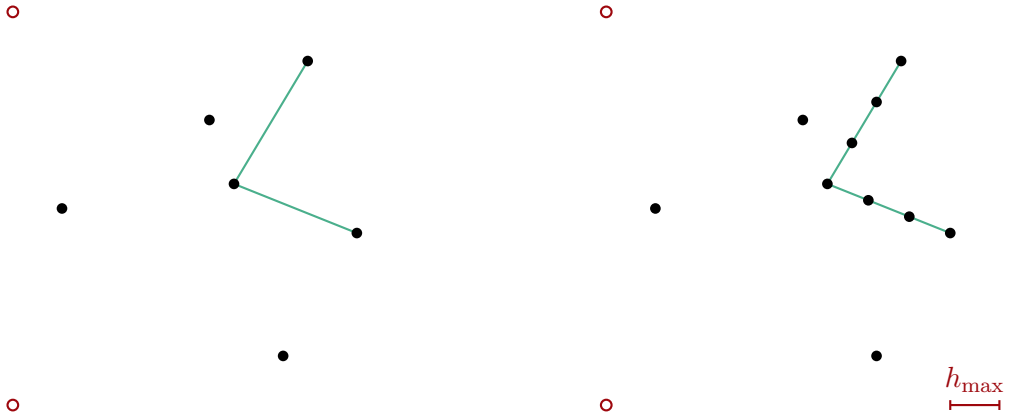


Abbildung 1.7: Aufteilen der Kanten der Nebenbedingung

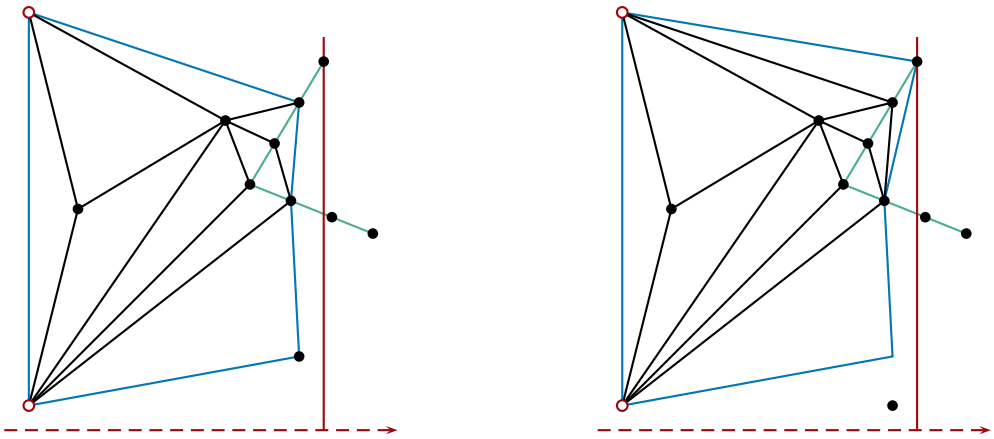


Abbildung 1.8: Momentaufnahme *planesweep* bei und nach Einfügen eines Punktes

1.2 Implementierung

Der Algorithmus erhält eine vorgegebene Punktmenge P, eine Nebenbedingung N sowie eine maximale Kantenlänge $h_{\max}$. Dabei wird durch die Kanten aus N steht ein Gebiet umschlossen. Dieses kann, wie in der folgenden Abbildung 1.9 skizziert, durch weitere Kanten definierte „Löcher“ haben. Der schraffierte Teil ist der zu triangulierende.

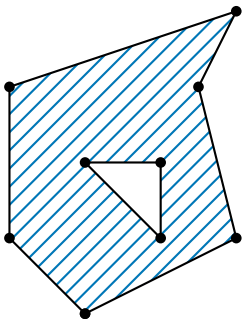


Abbildung 1.9: Beispiel eines zu triangulierenden Gebiets

Die Punkte liegen in einer beliebigen Reihenfolge vor. Kanten der Nebenbedingung werden angegeben durch den Index ihres Start- und Endpunktes.

Beispiel 1.16 Anhand des in Abbildung 1.9 gezeigten Gebiets soll das Vorgehen des *planesweep* noch einmal veranschaulicht werden. Gegeben seien folgende Punkte und Kanten. Außerdem sei als maximale Kantenlänge $h_{\max} = 3$ gegeben.

```
P = [ 1    0          N = [ 1  2
      3    1                2  3
      2.5  3                3  4
      3    4                4  5
      0    3                5  6
      0    1                6  1
      2    1                7  8
      2    2                8  9
      1    2  ]             9  7 ]
```

Tragen wir die Punkte und Kanten in ein Koordinatensystem ein, so ergibt sich die in der folgenden Abbildung 1.10 dargestellte Anfangsbedingung.

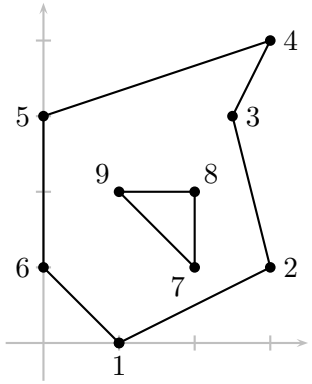


Abbildung 1.10: Beispiel für P und N

Der *planesweep*-Algorithmus hat den folgenden Ablauf.

Pseudocode 1.17 (*planesweep*)

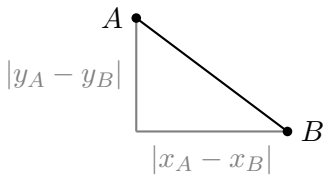
füge die beiden führenden Hilfspunkte ein
teile alle zu langen Kanten des Anfangspolygons in Teilkanten auf
sortiere die Punkte aufsteigend nach ihrer x-Koordinate
for all Punkte P **do**
    verbinde P mit allen sichtbaren Punkten der konvexen Hülle
    füge P in die konvexe Hülle ein
end for
entferne die beiden Hilfspunkte und alle zugehörigen Dreiecke

Bevor wir die Realisierung des *planesweep* betrachten, benötigen wir offensichtlich noch Funktionen zum Verkürzen der Kanten und zum Sortieren der Punkte.

1.2.1 Hilfsfunktionen

Diese für den *planesweep* benötigten Hilfsfunktionen werden wir im Folgenden betrachten.

Hilfsfunktion 1.18 (`length_edge`)
Wir benötigen eine Funktion, um die Länge einer Kante AB zu berechnen. Dies werden wir mit Hilfe des Satzes von Pythagoras realsieren.



Damit ergibt sich folgende Funktion.

Programm 1.1: length_edge.m

```
1 function [edge_length] = length_edge(a, b)
2
3   edge_length = sqrt((a(1)-b(1))^2 + (a(2)-b(2))^2);
4
5 end
```

Hilfsfunktion 1.19 (`split_edges`)
Für die fertige Triangulierung ist eine maximale Netzweite $h_{\max}$ gewünscht. Um später einen hohen Aufwand zu vermeiden, werden die Kanten des begrenzenden Polygonzugs schon vor der Erstellung der Triangulierung durch Einfügen neuer Punkte auf entsprechend kurze Strecken unterteilt.

Die Kanten liegen zunächst wie in Beispiel 1.16 bei k Kanten in einer $k \times 2$ Matrix vor. Es ist sehr aufwändig, zu überprüfen, ob eine bestimmte Kante von einem Punkt `i` zu einem Punkt `j` existiert. Im schlechtesten Fall müssen dabei sämtliche Kanten überprüft werden.
Wir werden daher in dieser Funktion auch die Speicherstruktur der Kanten ändern, um einen schnelleren Zugriff zu haben. Dafür benötigen wir folgende Definition.

Definition 1.20 (sparse) Gegeben sei eine $n \times n$ Matrix A. Wir bezeichnen mit $\mathrm{nz}(A)$ („*non-zero*“) die Anzahl der Einträge von A, die ungleich null sind.
Wir nennen A *sparse*, falls es eine Konstante $c \ll n$ gibt, so dass gilt

$$\mathrm{nz}(A) \leq c \cdot n.$$

Für *sparse* Matrizen gibt es effiziente Speicherformen, die nur die Position der Einträge sowie deren Inhalt speichert.

Wir werden die Kanten in der im Folgenden beschriebenen Form speichern.
Bei n Punkten legen wir eine $n \times n$ *sparse* Matrix an. Diese Matrix hat an Position `(i,j)`, `i>j` genau dann einen Eintrag ungleich null, wenn die Kante `(i,j)` oder `(j,i)` existiert. Um zu überprüfen, ob eine Kante zwischen zwei Punkten `i` und `j` mit `i>j` existiert, muss diese damit

nicht mehr direkt gesucht werden. Es genügt statt dessen, den Eintrag `(i,j)` in der Matrix der Kanten abzufragen.
Wir sagen eine *sparse* Matrix `M` hat an Position `(i,j)` einen Eintrag, falls `M(i,j)` nicht null ist. Andernfalls sagen wir dass `M` dort keinen Eintrag hat.

Wir betrachten nun die Fortsetzung von Beispiel 1.16.

Beispiel 1.21 Eine der Kanten aus Abbildung 1.10 - die Kante `(4,5)` - überschreitet die maximale Kantenlänge $h_{\max} = 3$. Diese Kante wird durch Einfügen eines neuen Punktes halbiert.

Durch Halbieren der Kante `(5,4)` ergibt sich Abbildung 1.11.

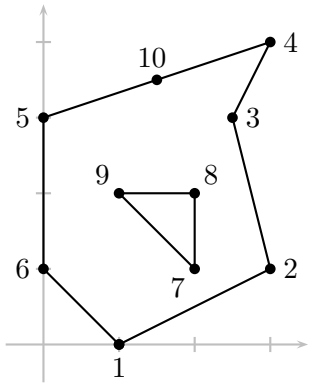


Abbildung 1.11: Beispiel für P und N

Nach Einfügen des neuen Punktes und Anpassen der Kanten auf die neue Speicherstruktur in einer *sparse* Matrix erhalten wir folgende Koordinaten und Kanten.

```
coordinates = [ 1    0          edges = [ 0 0 0 0 0 0 0 0 0 0 0
                3    1                    1 0 0 0 0 0 0 0 0 0 0
                2.5  3                    0 1 0 0 0 0 0 0 0 0 0
                3    4                    0 0 1 0 0 0 0 0 0 0 0
                0    3                    0 0 0 0 0 0 0 0 0 0 0
                0    1                    0 0 0 0 1 0 0 0 0 0
                2    1                    0 0 0 0 0 0 0 0 0 0 0
                2    2                    0 0 0 0 0 0 1 0 0 0
                1    2                    0 0 0 0 0 0 1 1 0 0
                1.5  2.5 ]                0 0 0 1 1 0 0 0 0 0 ]
```

Die Positionen der Einträge der Kantenmatrix erhält man über den Matlab-internen `find`-Befehl. Dabei werden die Einträge spaltenweise und innerhalb einer Spalte von oben nach unten angegeben. Der Aufruf von `find` liefert das folgende Ergebnis.

```
[zeilenindex, spaltenindex, eintrag] = find(edges)
```

Wird der Eintrag in der Matrix nicht benötigt (in unserem Fall ist er beispielsweise immer eins), so kann das letzte Argument weggelassen werden. Der `find`-Befehl aufgerufen mit unserer Kantenmatrix `edges` liefert folgendes Ergebnis.

```
[start, end] = find(edges);
[start, end] = [ 2  1
                 6  1
                 3  2
                 4  3
                10  4
                 6  5
                10  5
                 8  7
                 9  7
                 9  8 ]
```

Im Folgenden betrachten wir das entsprechende Programm.

Programm 1.2: split_edges.m

```
1  function [coordinates, edges] = split_edges(coordinates, edges, h_max)
2
3    [edges_start, edges_end] = find(edges);
4
5    cnt_edges = size(edges_start, 1);
6    cnt_points = size(coordinates, 1);
7
8    edges = sparse(cnt_points, cnt_points);
9
10   while ~isempty(edges_start)
11     % aktuelle Kante steht in edges_start/end an Pos 1
12
13     length_act_edge = length_edge(coordinates(edges_start(1), :),...
14       coordinates(edges_end(1), :));
15
16     if  length_act_edge <= h_max % Kante kurz genug
17       edges(edges_start(1), edges_end(1)) = 1;
18
19     else % Kante zu lang
20       cnt_points = ceil(length_act_edge / h_max);
21
22       a = coordinates(edges_start(1),:);
23       b = coordinates(edges_end(1),:);
24
25       coordinates = [coordinates; a + 1/cnt_points * (b-a)];
26       size_coordinates = size(coordinates,1);
27
28       edges(size_coordinates, edges_start(1)) = 1;
29
30       for i = 2 : cnt_points-1
31         coordinates = [coordinates; a + i/cnt_points * (b-a)];
32         edges(size_coordinates, size_coordinates) = 1;
33         size_coordinates = size_coordinates+1;
34       end
35
36       edges(size_coordinates, edges_end(1)) = 1;
37     end
38
39     edges_start(1) = [];
```

```
40      edges_end(1) = [];
41    end
42
43  end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
3	Erzeuge eine Liste aller Kanten.
5-6	Sei n die Anzahl der Punkte. Erzeuge eine leere $n \times n$ *sparse* Matrix für die Kanten.
ab 10	Betrachte alle Kanten. Die aktuelle Kante steht dabei immer in `edges_start(1)` und `edges_end(1)`.
13-14	Bestimme die Länge der aktuellen Kante.
16-17	Ist die Kante kurz genug, muss sie nicht geteilt werden. Trage die Kante in die *sparse* Matrix ein. In diesem Fall geht es in Zeile 39 weiter.
ab 19	Die Kante ist zu lang und muss geteilt werden.
20	Bestimme, in wie viele Teilkanten die aktuelle Kante unterteilt werden muss.
22-23	Speichere in `a` und `b` den Anfangs- und Endpunkt der aktuellen Kante.
25-28	Füge den ersten Punkt ein sowie die Kante von `a` zum neuen Punkt.
30-34	Füge alle weiteren Zwischenpunkte ein und verbinde sie jeweils durch Kanten.
36	Füge die Kante zwischen dem letzten Zwischenpunkt und `b` ein.
39-41	Entferne die aktuelle Kante, um die nächste Kante zu erhalten.

Hilfsfunktion 1.22 (`sort_coordinates`)
Im Planesweep-Algorithmus werden alle Punkte durch die sogenannte *sweepline* durchlaufen. Diese bewegt sich von links nach rechts über alle Punkte hinweg. Sobald ein Punkt von der *sweepline* geschnitten wird, wird dieser in die bestehende Triangulierung eingefügt. Dafür werden alle Punkte aufsteigend nach ihrer x-Koordinate sortiert. Punkte mit gleicher x-Koordinate werden werden zusätzlich aufsteigend nach ihrer y-Koordinate sortiert.
Nach dem Sortieren kann die Liste der Punkte von Anfang bis Ende durchlaufen werden, um das Verhalten der *sweepline* zu imitieren.

Bevor die Funktion zum Sortieren der Punkte aufgerufen wird, müssen die beiden führenden Hilfspunkte hinzugefügt werden. Diese werden in der Liste der sortierten Punkte die Indizes `1` und `2` erhalten. Mit diesem Wissen können wir die Hilfspunkte am Ende leicht wieder entfernen.

Beispiel 1.23 Die Punkte aus Beispiel 1.21 werden wie oben beschrieben umsortiert. Die beiden roten Kreise zeigen die Positionen der Hilfspunkte an. Diese sind von der Funktionalität in diesem Beispiel nicht signifikant und werden nur zur Vollständigkeit mitbetrachtet. Daher sind sie in den kommenden Abbildungen mit deutlich näher beieinanderliegenden y-Koordinaten dargestellt. Beim Sortieren müssen auch die Kanten angepasst werden, da wir diese über die Indizes ihrer Endpunkte definiert hatten.

```
coordinates = [ -1   -0.5    edges = [ 0  0  0  0  0  0  0  0  0  0
                -1    4.5              0  0  0  0  0  0  0  0  0  0
                 0    1                0  0  0  0  0  0  0  0  0  0
                 0    3                0  0  1  0  0  0  0  0  0  0
                 1    0                0  0  1  0  0  0  0  0  0  0
                 1    2                0  0  0  0  0  0  0  0  0  0
                 1.5  3.5              0  0  0  1  0  0  0  0  0  0
                 2    1                0  0  0  0  0  1  0  0  0  0
                 2    2                0  0  0  0  0  1  0  1  0  0
                 2.5  3                0  0  0  0  0  0  0  0  0  0
                 3    1                0  0  0  0  1  0  0  0  0  1
                 3    4 ]              0  0  0  0  0  0  1  0  0  1 ]
```

In der folgenden Abbildung 1.12 betrachten wir die umsortierten Punkte mit den zugefügten Hilfspunkten noch einmal im Koordinatensystem.

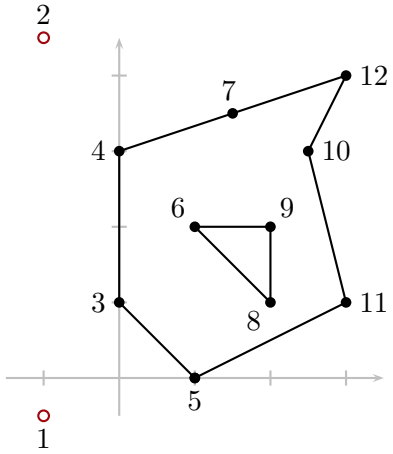


Abbildung 1.12: Punkte sortiert, Kanten entsprechend umbenannt

Wir betrachten nun den Algorithmus, der dies realisiert.

Programm 1.3: sort_coordinates.m

```
1  function [coordinates, edges]...
2                   = sort_coordinates(coordinates, edges)
3
4    % ------------------------ Punkte sortieren --------------------------
5    [coordinates, permutation] = sortrows(coordinates, [1,2]);
6
7    % ------------------------- Kanten anpassen --------------------------
8    [rows, columns] = find(edges);
9
10   cnt_edges = size(rows,1);
11   cnt_coord = size(coordinates, 1);
12
13   tmp = zeros(cnt_edges, 2);
14
15   for i = 1:cnt_edges
16     tmp(i, 1:2) = [(find(permutation(:) == rows(i))), ...
```

```
17        (find(permutation(:) == columns(i)))];
18    end
19
20    edges = sparse(cnt_edges,cnt_edges);
21
22    for i = 1:cnt_edges
23      edges(max(tmp(i,:)), min(tmp(i,:))) = 1;
24    end
25
26  end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
5	Sortiere alle Punkte aufsteigend nach der x-Koordinate, bei gleichen x-Koordinaten aufsteigend nach der y-Koordinate. Die dabei verwendete Permutation wird gespeichert.
8	`edges` ist eine $n \times n$ sparse Matrix mit einem Eintrag an Position `(i,j)`, falls es eine Kante von `i` nach `j` gibt. Dabei stehen alle Einträge unterhalb der Diagonalen. Speichere jeweils den Zeilen- und Spaltenindex der Einträge.
10-11	Bestimme die Anzahl der Kanten und der Koordinaten.
13-18	Speichere die entsprechende Permutation der Start- und Endpunkte alle Kanten aus `rows` und `columns`.
20-24	Überschreibe die Matrix `edges` so, dass die alte Struktur wiederhergestellt ist und die permutierten Punkte korrekt miteinander verbunden werden.

Hilfsfunktion 1.24 (`is_viewable`) Im Verlauf des *planesweep* wird jeder neue Punkt in eine bestehende konvexe Hülle eingefügt. Dieser wird zwischen den ersten und den letzten nach Definition 1.5 sichtbaren Punkt gesetzt.
Dabei überprüfen wir

(i) ob beide Punkte gleich sind (dann sind sie sichtbar zueinander),

(ii) ob ein anderer Punkt auf der Verbindungsgeraden liegt oder

(iii) ob eine andere Gerade die Verbingunsgerade schneidet.

Gegeben seien zwei Punkte $A = (x_A,\ y_A)^T$ und $B = (x_B,\ y_B)^T$. Dabei gilt: $f = A + \lambda(B - A)$ ist die Verbindungsgerade von A und B. Liegt ein Punkt $C = (x_C,\ y_C)^T$ auf dieser Geraden, so hat das Gleichungssystem

$$(1 - \lambda)x_A + \lambda x_B = x_C$$
$$(1 - \lambda)y_A + \lambda y_B = y_C$$

eine Lösung. Da wir nur an Punkten auf der Geraden interessiert sind, die die Sichtbarkeit beeinflussen könnten, betrachten wir dabei nur die Punkte C, deren x-Koordinaten zwischen denen von A und B liegen. Diese sind durch die Sortierung der Koordinaten genau die, die einen Index zwischen dem von A und B haben.

Der Punkt C unterbricht offensichtlich die Strecke AB, falls

$$\left(1 - \frac{x_C - x_A}{x_B - x_A}\right) y_A + \frac{x_C - x_A}{x_B - x_A} \cdot y_B = y_C \quad \text{gilt.} \tag{1.1}$$

Wir müssen nun noch überprüfen können, ob die Strecke AB durch eine andere Strecke UV geschnitten wird. Ein solcher Schnittpunkt erfüllt die Gleichung

$$A + \lambda(B - A) = U + \mu(V - U)$$

Wir können dies umformulieren in das Gleichungssystem

$$\begin{pmatrix} x_B - x_A & x_U - x_V \\ y_B - y_A & y_U - y_V \end{pmatrix} \begin{pmatrix} \lambda \\ \mu \end{pmatrix} = \begin{pmatrix} x_U - x_A \\ y_U - y_A \end{pmatrix}.$$

Zwei Geraden haben genau dann einen Schnittpunkt, wenn ihre Richtungsvektoren linear unabhängig sind. Die Richtungsvektoren der beiden Strecken AB und UV sind die Spalten der linken Seite des Gleichungssystems. Diese sind offensichtlich linear unabhängig, falls

$$\det \begin{pmatrix} x_B - x_A & x_U - x_V \\ y_B - y_A & y_U - y_V \end{pmatrix} \neq 0 \quad \text{gilt.}$$

λ und μ bestimmen wir mit Hilfe der Cramer'schen Regel. Es gilt

$$\begin{aligned} \lambda &= \det \begin{pmatrix} x_U - x_A & x_U - x_V \\ y_U - y_A & y_U - y_V \end{pmatrix} \Big/ \det \begin{pmatrix} x_B - x_A & x_U - x_V \\ y_B - y_A & y_U - y_V \end{pmatrix} \quad \text{und} \\ \mu &= \det \begin{pmatrix} x_B - x_A & x_U - x_A \\ y_B - y_A & y_U - y_A \end{pmatrix} \Big/ \det \begin{pmatrix} x_B - x_A & x_U - x_V \\ y_B - y_A & y_U - y_V \end{pmatrix}. \end{aligned} \tag{1.2}$$

Offensichtlich liegt der Schnittpunkt der beiden Geraden AB und UV genau dann zwischen A und B, wenn sowohl λ als auch μ echt zwischen null und eins liegen.

Wir betrachten nun wieder den verwendeten Quellcode.

Programm 1.4: is_viewable.m

```
1  function [viewable] = is_viewable(coordinates, active_edges, index_a,...
2                                    index_b, b_not_in_coord)
3  % bestimmt, ob die Punkte a und b zueinander sichtbar sind
4
5    tol = 1e-14;
6
7    if nargin == 4
8      a = coordinates(index_a, :);
9      b = coordinates(index_b, :);
10   else
11     a = coordinates(index_a, :);
12     b = index_b;
13   end
14
15   % überprüfe, ob active_edges in nx2 oder sparse kxk vorliegt -
```

```
16  % hinterher auf jeden Fall nx2
17  if active_edges(1,2) == 0
18    % ist active_edges sparse, ist hier kein Eintrag
19    if nargin == 4
20      if active_edges(max(index_a, index_b), min(index_a, index_b))
21        % Kante von a nach b vorhanden
22        viewable = 1;
23        return
24      end
25    end
26
27    [tmp(:,1), tmp(:,2)] = find(active_edges);
28    active_edges = tmp;
29  end
30
31  % überprüfe für alle Punkte zwischen min(index_a, index_b) und
32  % max(index_a, index_b), ob die Punkte auf der Geraden liegen.
33  if nargin == 4
34    if a(1) == b(1)
35      if index_a-index_b > 1
36        viewable = 0;
37        return
38      end
39    end
40    index_min = min(index_a, index_b);
41    index_max = max(index_a, index_b);
42    lambda = (coordinates(index_min+1:index_max-1,1) - a(1)) ...
43             ./ (b(1)-a(1));
44    if find(abs((1-lambda) .* a(2) + lambda .* b(2) ...
45            - coordinates(index_min+1:index_max-1,2)) < tol)
46      viewable = 0;
47      return
48    end
49  else
50    if a(1) == b(1)
51      if find(coordinates([1:index_a-1, index_a+1:end],1) == a(1),1)
52        viewable = 1;
53        return
54      end
55    end
56    lambda = (coordinates(1:size(coordinates,1),1) - a(1)) ...
57             ./ (b(1) - a(1));
58    if find(abs((1-lambda) .* a(2) + lambda .* b(2) ...
59            - coordinates(1:size(coordinates,1),2)) < tol, 1)
60      viewable = 0;
61      return
62    end
63  end
64
65  % bestimme Schnittpunkt der Gerade ab mit allen aktiven Kanten uv:
66  % a + lambda * (b-a) = u + mu * (v-u)
67  u = coordinates(active_edges(:, 1) , :);
68  v = coordinates(active_edges(:, 2) , :);
69
```

```
70    % bestimme die Geraden, bei denen der Nenner der Cramerschen Regel
71    % ungleich 0 wäre - schließt aus, dass a,b,u und v auf einer
72    % Geraden liegen bzw ab und uv parallel sind
73    nenner = find((b(1) - a(1)) .* (u(:, 2) - v(:, 2))...
74                  + (a(2) - b(2)) .* (u(:, 1) - v(:, 1)));
75
76    lambda = ((u(nenner, 1) - a(1)) .* (u(nenner, 2) - v(nenner, 2))...
77          + (a(2) - u(nenner, 2)) .* (u(nenner, 1) - v(nenner, 1)))...
78          ./ ((b(1) - a(1)) .* (u(nenner, 2) - v(nenner, 2))...
79          + (a(2) - b(2)) .* (u(nenner, 1) - v(nenner, 1)));
80
81    mu = ((b(1) - a(1)) .* (u(nenner, 2) - a(2))...
82          + (a(2) - b(2)) .* (u(nenner, 1) - a(1)))...
83          ./ ((b(1) - a(1)) .* (u(nenner, 2) - v(nenner, 2))...
84          + (a(2) - b(2)) .* (u(nenner, 1) - v(nenner, 1)));
85    % falls lambda UND mu zwischen 0 und 1, also Schnittpunkt liegt
86    % auf der Strecke ~> a, b nicht sichtbar
87
88    lambda = [lambda'; 1:length(lambda)]';
89    mu = [mu'; 1:length(mu)]';
90    lambda_upper = lambda(lambda(:,1) < 1-tol, 2)';
91    mu_upper = mu(mu(:,1) < 1-tol, 2)';
92    lambda_lower = lambda(lambda(:,1) > tol, 2)';
93    mu_lower = mu(mu(:,1) > tol, 2)';
94
95    indice_lambda = [];
96    indice_mu = [];
97    for i = 1:length(lambda_upper)
98      indice_lambda = [indice_lambda; ...
99                       find(lambda_upper(i) == lambda_lower(:))];
100   end
101
102   for i = 1:length(mu_upper)
103     indice_mu = [indice_mu; find(mu_upper(i) == mu_lower(:))];
104   end
105
106   intersec_lambda = lambda_lower(indice_lambda);
107   intersec_mu = mu_lower(indice_mu);
108
109   intersec = [];
110   for i=1:length(intersec_mu)
111     intersec = [intersec; ...
112                 find(intersec_lambda(:) == intersec_mu(i))];
113   end
114
115   if isempty(intersec)
116     viewable = 1;
117   else
118     viewable = 0;
119   end
120 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
5	vorgegebene Rundungsfehlertoleranz
7-13	Werden vier Argumente übergeben, überprüft die Funktion die Sichtbarkeit von zwei Punkten A und B, die in `coordinates` gespeichert sind. Ist `b_not_in_coord` gesetzt, so ist in `index_b` kein Index sondern ein Punkt gespeichert. Dieser kommt im Allgemeinen nicht in `coordinates` vor.
ab 17	Überprüfe, ob die Kanten als $n \times 2$-Matrix oder als quadratische *sparse* Matrix vorliegen.
19-25	Sind beide Punkte aus `coordinates`, so sind sie sichtbar, falls sie durch eine Kante verbunden werden.
27-28	Ist eine *sparse* Matrix für die Kanten gegeben, wandle sie um in $n \times 2$-Matrix.
ab 33	Überprüfe, ob die Verbindungsgerade AB durch andere Punkte unterbrochen wird. Sind beide Punkte in `coordinates`, so genügt es, die zwischen den beiden Indizes liegenden Punkte zu untersuchen. Alle anderen Punkte liegen von ihrer x-Koordinate her nicht zwischen A und B und können damit die Sichtbarkeit nicht beeinflussen.
34-39	Vermeide in Gleichung 1.1 die Division durch null, falls beide Punkte die gleiche x-Koordinate haben. Liegt in der sortierten Liste noch ein Punkt dazwischen, sind sie nicht sichtbar zueinander.
40-48	Bestimme analog zur theoretischen Überlegung, ob ein Punkt die Gerade unterbricht.
49-62	Hier erfolgen die analogen Überlegungen zu den Zeilen 34-48 für einen Punkt B, der nicht in `coordinates` einsortiert ist. Dementsprechend müssen alle Punkte betrachtet werden statt nur denen zwischen A und B.
67-68	Speichere alle Anfangs- und Endpunkte der Kanten.
73-74	Bestimme alle Geraden, bei denen der Nenner in Gleichung 1.2 ungleich null ist. Alle anderen Geraden sind parallel und haben demnach auch keinen Schnittpunkt.
76-84	Bestimme λ und μ nach Gleichung 1.2.
88-107	Finde die Indizes aller Geraden, für die λ oder μ zwischen null und eins liegt. Um Rundungsfehler zu vermeiden, verwende `tol` als Fehlertoleranz.
110-113	Finde aus den in den Zeilen 88-107 gefundenen Geraden alle, für die gleichzeitig λ *und* μ zwischen null und eins liegen.
115-119	Wurde in den Zeilen 110-113 mindestens eine Gerade gefunden, so sind A und B nicht sichtbar. Andernfalls wird die Verbindungsgerade AB weder durch andere Punkte noch durch andere Kanten geschnitten und die Punkte sind sichtbar.

1.2.2 Realisierung *planesweep*

Wir haben nun sämtliche Hilfsfunktionen realisiert, die wir für den in Pseudocode 1.17 beschriebenen *planesweep* benötigen. Bevor wir die Realisierung des *planesweep* beginnen, betrachten wir erneut unser Beispiel.

Beispiel 1.25 Die Punkte werden von links nach rechts in die Triangulierung eingefügt. Aus optischen Gründen wurden im Beispiel die Hilfspunkte mit deutlich näher zusammen liegenden y-Koordinaten verwendet und dargestellt.

```
coordinates = [ -1   -0.5
                -1    4.5
                 0    1
                 0    3
                 1    0
                 1    2
                 1.5  3.5
                 2    1
                 2    2
                 2.5  3
                 3    1
                 3    4
```

```
triangulation = [ 2  1  3
                  3  4  2
                  1  5  3
                  3  5  4
                  5  6  4
                  6  7  4
                  4  7  2
                  5  8  6
                  8  9  6
                  6  9  7
                  8 10  9
                  9 10  7
                  5 11  8
                  8 11 10
                 11 12 10
                 10 12  7
                  7 12  2 ]
```

Im Koordinatensystem ergibt sich damit das in Abbildung 1.13 gezeigte Bild. Die blauen Kanten zeigen die konvexe Hülle. Die grünen Kanten sind die nicht von der konvexen Hülle überdeckten Kanten der Nebenbedingung.

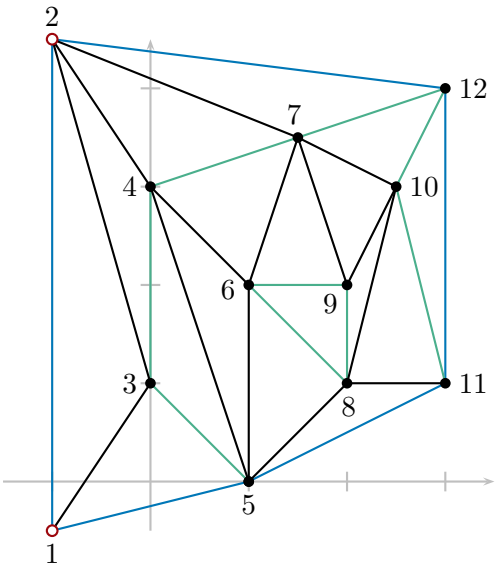


Abbildung 1.13: Triangulierung der Punkte

Am Ende des *planesweep*-Algorithmus werden die beiden führenden Hilfspunkte wieder aus `coordinates` entfernt. Gleichzeitig werden alle Dreiecke, die einen der beiden Punkte beinhalten, aus `triangulation` gelöscht.
Da die Einträge von `triangulation` sich nun offenbar nicht mehr auf die richtigen Indizes der entsprechenden Punkte beziehen und die beiden Hilfspunkte die ersten beiden Einträge aus `coordinates` waren, werden sämtliche Einträge aus `triangulation` um 2 verringert.

Beispiel 1.26 Wir betrachten nach dem Entfernen der Hilfspunkte noch einmal das Beispiel 1.25. Dabei erhalten wir die folgenden Einträge für `coordinates` und `triangulation`.

```
coordinates = [  0   1          triangulation = [ 1  3  2
                 0   3                            3  4  2
                 1   0                            4  5  2
                 1   2                            3  6  4
                 1.5 3.5                          6  7  4
                 2   1                            4  7  5
                 2   2                            6  8  7
                 2.5 3                            7  8  5
                 3   1                            3  9  6
                 3   4                            6  9  8
                                                  9 10  8
                                                  8 10  5 ]
```

In der folgenden Abbildung 1.14 sind im linken Bild alle Dreiecke grau hinterlegt, die einen der beiden Hilfspunkte beinhalten.

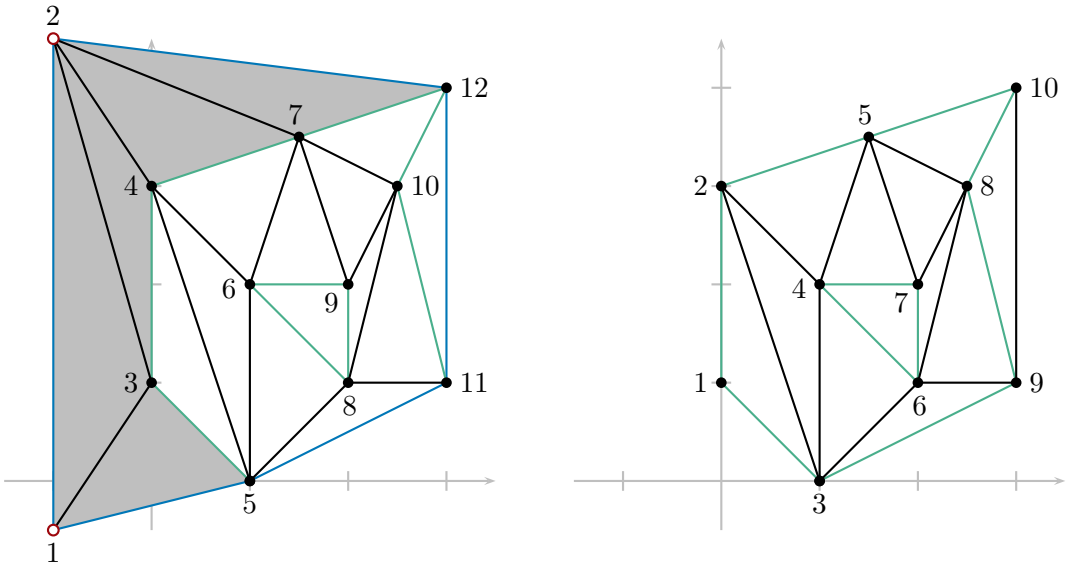


Abbildung 1.14: Triangulierung der Punkte mit und ohne Hilfspunkte

Das rechte Bild zeigt die Punkte und die Triangulierung, die der *planesweep* erzeugt. Dabei bilden weiter die blauen Kanten die konvexe Hülle. Die grünen Kanten sind die nicht von der konvexen Hülle überdeckten Kanten der Nebenbedingung.

Wir betrachten anschließend die Implementierung des *planesweep*-Algorithmus.

Programm 1.5: planesweep.m

```
1 function [coordinates, triangulation, boundaries]...
2   = planesweep(coordinates, edges, h_max)
3
4   % ------------------- Vorbereitung der Variablen ---------------------
5
6   % erzeuge sparse Matrix für Kanten der Form edge(i,j)=1, falls Kante
7   % von i nach j vorhanden
8   edges = sparse(max(edges,[],2), min(edges,[],2), ...
9                  ones(size(edges,1),1));
```

```
10
11    % Kanten splitten
12    [coordinates, edges] = split_edges(coordinates, edges, h_max);
13
14    % füge Kante vor dem Polygonzug ein, um Anfangsprobleme zu vermeiden
15    y_min = min(coordinates(:,2));
16    y_max = max(coordinates(:,2));
17
18    dummy_points = [
19      min(coordinates(:,1)) - 1, 2 * y_max - y_min;
20      min(coordinates(:,1)) - 1, 2 * y_min - y_max];
21    coordinates = [coordinates; dummy_points];
22
23    % Punkte nach x-Koordinate sortieren
24    [coordinates, edges] = sort_coordinates(coordinates, edges);
25
26    % Grundkanten merken
27    boundaries = edges(3:end, 3:end);
28
29    % Variablen vorinitialisieren
30    length_coordinates = size(coordinates,1);
31    triangulation = zeros(2*length_coordinates, 3);
32    triangulation(1,:) = [1 3 2];
33    index_next_triangle = 2;
34    conv_chain = [2 1 3];
35    edges(2,1) = 1;
36    edges(3,1) = 1;
37    edges(3,2) = 1;
38
39    % --------------------------- planesweep -----------------------------
40    for index_next_point = 4:length_coordinates
41
42      length_conv_chain = size(conv_chain,2);
43
44      % arbeite am Anfang der konvexen Kette alle ab, die sichtbar sind,
45      % am Ende der Schleife hat man den ersten nicht sichtbaren Punkt
46      % last_viewable und first_viewable sind Indize in conv_chain
47      last_viewable = 1;
48      first_viewable = 0;
49      all_viewable = 0;
50      last_is_set = 0;
51      while is_viewable(coordinates, edges, conv_chain(last_viewable), ...
52                        index_next_point)
53        last_is_set = 1;
54        if last_viewable == length_conv_chain % alle Punkte sichtbar
55          first_viewable = 1;
56          break
57        end
58
59        last_viewable = last_viewable + 1;
60      end
61      % der Punkt last_viewable ist nicht mehr sichtbar
62      if last_viewable > 1
63        last_viewable = last_viewable - 1;
```

```
64      end
65
66      % überspringe alle Punkte, die nicht sichtbar sind
67      % sobald die Schleife abbricht, ist der aktuelle Punkt sichtbar
68      if first_viewable == 0
69        first_viewable = last_viewable + 1;
70        while ~is_viewable(coordinates, edges, ...
71                           conv_chain(first_viewable), index_next_point)
72          if first_viewable == length_conv_chain
73            % ist der letzte Punkt von conv_chain nicht sichtbar,
74            % ist der erste der erste sichtbare, da er in der
75            % letzten Schleife übersprungen wurde
76            first_viewable = 1;
77            break
78          end
79          first_viewable = first_viewable+1;
80
81        end
82      end
83
84      % falls am Anfang kein Punkt sichtbar war, finde den nächsten
85      if ~last_is_set
86        last_viewable = first_viewable; % ist sichtbar
87        while is_viewable(coordinates, edges, ...
88                          conv_chain(last_viewable), index_next_point)
89          if last_viewable == length_conv_chain % alle Punkte sichtbar
90            all_viewable = 1;
91            break
92          end
93          last_viewable = last_viewable + 1;
94        end
95        if ~all_viewable
96          last_viewable = last_viewable - 1;
97        end
98      end
99
100     %liegt der erste sichtbare hinter dem letzten sichtbaren, so tausche
101     %die beiden Array-Hälften
102     if first_viewable > last_viewable
103       conv_chain = [conv_chain(last_viewable+1:end), ...
104                     conv_chain(1:last_viewable)];
105       first_viewable = first_viewable - last_viewable;
106       last_viewable = length_conv_chain;
107     end
108
109     % Triangulierung erweitern
110     for i = first_viewable:last_viewable-1
111       edges(index_next_point, conv_chain(i)) = 1;
112       triangulation(index_next_triangle,:) = [...
113           conv_chain(i), index_next_point, conv_chain(i+1)];
114       index_next_triangle = index_next_triangle + 1;
115     end
116     edges(index_next_point, conv_chain(last_viewable)) = 1;
117     conv_chain(first_viewable+1:last_viewable-1) = [];
```

```
118     conv_chain = [conv_chain(1:first_viewable), index_next_point, ...
119                   conv_chain(first_viewable+1:end)];
120   end
121
122   % ------------------------- Nachbereitung --------------------------
123
124   % entferne alle nicht verwendeten Speicherplätze in triangulation
125   triangulation(index_next_triangle:end, :) = [];
126
127   % finde alle Dreiecke, die den Punkt 1 oder 2 beinhalten
128   tmp =  triangulation(:,1) == 1 | ...
129          triangulation(:,2) == 1 | ...
130          triangulation(:,3) == 1 | ...
131          triangulation(:,1) == 2 | ...
132          triangulation(:,2) == 2 | ...
133          triangulation(:,3) == 2 ;
134
135   % entferne diese Dreiecke
136   triangulation(tmp,:) = [];
137
138   % passe Triangulierung an, dh subtrahiere von allen Punkten 2, um
139   % Wegfallen von führenden Punkten auszugleichen
140   triangulation = triangulation - 2;
141
142   % entferne Hilfspunkte
143   coordinates(1:2,:) = [];
144 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
8-9	Erstellen einer *sparse* Matrix für die Kanten wie in Hilfsfunktion 1.19 beschrieben.
12	Aufteilen der zu langen Kanten.
15-21	Einfügen der Hilfspunkte links des zu triangulierenden Gebiets.
24	Sortieren der Punkte nach x-Koordinate.
27	Speichere die Nebenbedingungskanten (das sind alle bisher gegebenen). Die ersten beiden Punkte in `coordinates` werden am Ende dieser Funktion entfernt - speichere daher gleich so, als seien sie nicht da.
31-37	Initialisiere die benötigten Variablen. Dabei sei zu Beginn stets ein Dreieck aus den ersten drei Punkten gegeben.
ab 40	Durchlaufe alle Punkte.
42	Bestimme die aktuelle Länge der konvexen Kette.
47-50	Initialisiere die übrigen Variablen. Gesucht werden die Indizes des ersten und des letzten sichtbaren Punktes.
51-64	Überspringe am Anfang alle Punkte, die vom neuen Punkt aus sichtbar sind. Bricht die Schleife ab, so ist der letzte Punkt nicht mehr sichtbar.
68-82	Überspringe alle Punkte, die nicht sichtbar sind. Bricht die Schleife ab, ist der aktuelle Index der des ersten sichtbaren Punktes.

Zeilen	Funktionalität
85-98	War am Anfang kein Punkt sichtbar, so wird ab dem ersten sichtbaren weitergesucht, bis ein nicht sichtbarer Punkt gefunden wird.
102-107	Liegt der erste sichtbare Punkt hinter dem letzten, wird das Array `conv_chain` beim Index `last_viewable` durchgeschnitten und beide Hälften vertauscht.
110-119	Füge den neuen Punkt in die Triangulierung ein. Dabei wird eine Kante zu jedem sichtbaren Punkt gezogen. Außerdem werden für k sichtbare Punkte $k-1$ Dreiecke eingefügt. Anschließend werden alle Punkte zwischen dem ersten und dem letzten sichtbaren aus der konvexen Kette entfernt und der neue Punkt eingefügt.
125	Entferne alle nicht genutzten Einträge von `triangulation`.
128-136	Entferne alle Dreiecke aus der Triangulierung, die einen der beiden Hilfspunkte beinhalten.
140	Verringere alle Einträge in `triangulation` um 2, um das Wegfallen der Hilfspunkte auszugleichen.
143	Entferne die beiden zusätzlich eingefügten Hilfspunkte.

1.2.3 Nachbearbeitung der Triangulierung

Nach Ausführen des *planesweep* haben wir nun eine Triangulierung T von P unter N. Bevor wir uns dem Verbessern der Triangulierung widmen, werden zwei weitere Funktionen ausgeführt.

Nachbearbeitung 1.27 (`assure_orientation`)
In der Triangulierung sollen sämtliche Dreiecke so angegeben werden, dass die Eckpunkte im mathematisch positiven Sinne aufgezählt werden. Dies ist in der durch den *planesweep* erzeugten Triangulierung nicht notwendigerweise der Fall.
Die Orientierung eines Dreiecks ABC lässt sich leicht über die Determinante einer Matrix bestimmen. Die Eckpunkte seien gegeben durch

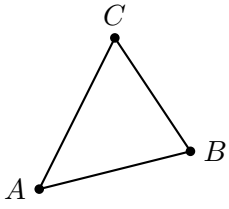
$$A = \begin{pmatrix} x_A \\ y_A \end{pmatrix}, \qquad B = \begin{pmatrix} x_B \\ y_B \end{pmatrix} \quad \text{und} \quad C = \begin{pmatrix} x_C \\ y_C \end{pmatrix}.$$

Dann betrachten wir die Matrix

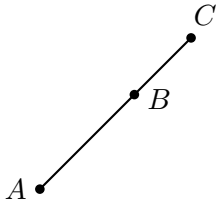
$$M = \begin{pmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{pmatrix} \tag{1.3}$$

Allgemein gibt es für die Determinante der Matrix M folgende drei Fälle.

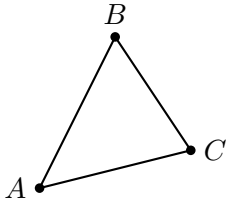
Fall 1: $\det(M) > 0$



Fall 2: $\det(M) = 0$



Fall 3: $\det(M) < 0$



Da der zweite Fall im Planesweep-Algorithmus nicht auftritt, kann er hier vernachlässigt werden. Unser Ziel ist es, dass für jedes Dreieck der Triangulierung die wie in Gleichung 1.3 aufgestellte Matrix eine positive Determinante hat.

Programm 1.6: Überprüfen der Orientierung aller Dreiecke

```
1  function [triangulation] = assure_orientation(coordinates, ...
2                                                triangulation)
3
4    for i = 1:size(triangulation,1)
5      if det([coordinates(triangulation(i,1), :),1; ...
6              coordinates(triangulation(i,2), :),1; ...
7              coordinates(triangulation(i,3), :),1]) < 0
8        tmp = triangulation(i,2);
9        triangulation(i,2) = triangulation(i,3);
10       triangulation(i,3) = tmp;
11     end
12   end
13
14 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
4	Durchlaufe alle erzeugten Dreiecke.
5-7	Bestimme die nach Gleichung 1.3 zum aktuellen Dreieck gehörende Matrix und berechne ihre Determinante.
8-10	Ist die bestimmte Determinante negativ, so werden im aktuellen Dreieck der zweite und dritte Punkt vertauscht; dadurch ändert sich die Orientierung.

Nachbearbeitung 1.28 (`pnpoly`)
Die durch die Punktemenge P und die Nebenbedingung N charakterisierte Menge sei nicht konvex oder habe Löcher. Dann werden während der Bestimmung einer Triangulierung T von P unter N mehr Dreiecke erzeugt, als im Endeffekt benötigt werden. Um Rechenaufwand zu sparen, werden wir im weiteren Verlauf nur noch Dreiecke bearbeiten, die im Innern des Polygonzugs der Nebenbedingung liegen, die also tatsächlich Teil des zu triangulierenden Gebiets sind.

Beispiel 1.29 In Beispiel 1.25 liegen sowohl das Dreieck `6 7 4` (ausgenommen durch das „Loch“ im Polygonzug) also auch das Dreieck `9 10 8` (ergänzt Nebenbedingung zur konvexen Hülle) außerhalb des Polygonzugs. Diese werden für die weitere Bearbeitung aus der Triangulierung gelöscht.

```
coordinates = [ 0    1        triangulation = [ 2  1  3
                0    3                          3  4  2
                1    0                          4  5  2
                1    2                          3  6  4
                1.5  3.5                        4  7  5
                2    1                          6  8  7
                2    2                          7  8  5
                2.5  3                          3  9  6
                3    1                          6  9  8
                3    4 ]                        8 10  5 ]
```

Damit ergibt sich das in Abbildung 1.15 dargestellte Bild. Nur die blau eingefärbten Dreiecke werden im weiteren Verlauf des Programms betrachtet. Die Koordinatenachsen und Punktbeschriftungen sind zur Verbesserung der Übersichtlichkeit weggelassen.

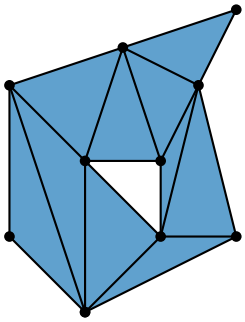


Abbildung 1.15: nur noch Dreiecke innerhalb des Polygons

Um diese Dreiecke zu markieren, wird der *Point Inclusion in Polygon Test* nach W. Randolph Franklin durchgeführt. Dieser betrachtet stets ein durch einen Polygonzug eingeschlossenes Gebiet und einen Punkt. Dabei wird entschieden, ob sich der Punkt innerhalb des Polygonzugs befindet.
Wir wollen bestimmen, ob sich ein Dreieck innerhalb Polygons befindet. Um den *Point Inclusion in Polygon Test* anwenden zu können, benötigen wir dementsprechend zuerst für jedes Dreieck ein Punkt aus dessen Innerem. Liegt dieser Punkt im Polygon, muss sich auch das ganze Dreieck darin befinden.

Der *Point Inclusion in Polygon Test* verläuft nach folgendem Schema: Wir wollen bestimmen, ob sich ein Punkt S aus $\mathbb{R}^2$ innerhalb des Polygonzugs befindet. Dafür betrachten wir eine Parallele zur x-Achse, die durch den Punkt S verläuft. Es werden die Schnittpunkte mit den Kanten des Polygons gezählt, welche sich *links* vom Punkt befinden. Ist die Anzahl der Schnittpunkte gerade, liegt der Punkt im Polygon, andernfalls liegt er außerhalb.

Beispiel 1.30 Für die in Abbildung 1.16 eingezeichneten Punkte S_1 und S_2 soll bestimmt werden, ob sie innerhalb oder außerhalb des Polygons liegen.

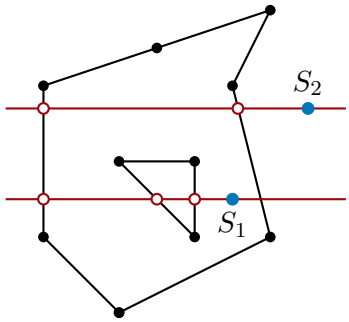


Abbildung 1.16: Beispiel *Point Inclusion in Polygon Test*

Die Parallele zur x-Achse, die durch S_1 läuft, schneidet den Polygonzug links von S_1 in drei Punkten. Der Punkt S_1 liegt daher im umschlossenen Gebiet.
Die Parallele durch S_2 schneidet den Polygonzug links von S_2 dagegen nur in zwei Punkten - S_2 liegt außerhalb des Gebiets.

Das Zählen der Schnittpunkte werden wir im folgenden Algorithmus nur modulo 2 vornehmen. Zu Beginn liegt jeder Punkt außerhalb des Polygonzugs. Jedes Mal, wenn wir einen der beschriebenen Schnittpunkte finden, *negieren* wir den Positionsmarker des entsprechenden Punktes. Das Negieren ist dabei im logischen Sinne zu verstehen: Aus 0 wird durch Negation 1 und umgekehrt.

Programm 1.7: pnpoly.m

```
1  function [is_inside] = pnpoly(coordinates, triangulation, boundary)
2    % überprüft zu jedem Dreieck der Triangulierung, ob es innerhalb
3    % des Polygonzugs liegt
4    %
5    % nach: PNPOLY - Point Inclusion in Polygon Test
6    % W. Randolph Franklin (WRF)
7    %
8    % Copyright (c) 1970-2003, Wm. Randolph Franklin
9
10   points = get_point_in_triangle(coordinates, triangulation);
11
12   is_inside = zeros(size(points,1),1);
13
14   [edges_start, edges_end] = find(boundary);
15   edges_start = coordinates(edges_start,:);
16   edges_end = coordinates(edges_end,:);
17
18   for j = 1:size(points,1)
19     for i = 1:size(edges_start,1)
20       if (edges_start(i,2) > points(j,2)) ...
21           ~= (edges_end(i,2) > points(j,2))
22         if points(j,1) < (edges_end(i,1)-edges_start(i,1)) * ...
23            (points(j,2) - edges_start(i,2)) ...
24            / (edges_end(i,2) - edges_start(i,2)) + edges_start(i,1)
25
26             is_inside(j) = ~ is_inside(j);
27         end
28       end
29     end
30   end
31
32 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
10	Zu jedem Dreieck wird ein Punkt aus seinem Innern bestimmt. Die hierfür verwendete Hilfsfunktion 1.31 wird als Hilfsfunktion 1.31 erläutert.
12	Markiere jeden Punkt als außerhalb des Polygons, indem ein Nullvektor als Marker erzeugt wird.
14-16	Speichere die Koordinaten aller Kantenanfänge und -enden.
ab 18	Durchlaufe alle Punkte.
ab 19	Durchlaufe alle Kanten.
20-21	Überprüfe, ob die y-Koordinate des aktuellen Punktes zwischen den y-Koordinaten des aktuellen Kantenanfangs und -endes liegt.

Zeilen	Funktionalität
22-27	Ist die Abfrage aus den Zeilen 20-21 erfüllt, überprüfe, ob eine Parallele zur x-Achse die Kante links des aktuellen Punktes schneidet. Ist dies der Fall, negiere den Positionsmarker des Punktes.

Hilfsfunktion 1.31 (`get_point_in_triangle`)
Wir bestimmen zu jedem Dreieck einen Punkt innerhalb des Dreiecks, indem wir die Koordinaten der Eckpunkte mitteln. Die Position dieser Punkte wird anschließend mit `pnpoly` bestimmt.

Programm 1.8: get_point_in_triangle.m

```
1 function [points] = get_point_in_triangle(coordinates, triangulation)
2   points = zeros(size(triangulation,1),2);
3
4   for i = 1:size(triangulation,1)
5     points(i,:) = (coordinates(triangulation(i,1),:) + ...
6                    coordinates(triangulation(i,2),:) + ...
7                    coordinates(triangulation(i,3),:)) ./ 3;
8   end
9 end
```

Bemerkung 1.32 (nicht zusammenhängendes Gebiet) Offensichtlich können wir mit der `pnpoly`-Funktion sogar eine Triangulierung für ein nicht zusammenhängendes Gebiet erstellen. Liegen dabei beispielsweise zwei getrennte Gebiete vor, so werden diese nicht durch Dreiecke der Triangulierung verbunden.

Beispiel 1.33 Wir betrachten in den folgenden Abbildungen die Triangulierung zweier nicht zusammenhängende Gebiete, die gleichzeitig als Nebenbedingung übergeben werden.
In Abbildung 1.17 wird die Nebenbedingung gezeigt. Wir betrachten unser bisheriges Beispiel, welches um einen zweiten Polygonzug erweitert wurde. Abbildung 1.18 zeigt das Ergebnis der Triangulierung durch den *planesweep*. Die Hilfspunkte wurden noch nicht entfernt. Dabei zeigen die blauen Kanten weiterhin die konvexe Hülle, die grünen Kanten sind die nicht von der konvexen Hülle überdeckten Kanten der Nebenbedingung. Anschließend zeigt Abbildung 1.19 die Triangulierung, mit der in den nächsten Kapiteln weiter gearbeitet wird. Dabei sind erneut nur die blau hinterlegten Dreiecke Teil der Triangulierung.

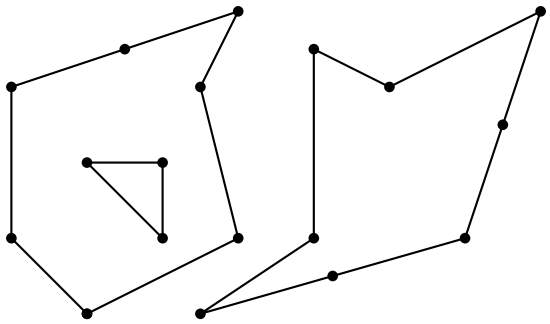


Abbildung 1.17: Beispiel für eine Nebenbedingung eines nicht zusammenhängenden Gebiets

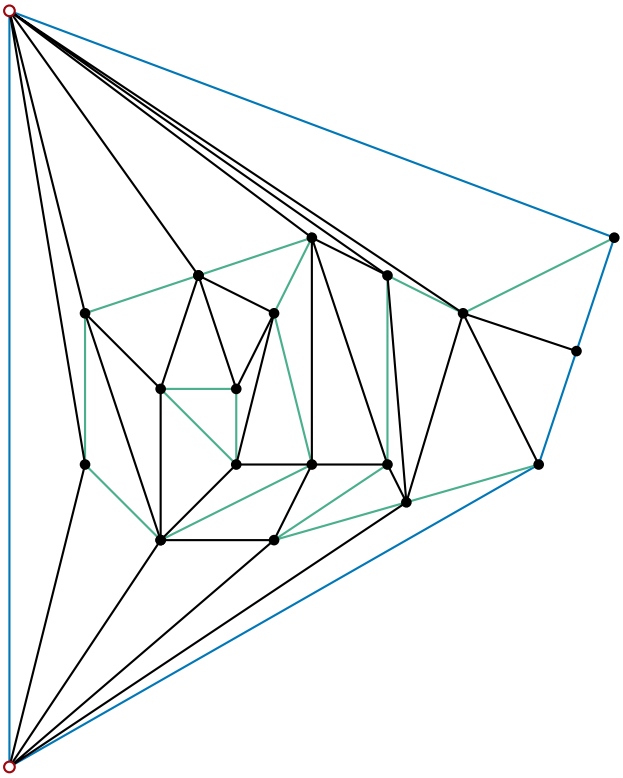


Abbildung 1.18: erzeugte Triangulierung vor entfernen der Hilfspunkte

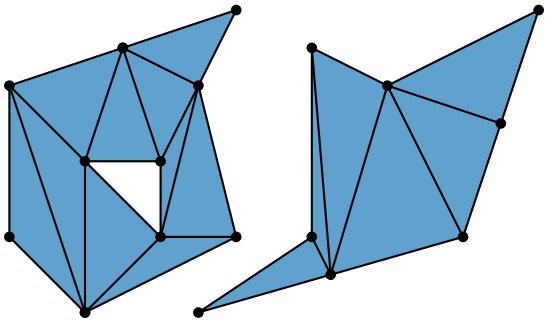


Abbildung 1.19: erzeugte Triangulierung nach Ausführen von `pnpoly`

2 Bedingte Delaunay-Triangulierung

Im letzten Kapitel haben wir gesehen, wie durch den *planesweep*-Algorithmus eine Triangulierung T einer Punktemenge P unter einer Nebenbedingung N erzeugt wird. Die Dreiecke dieser Triangulierung sind im Allgemeinen sehr spitzwinklig. Durch den in diesem Kapitel beschriebenen *edge-flip*-Algorithmus lassen sich kleine Winkel gegebenenfalls vergrößern. Dabei wird die Triangulierung T in eine sogenannte *Delaunay-Triangulierung* überführt.

Als Quellen wurden [Ed2001] Kapitel 1 und 2, [dB2000] Kapitel 9 sowie [MG2005] Kapitel 5 verwendet.

2.1 Theoretische Grundlagen

Wir werden in diesem Kapitel zunächst definieren, was eine Delaunay-Triangulierung ist. Anschließend werden wir zeigen, dass die Delaunay-Triangulierung zu einer gegebenen Punktemenge P den kleinsten vorkommenden Winkel maximiert. Dies vermeidet beim Rechnen auf dem Netz Rundungsfehler.

Definition 2.1 (leerer Kreis) Ein Kreis um eine Kante AB heißt *leer*, wenn er außer A und B keine weiteren Punkte enthält. Analog heißt ein Kreis um ein Dreieck ABC leer, wenn er außer A, B und C keinen Punkt enthält. Wir bezeichnen einen Kreis um ein Dreieck ABC als *bedingt leer*, wenn er leer ist oder alle weiteren Punkte im Kreis auf Grund der Nebenbedingung aus dem Inneren des Dreiecks nicht sichtbar sind.
Ein Beispiel zeigt Abbildung 2.1.

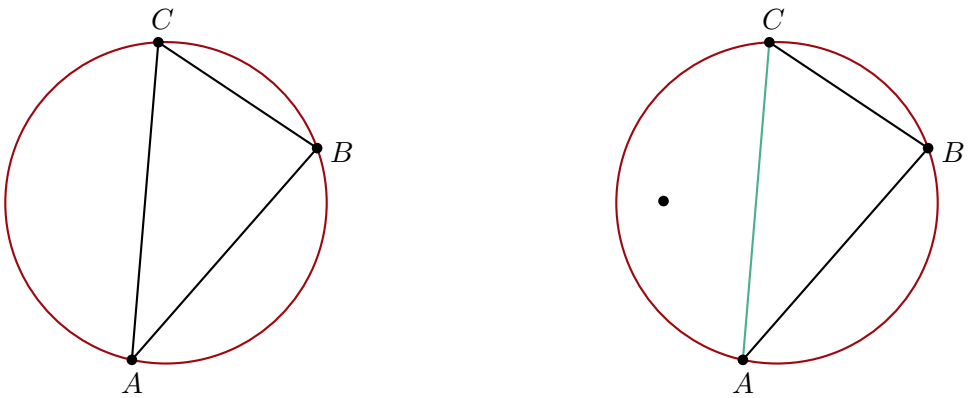


Abbildung 2.1: leerer Kreis, bedingt leerer Kreis mit CA als Teil der Nebenbedingung

Mit Hilfe dieser Definition werden wir nun die Delaunay-Triangulierung charakterisieren.

Definition 2.2 (Delaunay-Triangulierung) Gegeben sei eine Triangulierung T einer Punktemenge P. Wir nennen T eine *Delaunay-Triangulierung*, wenn der Umkreis jedes Dreiecks aus T leer ist.

Sei T die Triangulierung von P unter einer nichtleeren Nebenbedingung N. Dann nennen wir T eine *bedingte Delaunay-Triangulierung*, wenn der Umkreis jedes Dreiecks aus T bedingt leer ist.

Um eine beliebige Triangulierung in eine Delaunay-Triangulierung zu überführen, benötigen wir ein äquivalentes Kriterium, welches nicht die Dreiecke sondern deren Kanten betrachtet.

Definition 2.3 Eine Kante AB heißt *lokal Delaunay*, falls

(i) sie nur zu einem Dreieck ABC und damit zur konvexen Hülle oder der Nebenbedingung gehört oder

(ii) sie zu zwei Dreiecken ABC und BAD gehört und D nicht im Umkreis von ABC liegt.

Die zweite Bedingung nennen wir auch *Umkreisbedingung*.

Über diese Definition lässt sich das folgende Lemma formulieren.

Lemma 2.4 (Delaunay-Lemma) *Gegeben sei eine Triangulierung T der Punktmenge P. Ist jede Kante aus T lokal Delaunay, so ist T eine Delaunay-Triangulierung.*

Für den Beweis des Delaunay-Lemmas benötigen wir die folgende Definition. Diese liefert einen Begriff für den Abstand eines Punktes zu einem Kreis.

Definition 2.5 (Abstand zum Kreisrand) Der *Abstand zum Kreisrand* eines Punktes $x \in \mathbb{R}^2$ von einem Kreis U mit Mittelpunkt u und Radius ρ ist definiert durch

$$\pi_U(x) = \|x - u\|^2 - \rho^2.$$

$\pi_U(x)$ liefert ein Maß dafür, wie weit außerhalb des Kreises U der Punkt x liegt. Offensichtlich ist $\pi_U(x)$ positiv, falls x außerhalb von U liegt, negativ, falls x innerhalb von U liegt und gleich null, falls x auf dem Kreis U liegt.

Beweis (zu Lemma 2.4): Gegeben sei ein beliebiges Dreieck ABC aus T sowie ein beliebiger, von A, B und C verschiedener Punkt S aus der Punktemenge P. Wir zeigen, dass S nicht im Umkreis von ABC liegt. Durch die beliebige Wahl von S ist damit der Umkreis von ABC leer; durch die beliebige Wahl von ABC ist T eine Delaunay-Triangulierung.
Wähle einen Punkt X aus $\mathbb{R}^2$ aus dem Inneren des Dreiecks ABC so, dass die Verbindungsstrecke von X nach S keinen Punkt aus P außer S selbst beinhaltet. Es sei $\tau_0, \tau_1, \ldots, \tau_k$ mit $\tau_0 = ABC$ die Sequenz der wie in Abbildung 2.2 von XS geschnittenen Dreiecke.

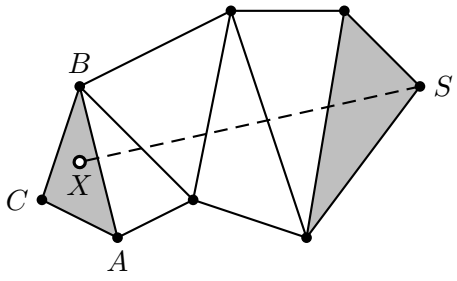


Abbildung 2.2: Sequenz aus Dreiecken aus T, die die Strecke XS schneiden

Wir bezeichnen mit $\pi_i(S)$ den Abstand von S zum Umkreis des Dreiecks τ_i gemäß Definition 2.5. Da sämtliche Kanten nach Voraussetzung lokal Delaunay sind, gilt

$$\pi_0(S) > \pi_1(S) > \ldots > \pi_k(S).$$

Da S eine der Ecken des Dreiecks τ_k ist und somit auf dessen Umkreis liegt, gilt $\pi_k(S) = 0$. Damit gilt $\pi_0(S) > 0$. Dies ist äquivalent zu der Aussage, dass S außerhalb des Umkreises von ABC liegt. □

Gegeben seien im Folgenden stets die Dreiecke ABC und BAD sowie deren gemeinsame Kante AB. Sei nun AB so, dass die Umkreisbedingung nicht erfüllt ist. Ersetzen wir diese Kante durch die Kante CD, so ist diese neue Kante lokal Delaunay. Diesen Vorgang nennen wir *flippen* der Kante AB. Abbildung 2.3 zeigt diesen Vorgang inklusive der beteiligten Umkreise.

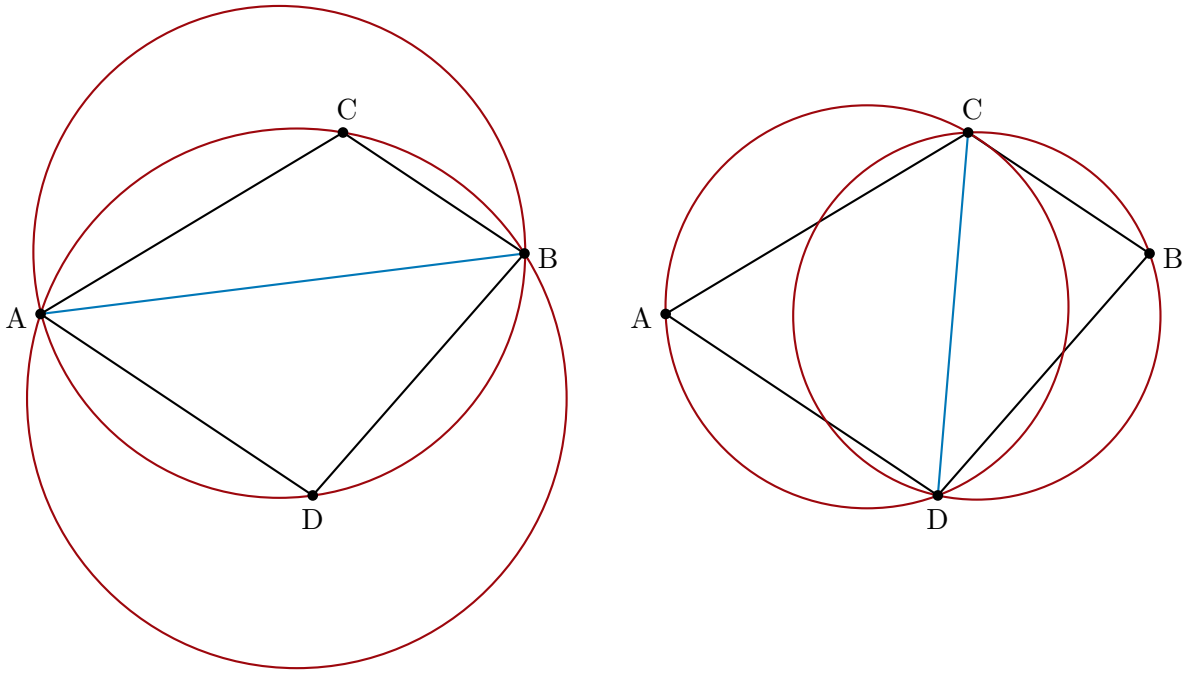


Abbildung 2.3: Kante AB verstößt gegen Umkreisbedingung, Kante CD ist lokal Delaunay

Definition 2.6 Eine Kante, die zwei Punkte auf einem Kreis verbindet, bezeichnen wir als *Sehne* des Kreises.

Satz 2.7 (Umkreiswinkelsatz des Euklid) *Eine Sehne eines Kreises wird von allen Punkten des Kreisrandes, die auf der gleichen Seite liegen, unter dem selben Winkel gesehen.*

Beweis: Seien die Winkelbezeichnungen wie in Abbildung 2.4.
Sei M der Umkreismittelpunkt des Dreiecks ABC. Sei die Kante AB die betrachtete Sehne. Zu zeigen ist dann, dass der Winkel in C, $\alpha + \gamma$, konstant ist.
Verbindet man beide Enden einer Kante des Dreiecks mit M, so ist das entstehende Dreieck gleichschenklig. Daher sind die beiden eingeschlossenen Winkel an den Punkten auf dem Kreis gleich groß.

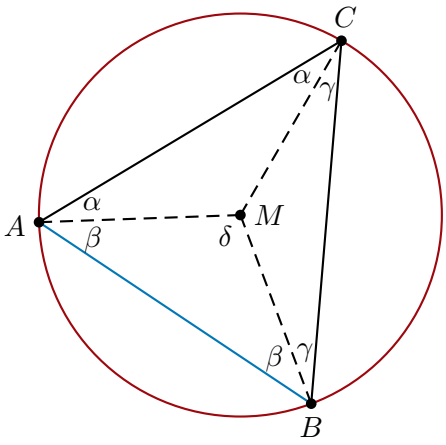


Abbildung 2.4: Winkelbezeichnungen zum Umkreiswinkelsatz

Im Bogenmaß beträgt die Winkelsumme im Dreieck stets π. Dementsprechend gilt

$$2\alpha + 2\beta + 2\gamma = \pi \quad \text{sowie} \quad 2\beta + \delta = \pi \quad \text{und damit} \quad \alpha + \gamma = \frac{\delta}{2} = \text{const.} \qquad \square$$

Korollar 2.8 *Es seien die Bezeichnungen wie in Abbildung 2.5. Ergänzt man die Sehne AB durch einen Punkt D_1 im Innern des Umkreises von ABC zum Dreieck ABD_1, so gilt für den entstehenden Winkel $\delta_1 > \gamma$. Wählt man stattdessen einen Punkt D_2 außerhalb des Umkreises, so gilt für den entstehenden Winkel $\delta_2 < \gamma$.*

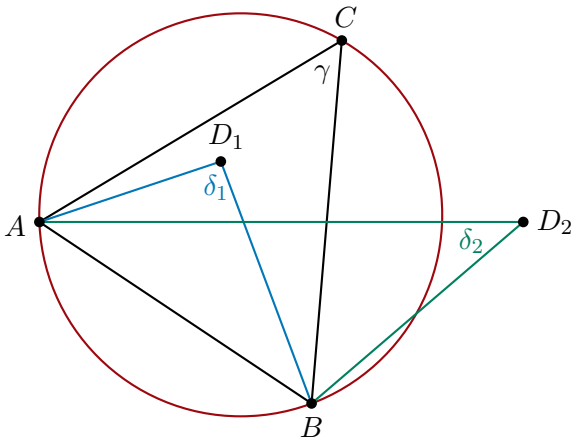


Abbildung 2.5: Winkelbezeichnungen zum Korollar des Umkreiswinkelsatzes

Definition 2.9 Ein Viereck, dessen Kanten Sehnen des gleichen Kreises sind, bezeichnen wir als *entartet*. Ein Beispiel ist in Abbildung 2.6 gegeben.

Seien für den *edge-flip* zwei Dreiecke ABC und BAD solcherart gegeben, dass das Viereck $ADBC$ entartet ist. Offensichtlich ist der Umkreis keines der vier möglichen Dreiecke ABC, BAD, DCA und CDB leer.
In diesem Fall macht es keinen Unterschied, ob die Dreiecke ABC und BAD oder DCA und CDB in der Triangulierung verwendet werden. Nach dem Umkreiswinkelsatz ist der kleinste

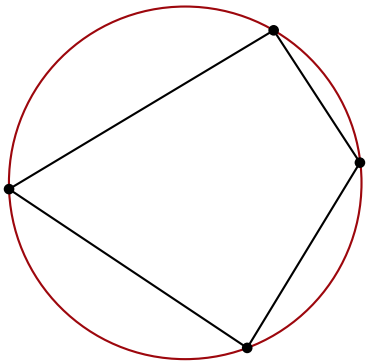


Abbildung 2.6: entartetes Viereck

Winkel in beiden Dreieckskombinationen jeweils gleich groß. Der *edge-flip* behandelt diesen Spezialfall deshalb als sei der Umkreis eines der Anfangsdreiecke leer.

Da demnach für ein entartetes Vierecke $ADBC$ sowohl AB als auch CD lokal Delaunay sind, kann die Delaunay-Triangulierung offensichtlich nicht eindeutig sein.

Satz 2.10 (Winkelmaximierung) *Gegeben seien die Dreiecke ABC und BAD, und es sei der kleinste Winkel eines der Dreiecke gleich φ. Ist AB nicht lokal Delaunay, so ist in den Dreiecken DCA und CDB der kleinste Winkel größer oder gleich φ.*

Beweis: Zu zeigen ist, dass sich jeder Winkel in den Dreiecken nach dem *edge-flip* durch einen Winkel aus den alten Dreiecken nach unten abschätzen lässt.

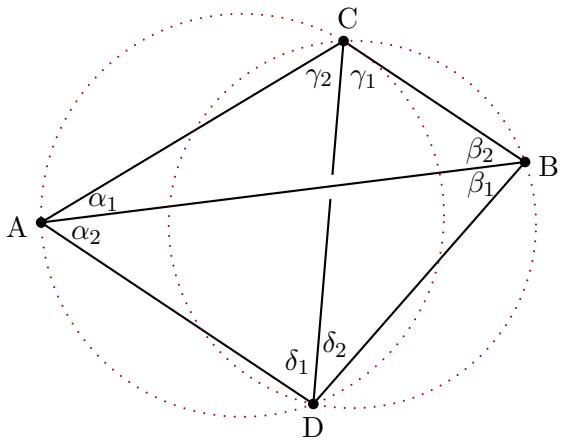


Abbildung 2.7: Winkel in den betroffenen Dreiecken

Es seien die Winkelbezeichnungen wie in Abbildung 2.7. Demnach haben die Winkel im Dreieck DCA die Größen δ_1, γ_2 und $\alpha_1 + \alpha_2$, die Winkel im Dreieck CDB die Größen γ_1, δ_2 und $\beta_1 + \beta_2$.
Offensichtlich gilt $\alpha_1 + \alpha_2 \geq \alpha_1$ sowie $\beta_1 + \beta_2 \geq \beta_1$. Noch zu bestimmen sind dementsprechend untere Schranken für γ_1, γ_2, δ_1 und δ_2.

Diese sind gegeben durch $\gamma_2 \geq \beta_1$, $\delta_1 \geq \beta_2$, $\gamma_1 \geq \alpha_2$ und $\delta_2 \geq \alpha_1$. Jedes dieser Winkelpaare aus je einer Ungleichung gehört zu zwei Dreiecken, die eine gemeinsame Kante haben. Da die Umkreise von DCA und CDB leer sind, folgt die Behauptung aus Korollar 2.8. □

Sind alle in der Triangulierung vorkommenden Kanten lokal Delaunay, so kann offensichtlich auch der kleinste Winkel nicht mehr durch flippen einer Kante vergrößert werden: Nach Korollar 2.8 müsste dafür ein nichtleerer Umkreis existieren.

2.2 Implementierung

Nach dem Delaunay-Lemma 2.4 ist eine Triangulierung T eine Delaunay-Triangulierung, wenn alle Kanten lokal Delaunay sind. Die Bedingung *lokal Delaunay* lässt sich nur sukzessive für die einzelnen Kante überprüfen. Deshalb benötigen wir eine Möglichkeit, noch nicht abgearbeitete oder erneut zu überprüfende Kanten zu markieren. Zu diesem Zweck führen wir die Variable `edge_marker` ein. In welcher Reihenfolge die Kanten hierbei angesprochen werden, werden wir im späteren Verlauf dieses Kapitels sehen.
Beim Markieren einer Kante `i` unterscheiden wir zwei mögliche Fälle.

- `edge_marker(i) = 1` falls die Kante noch überprüft werden muss und
- `edge_marker(i) = 0` falls die Kante überprüft und lokal Delaunay ist.

Dabei sind sämtliche Kanten der Nebenbedingung dank der Nachbearbeitung in Abschnitt 1.2.3 nur Teil eines Dreiecks. Diese sind nach Definition 2.3 lokal Delaunay. Wir können also sicher sein, dass Kanten der Nebenbedingung nicht verändert werden.

Der *edge-flip* funktioniert nach folgendem Vorgehen.

Pseudocode 2.11 (*edge-flip*)

---

while markierte Kante vorhanden **do**
  betrachte die erste markierte Kante AB
  if AB nicht lokal Delaunay **then**
    ersetze AB durch CD
    for all $XY \in \{AC, CB, BD, DA\}$ **do**
      falls XY nicht markiert, Markierung setzen
    end for
  end if
  entferne die Markierung von AB
end while

2.2.1 Aufbau benötigter Datenstrukturen

Um den *edge-flip*-Algorithmus effizient zu implementieren, benötigen wir außer den vorhanden Punkten, Kanten und Dreiecken einige weitere Datenstrukturen.
In diesem Abschnitt sollen vier verschiedene Datenstrukturen vorgestellt werden. Die erste dient dabei lediglich der Erzeugung der weiteren Strukturen. Die zweite und dritte Datenstruktur ermöglichen einen schnellen Zugriff von Kanten auf die beteiligten Dreiecke und umgekehrt von

den Dreiecken auf die beteiligten Kanten. Außerdem benötigen wir die Längen aller Kanten - auch diese werden wir im Voraus bestimmen.

Datenstruktur 2.12 (`edges`)
Durch diese Funktion wird eine erste Verbindung zwischen Dreiecken und Kanten erzeugt.
Es sei n die Anzahl der Punkte aus P. Die Variable `edges` ist eine $n \times n$ *sparse* Matrix. Diese hat für `i>j` an der Position `(i,j)` den Index des Dreiecks, das die Kante von `i` nach `j` beinhaltet, gespeichert. Ist die Kante Teil zweier Dreiecke, so wird der Index des zweiten Dreiecks an der Position `(j,i)` gespeichert.

Wir betrachten nun den Quellcode des Programms, das diese Datenstruktur erzeugt.

Programm 2.1: generate_edges.m

```
1  function [edges] = generate_edges(triangulation)
2
3    size_edges = max(triangulation(:));
4
5    edges = sparse(size_edges, size_edges);
6    j_succ = [2 3 1];
7
8    for i = 1:size(triangulation,1)
9      for j = 1:3
10       ii = max(triangulation(i,j), triangulation(i, j_succ(j)));
11       jj = min(triangulation(i,j), triangulation(i, j_succ(j)));
12       if edges(ii,jj)
13         edges(jj,ii) = i;
14       else
15         edges(ii,jj) = i;
16       end
17     end
18   end
19
20 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
3-5	Finde die höchste vorkommende Zahl in `triangulation` - dies ist der Index des letzten Punktes - anschließend wird die *sparse* Matrix `edges` als Nullmatrix initialisiert.
8	Durchlaufe alle Dreiecke.
9	Betrachte in jedem Dreieck ABC die Kanten AB, BC und CA - hierfür wird die Variable `j_succ` benötigt.
10-11	Finde für jede Kante im Dreieck jeweils den größeren und den kleineren Eintrag.
12-13	Trage für eine Kante `(ii,jj)` mit `ii>jj` an der Position `(ii,jj)` in `edges` den Index des Dreiecks ein - ist schon ein Eintrag vorhanden, so trage den Index des Dreieck in `(jj,ii)` ein.

Durch diesen Aufbau der Variablen `edges` können wir auch Randkanten sofort erkennen: Gehört eine Kante `(i,j)` mit `i>j` nur zu einem Dreieck, hat `edges` an der Position `(j,i)` keinen Eintrag. Wir betrachten dies erneut an unserem Beispiel aus Kapitel 1.

Beispiel 2.13 In der Abbildung 2.8 geben die roten, eingekreisten Ziffern innerhalb der Dreiecke jeweils den Index des Dreiecks, also die Zeile in `triangulation`, an.
Zur Erinnerung: Die Dreiecke `6 7 4` und `9 10 8` wurden nach dem *planesweep* aus der Triangulierung entfernt. Sie gehören auf Grund der Nebenbedingung nicht zu dem Gebiet, das zu triangulieren ist.

Der Übersichtlichkeit halber wird in zukünftigen Abbildungen zu diesem Beispiel das Koordinatensystem nicht mehr angezeigt. Die schwarzen Ziffern sind weiterhin die Indizes der Punkte.

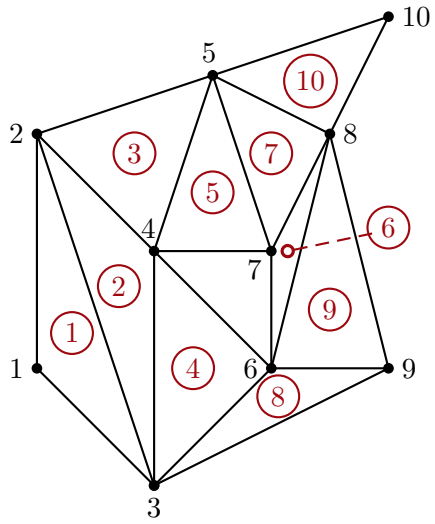


Abbildung 2.8: Triangulierung der Punkte

Wir betrachten dazu den Zusammenhang zwischen der Triangulierung und der Belegung der Variablen `edges`.

```
triangulation = [ 1  3  2        edges = [ 0  0  0  0  0  0  0  0  0  0
                  3  4  2                  1  0  2  3  0  0  0  0  0  0
                  4  5  2                  1  1  0  4  0  8  0  0  0  0
                  3  6  4                  0  2  2  0  5  0  5  0  0  0
                  4  7  5                  0  3  0  3  0  0  7 10  0  0
                  6  8  7                  0  0  4  4  0  0  6  9  9  0
                  7  8  5                  0  0  0  0  5  0  0  7  0  0
                  3  9  6                  0  0  0  0  7  6  6  0  0 10
                  6  9  8                  0  0  8  0  0  8  0  9  0  0
                  8 10  5 ]                0  0  0  0 10  0  0  0  0  0 ]
```

Datenstruktur 2.14 (`edges_with_triangles`)
Jede Kante, die zwei Punkte aus P verbindet und zur Triangulierung T gehört, ist Teil von einem oder zwei Dreiecken aus T. Um zügig auf beide Dreiecke zugreifen zu können, wird die Datenstruktur `edges_with_triangles` angelegt.
Gibt es k Kanten, so ist `edges_with_triangles` eine $k \times 6$ Matrix. Zu jeder Kante werden die beiden betroffenen Dreiecke sowie die Position der Kante im entsprechenden Dreieck gespeichert. Jede Kantenposition wird dabei über die Position ihres Anfangspunktes festgelegt. Ist also das Dreieck ABC gegeben, so hat die Kante AB die Position 1, die Kante BC die Position 2 und die Kante CA die Position 3.

Die Kante AB sei Teil der Dreiecke ABC und BAD. Dann ergibt sich die im folgenden beschriebene Struktur. Dabei seien die beiden Einträge `A` und `B` die Indizes der entsprechenden Punkte in `coordinates`. Sowohl `index_ABC` als auch `index_BAD` sind Indizes der Dreiecke in `triangulation`.

```
edges_with_triangles = [ A B index_ABC pos_AB_in_ABC index_BAD pos_AB_in_BAD ]
```

Beim Erstellen dieser Datenstruktur wird nun auch die Indizierung der Kanten klar: Wir finden die Kante `i` in Position `edges_with_triangles(i,:)`.

Beispiel 2.15 Die Dreiecksindizierung ist analog zu Abbildung 2.8. Die Kantenindizierung ist aus Abbildung 2.9 ersichtlich.

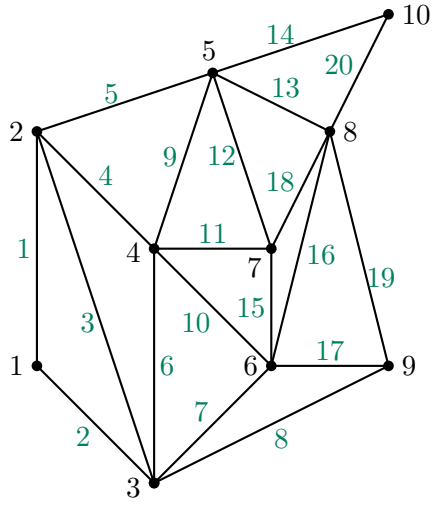


Abbildung 2.9: Indizierung der Kanten

Die entsprechende Datenstruktur ergibt sich wie folgt.

```
triangulation = [ 1  3  2
                  3  4  2
                  4  5  2
                  3  6  4
                  4  7  5
                  6  8  7
                  7  8  5
                  3  9  6
                  6  9  8
                  9 10  8
                  8 10  5 ]
```

```
edges_with_triangles = [ 2  1  1  3  0  0
                         3  1  1  1  0  0
                         3  2  1  2  2  3
                         4  2  2  2  3  3
                         5  2  3  2  0  0
                         4  3  2  1  4  3
                         6  3  4  1  8  3
                         9  3  8  1  0  0
                         5  4  3  1  5  3
                         6  4  4  2  0  0
                         7  4  5  1  0  0
                         7  5  5  2  7  3
                         8  5  7  2 10  3
                        10  5 10  2  0  0
                         7  6  6  3  0  0
                         8  6  6  1  9  3
                         9  6  8  2  9  1
                         8  7  6  2  7  1
                         9  8  9  2  0  0
                        10  8 10  1  0  0 ]
```

Offensichtlich erkennen wir die Kanten der Nebenbedingung daran, dass sie in der fünften und sechsten Spalte von `edges_with_triangles` keinen Eintrag ungleich null haben.

Wir betrachten nun wieder die Implementierung der entsprechenden Funktion.

Programm 2.2: triangle2edge.m

```
function [edges_with_triangles] = triangle2edge(triangulation, edges)
  % bearbeite erst die untere Dreiecksmatrix von edges
  [edge_start, edge_end, triangle] = find(tril(edges));

  edges_with_triangles = zeros(size(edge_start,1),6);

  for i = 1:size(edge_start)
    % trage 1. Dreieck der Kante ein
    edges_with_triangles(i, 1:3) = [edge_start(i), edge_end(i), ...
                                                        triangle(i)];
    % trage ein, die wievielte Kante im Dreieck es ist: im Dreieck ABC
    % heißt das: AB: 1; BC: 2; CA: 3
    if triangulation(triangle(i), 1) == edge_start(i)
      if triangulation(triangle(i), 2) == edge_end(i)
        % Kante zwischen 1. und 2. Punkt
        edges_with_triangles(i, 4) = 1;
      else
        % Kante zwischen 3. und 1. Punkt
        edges_with_triangles(i, 4) = 3;
      end
    elseif triangulation(triangle(i), 2) == edge_start(i)
      if triangulation(triangle(i), 3) == edge_end(i)
        % Kante zwischen 2. und 3. Punkt
        edges_with_triangles(i, 4) = 2;
      else
        % Kante zwischen 2. und 1. Punkt
        edges_with_triangles(i, 4) = 1;
      end
    else
      if triangulation(triangle(i), 1) == edge_end(i)
        % Kante zwischen 3. und 1. Punkt
        edges_with_triangles(i, 4) = 3;
      else
        % Kante zwischen 2. und 3. Punkt
        edges_with_triangles(i, 4) = 2;
      end

    end

    %entweder hier ist ein Eintrag, oder es wird 0 eingetragen
    edges_with_triangles(i, 5) = edges(edge_end(i), edge_start(i));
    if edges_with_triangles(i, 5)
      % trage ein, die wievielte Kante im Dreieck es ist: im Dreieck ABC
      % heißt das: AB: 1; BC: 2; CA: 3
      if triangulation(edges_with_triangles(i, 5), 1) == edge_start(i)
        if triangulation(edges_with_triangles(i, 5), 2) == edge_end(i)
          % Kante zwischen 1. und 2. Punkt
          edges_with_triangles(i, 6) = 1;
```

```
49          else
50            % Kante zwischen 3. und 1. Punkt
51            edges_with_triangles(i, 6) = 3;
52          end
53        elseif triangulation(edges_with_triangles(i, 5), 2) ...
54                                                    == edge_start(i)
55          if triangulation(edges_with_triangles(i, 5), 3) == edge_end(i)
56            % Kante zwischen 2. und 3. Punkt
57            edges_with_triangles(i, 6) = 2;
58          else
59            % Kante zwischen 2. und 1. Punkt
60            edges_with_triangles(i, 6) = 1;
61          end
62        else
63          if triangulation(edges_with_triangles(i, 5), 1) == edge_end(i)
64            % Kante zwischen 3. und 1. Punkt
65            edges_with_triangles(i, 6) = 3;
66          else
67            % Kante zwischen 2. und 3. Punkt
68            edges_with_triangles(i, 6) = 2;
69          end
70        end
71      end
72    end
73
74  end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
3	Zu jeder Kante, die in der linken unteren Dreiecksmatrix von `edges` gespeichert ist, werden Indizes von Start- und Endpunkt sowie der Index des entsprechenden Dreiecks gespeichert. Es genügt dabei, die linke untere Dreiecksmatrix zu betrachten, da in der rechten oberen Hälfte keine zusätzlichen Kanten vermerkt sind. Dies wurde durch die Hilfsfunktion 2.12 (`generate_edges`) sichergestellt.
5	Die Größen der in Zeile 3 bestimmten Vektoren geben die Anzahl der Kanten an. Erzeuge eine sechsspaltige Datenstruktur mit entsprechend vielen Zeilen.
7	Durchlaufe alle Kanten, die in Zeile 3 gespeichert wurden.
9	In die ersten drei Spalten werden die Indizes von Start- und Endpunkt sowie der Index des Dreiecks gespeichert. Da aktuell nur die Kanten bearbeitet werden, die in `edges` unterhalb der Diagonalen gespeichert waren, ist jeweils der Index des Startpunkts größer als der des Endpunkts.
13-36	Durch Vergleichen von Start- und Endpunkt mit den Einträgen in `triangulation` zum entsprechenden Dreieck wird herausgefunden, ob die Kante zwischen dem ersten und zweiten Punkt, zwischen dem zweiten und dritten Punkt oder zwischen dem dritten und ersten Punkt liegt. Dies wird entsprechend in der vierten Spalte von `edges_with_triangles` gespeichert.

Zeilen	Funktionalität
41	Ist die aktuelle Kante nicht Teil der Nebenbedingung, also Teil von zwei Dreiecken, so hat `edges` nicht nur an Position `(i,j)` mit `i>j` einen Eintrag, sondern auch an `(j,i)`. Dies ist der Index des zweiten Dreiecks und wird in der fünften Spalte gespeichert. Oberhalb der Diagonalen ist keine Kante gespeichert, deren Pendant nicht auch unterhalb der Diagonalen zu finden ist. Daher ist kein weiterer `find`-Befehl nötig.
42	Befindet sich nun an der soeben gespeicherten Stelle ein Eintrag ungleich null, so ist die Kante keine Randkante, sondern Teil zweier Dreiecke.
45-70	Hier wird analog zu den Zeilen 12-35 wieder die Position der entsprechenden Kante im zweiten Dreieck bestimmt, falls keine Randkante vorliegt.

Datenstruktur 2.16 (`triangles_with_edges`)
Es ist nicht nur wichtig, von jeder Kante schnell auf die betroffenen Dreiecke zugreifen zu können. Weiter muss auch der Zugriff von einem Dreieck auf die entsprechenden Kanten zügig möglich sein.
Für diesen Zweck wird die Datenstruktur `triangles_with_edges` angelegt. In dieser werden zu jedem Dreieck mit dem jeweils gleichen Index die Indizes dreier Kanten abgelegt. Dieser Index bezieht sich auf die entsprechende Kante in `edges_with_triangles`. Dabei steht die Kante vom ersten zum zweiten Punkt des Dreiecks an erster, die vom zweiten zum dritten Punkt an zweiter und die vom dritten zum ersten Punkt an dritter Stelle. Diese Position entspricht den Einträgen von `edges_with_triangles` in der vierten beziehungsweise sechsten Spalte.

Wir betrachten erneut unser Beispiel.

Beispiel 2.17 In der folgenden Abbildung 2.10 findet sich im linken Bild die Indizierung der Dreiecke, im rechten die Indizierung der Kanten.

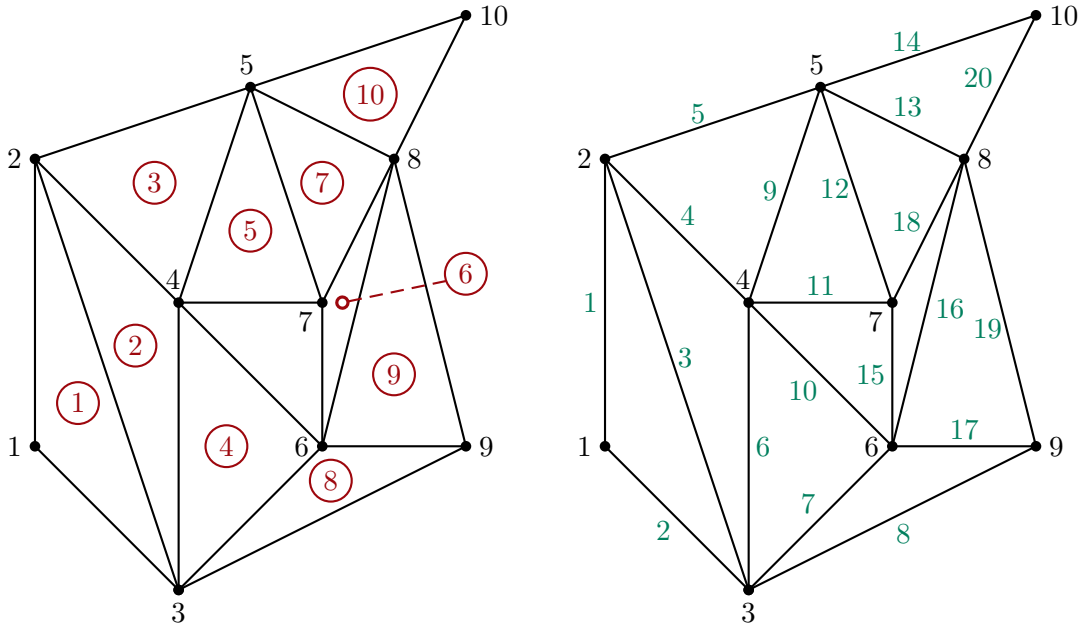


Abbildung 2.10: Verbindung zwischen Dreiecks- und Kantenindizierung

Wir erhalten durch diese Funktion als neue Variable `triangles_with_edges`. Zum Vergleich sei außerdem noch die Variable `triangulation` angegeben.

```
triangulation = [ 2  1  3
                  3  4  2
                  4  5  2
                  3  6  4
                  4  7  5
                  6  8  7
                  7  8  5
                  3  9  6
                  6  9  8
                  8 10  5 ]
```

```
triangles_with_edges = [ 1  2  3
                         6  4  3
                         9  5  4
                         7 10  6
                        11 12  9
                        16 18 15
                        18 13 12
                         8 17  7
                        17 19 16
                        20 14 13 ]
```

Über diese beiden Variablen können wir schnelle Verbindungen ziehen. Wir betrachten beispielsweise das Dreieck 5. Dieses hat die Eckpunkte 4, 7 und 5. Laut `triangles_with_edges` erhalten wir folgende Verbindungen:

- Kante 11 geht von Punkt 4 nach Punkt 7
- Kante 12 geht von Punkt 7 nach Punkt 5
- Kante 9 geht von Punkt 5 nach Punkt 4

Programm 2.3: Erzeugen von `triangles_with_edges`

```
1  function [triangles_with_edges] = edge2triangle(edges_with_triangles)
2
3    cnt_triangles = max(max(edges_with_triangles(:, 3:6)));
4    triangles_with_edges = zeros(cnt_triangles, 3);
5
6    for i = 1:size(edges_with_triangles,1)
7      triangles_with_edges(edges_with_triangles(i, 3),...
8        edges_with_triangles(i,4)) = i;
9      if edges_with_triangles(i,5)
10       triangles_with_edges(edges_with_triangles(i, 5),...
11         edges_with_triangles(i,6)) = i;
12     end
13   end
14
15 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
3	Die höchste Zahl aus der 3. und 5. Spalte von `edges_with_triangles` gibt die Anzahl an Dreiecken an
4	Initialisierung der Variablen `triangles_with_edges` - diese hat die gleichen Dimensionen wie `triangulation`.
6	Durchlaufe alle Kanten.
7 - 12	Trage in `triangles_with_edges` für das angegebene Dreieck und die angegebene Position die aktuelle Kantennummer ein. Sind zwei Dreiecke für die entsprechende Kante angegeben, verfahre mit dem zweiten Dreieck analog.

Datenstruktur 2.18 (`edges_length`)
Die Längen aller Kanten werden in vielen Berechnungen benötigt. Daher werden diese einmalig berechnet und in `edges_length` abgespeichert. Diese Datenstruktur hat liefert einen Eintrag je Kante in `edges_with_triangles`.

Programm 2.4: edge_length_in_triangle.m

```
1  function [edges_length] = edge_length_in_triangle(coordinates, ...
2              edges_with_triangles)
3
4      edges_length = zeros(size(edges_with_triangles,1),1);
5
6      cnt_edges = find(edges_with_triangles(:,1) == 0, 1) - 1;
7      if isempty(cnt_edges)
8          cnt_edges = size(edges_with_triangles, 1);
9      end
10
11     for i = 1:cnt_edges
12         edges_length(i) = ...
13           length_edge(coordinates(edges_with_triangles(i,1),:),...
14                       coordinates(edges_with_triangles(i,2),:));
15     end
16
17 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
4	Initialisiere den entsprechenden Speicherplatz.
6-9	Bestimme die Anzahl an Kanten - `edges_with_triangles` ist gegebenenfalls schon verlängert und hat Nullzeilen.
ab 11	Durchlaufe alle Kanten.
12-14	Bestimme die Länge jeder Kante `i` und speichere sie zum entsprechenden Index.

2.2.2 Hilfsfunktion

Nachdem nun alle nötigen Datenstrukturen aufgebaut sind, benötigen wir nur noch eine Hilfsfunktion zur Implementierung des *edge-flip*.

Hilfsfunktion 2.19 (`inside_circumcircle`)
Wir benötigen eine Funktion, die überprüft, ob eine Kante AB lokal Delaunay ist. Die ersten beiden Kriterien aus Definition 2.3 - die Kante gehört zur konvexen Hülle oder zur Nebenbedingung - können wir durch einfache Abfragen von Variablenwerten überprüfen. Dementsprechend müssen wir nur noch entscheiden können, ob der Kreis um AB leer ist.
Zu einer Kante AB, die zu den Dreiecken ABC und BAD gehört, sind nur die Punkte C und D sichtbar. Daher genügt es, zu überprüfen, ob D im Umkreis von ABC liegt oder C im Umkreis von BAD. Dies übernimmt die Funktion `inside_circumcircle`.

Im Folgenden überprüfen wir, ob ein Punkt D im Umkreis eines Dreiecks ABC liegt. Hierzu

werden alle Punkte um eine Dimension erweitert:

$$P = \begin{pmatrix} x \\ y \end{pmatrix} \qquad \text{wird zu} \qquad P' = \begin{pmatrix} x \\ y \\ x^2 + y^2 \end{pmatrix}.$$

Durch diese Erweiterung können alle Punkte aus der x, y-Ebene als Punkte auf einem Paraboloiden betrachtet werden.
Auf diesem Paraboloiden spannen die Eckpunkte des Dreiecks ABC eine Ebene auf, die denselben in einer Ellipse schneidet. Die Projektion dieser Ellipse auf die x, y-Ebene bildet den Umkreis des Dreiecks. Veranschaulicht wird dies in Abbildung 2.11.

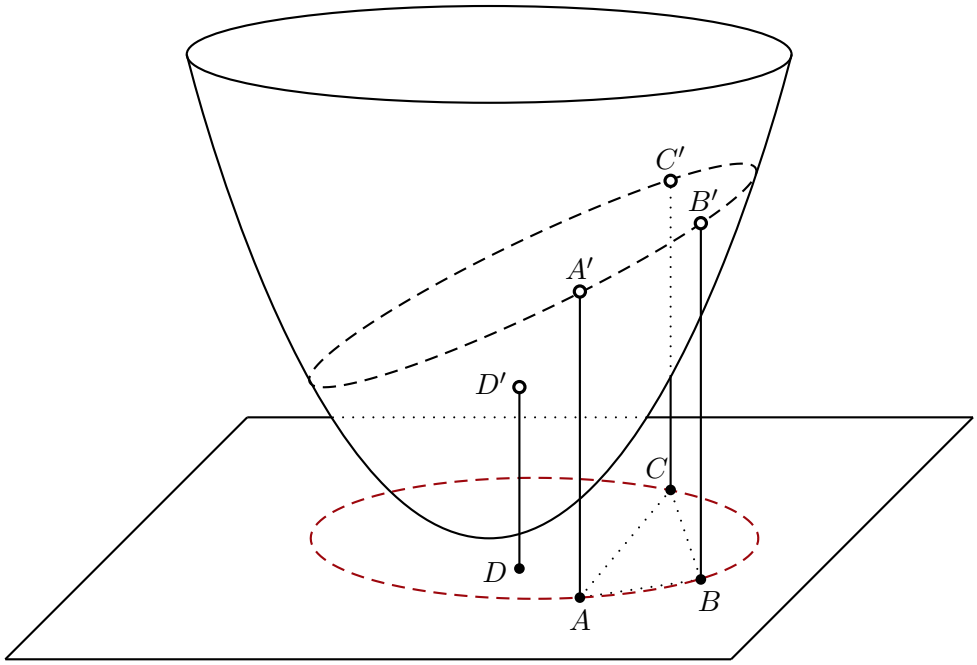


Abbildung 2.11: D liegt innerhalb des Umkreises von ABC

Nun gilt: Liegt D innerhalb des Umkreises von ABC, so liegt der um eine Dimension erweiterte Punkt D' unterhalb der von A', B' und C' aufgespannten Ebene. Ist D' Teil der Ebene, so liegt D auf dem Umkreis. Liegt D' oberhalb der Ebene, ist D nicht im Umkreis.
Sei also das Dreieck ABC (und damit auch das Dreieck $A'B'C'$) positiv orientiert. Dann liegt D' genau dann unterhalb der von A', B' und C' aufgespannten Ebende (und damit D im Umkreis von ABC), falls gilt

$$\det \begin{pmatrix} 1 & x_A & y_A & x_A^2 + y_A^2 \\ 1 & x_B & y_B & x_B^2 + y_B^2 \\ 1 & x_C & y_C & x_C^2 + y_C^2 \\ 1 & x_D & y_D & x_D^2 + y_D^2 \end{pmatrix} < 0.$$

Zurückgegeben wird das Negative der Determinante: Liegt der Punkt innerhalb des Umkreises, so gibt die Funktion 1 zurück, auf dem Umkreis 0, andernfalls -1. Dadurch passt sich der Rückgabewert den gebräuchlichen logischen Werten an.

Programm 2.5: inside_circumcircle.m

```
function [is_in_circle] = inside_circumcircle(a, b, c, d)

  delta = [
    1, a(1), a(2), a(1)^2+a(2)^2;
    1, b(1), b(2), b(1)^2+b(2)^2;
    1, c(1), c(2), c(1)^2+c(2)^2;
    1, d(1), d(2), d(1)^2+d(2)^2
    ];

  is_in_circle = - det(delta);

end
```

2.2.3 Realisierung *edge-flip*

Wir haben nun alle nötigen Datenstrukturen und sind in der Lage, die Umkreisbedingung zu überprüfen. Mit diesen Werkzeugen werden wir den *edge-flip* implementieren.

Zur Erinnerung: Eine Kante AB heißt *lokal Delaunay*, falls

(i) sie nur zu einem Dreieck ABC und damit zur konvexen Hülle oder der Nebenbedingung gehört oder

(ii) sie zu zwei Dreiecken ABC und BAD gehört und D nicht im Umkreis von ABC liegt.

Der Fall (i) ist schnell abgearbeitet. Durch die Nachbearbeitung in Abschnitt 1.2.3 gehören deratige Kanten nur zu einem Dreieck. Gehört eine Kante AB zu nur einem Dreieck ABC, so hat sie in `edges_with_triangles` die Form

`edges_with_triangles(i) = [index_A, index_B, index_ABC, Position, 0, 0].`

Liefert also `edges_with_triangles(i, 5)` den Wert 0, so ist diese Kante lokal Delaunay.

Wir betrachten nun den Fall (ii). Für das Überprüfen der Umkreisbedingung wird die Hilfsfunktion `inside_circumcircle` aufgerufen. Betrachte die Dreiecke ABC und BAD und damit die Kante AB. Es können dabei die folgenden Fälle auftreten.

- `inside_circumcircle(`A`, `B`, `C`, `D`) < 0` falls D nicht im Umkreis von ABC liegt - gilt auch `inside_circumcircle(`B`, `A`, `D`, `C`) < 0`, liegt also C nicht im Umkreis von BAD, so ist AB lokal Delaunay.

- `inside_circumcircle(`A`, `B`, `C`, `D`) = 0` falls das Viereck $ADBC$ entartet ist - AB betrachten wir als lokal Delaunay, da das Ändern der Kante wieder die gleiche Situation hervorruft.

- `inside_circumcircle(`A`, `B`, `C`, `D`) > 0` falls D im Umkreis von ABC liegt - AB ist nicht lokal Delaunay und wird deshalb durch CD ersetzt. CD ist lokal Delaunay. Es wird eine erneute Überprüfung der Kanten AD, DB, BC und CA nötig.

Das Flippen der Kante AB wird folglich nur dann durchgeführt, wenn die Umkreisbedingung überprüft wurde und für eines der beteiligten Dreiecke verletzt ist.

Es seien also im Folgenden die Dreiecke ABC und BAD so, dass für AB die Umkreisbedingung verletzt und das Viereck $ADBC$ nicht entartet ist.
Wir betrachten zunächst in Abbildung 2.12 die Ausgangsposition vor dem *edge-flip*. Die Kante AB verstößt gegen die Umkreisbedingung. Es wird durch `index_ABC` und `index_BAD` der entsprechende Index der Dreiecke in `triangulation` und `triangles_with_edges` angegeben. Die Kantenindizes `index_AB` etc. geben jeweils den Index in `edges_with_triangles`, `edges_length` und `edge_marker` sowie die Einträge aus `triangles_with_edges` an.

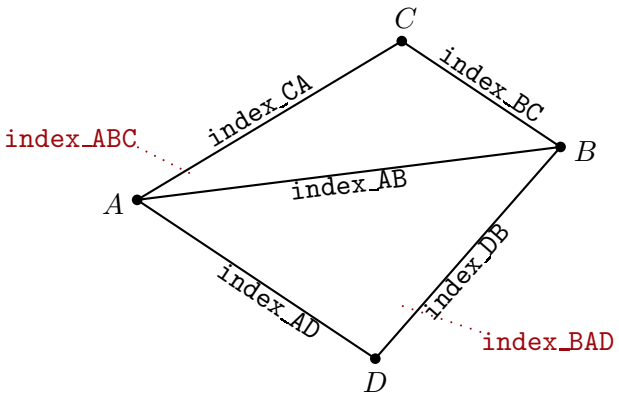


Abbildung 2.12: Indizierung vor *edge-flip*

Für die gespeicherten Daten bedeutet dies:

- Entferne die Markierung von AB, markiere AD, DB, BC und CA.
- ABC wird ersetzt durch DCA, BAD wird ersetzt durch CDB.
- Die Kanten AD und BC gehören damit nun zum jeweils anderen Dreieck, bei CA und DB muss nur die Position im Dreieck angepasst werden.
- Die Kante AB wird durch CD ersetzt.
- Die Kantenlängen werden entsprechend angepasst. Dabei muss nur die Länge der Kante CD neu berechnet werden.

Die Variablen werden mit Einträgen gemäß Abbildung 2.13 belegt.

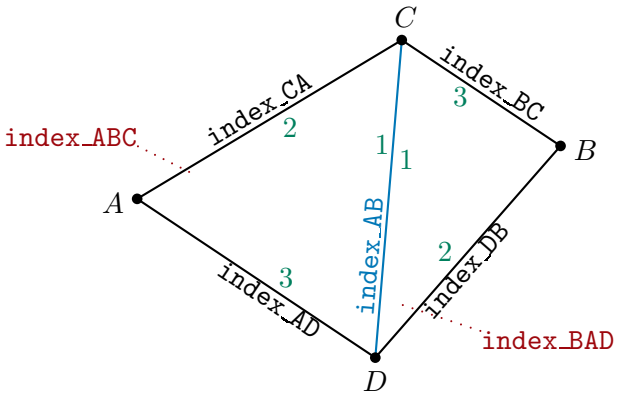


Abbildung 2.13: Indizierung nach *edge-flip*

Dabei übernimmt die Kante CD offensichtlich die Speicherposition von AB. Die Reihenfolge der Punkte in den neuen Dreiecken wird als $D-C-A$ beziehungsweise $C-D-B$ festgelegt. Dadurch ergeben sich auch gleichzeitig die neuen Kantenpositionen. Die an den Kanten stehenden Ziffern geben dabei jeweils die Position der Kante an, die sowohl in `edges_with_triangles` als auch (indirekt durch Positionierung des Eintrags) in `triangles_with_edges` benötigt wird.

Wir betrachten im Folgenden die Implementierung von Pseudocode 2.11.

Programm 2.6: edge_flip.m

```
function [triangulation, edges_with_triangles, triangles_with_edges,...
    edges_length, edge_marker] ...
  = edge_flip(coordinates, triangulation, edges_with_triangles,...
    triangles_with_edges, edges_length, edge_marker, h_max)
  % flippt Kanten, bis alle lokal Delaunay sind

  tol = h_max*10^(-13);
  succ = [2 3 1];

  % finde die erste Kante, die überprüft werden muss
  not_delaunay = find(edge_marker,1);

  while not_delaunay

    if edges_with_triangles(not_delaunay(1), 5) == 0
      edge_marker(not_delaunay(1)) = 0;
    else
      % Kante AB gehört zu ABC und BAD
      index_A = edges_with_triangles(not_delaunay(1), 1);
      index_B = edges_with_triangles(not_delaunay(1), 2);

      % bestimme, welches Dreieck ABC bzw BAD ist sowie C und D
      if triangulation(edges_with_triangles(not_delaunay(1), 3), ...
          edges_with_triangles(not_delaunay(1), 4)) == index_A

        index_ABC = edges_with_triangles(not_delaunay(1), 3);
        index_BAD = edges_with_triangles(not_delaunay(1), 5);
        pos_A_in_ABC = edges_with_triangles(not_delaunay(1), 4);
      else
        index_ABC = edges_with_triangles(not_delaunay(1), 5);
        index_BAD = edges_with_triangles(not_delaunay(1), 3);
        pos_A_in_ABC = edges_with_triangles(not_delaunay(1), 6);
      end

      pos_B_in_ABC = succ(pos_A_in_ABC);
      pos_C_in_ABC = succ(pos_B_in_ABC);
      pos_D_in_BAD = succ(triangulation(index_BAD, :) == index_A);

      index_C = triangulation(index_ABC, pos_C_in_ABC);
      index_D = triangulation(index_BAD, pos_D_in_BAD);

      % ---------- überprüfe, ob Umkreisbedingung verletzt ----------
      if inside_circumcircle(coordinates(index_A,:),...
                             coordinates(index_B,:),...
                             coordinates(index_C,:),...
```

```
46                          coordinates(index_D,:)) > tol || ...
47          inside_circumcircle(coordinates(index_B,:),...
48                          coordinates(index_A,:),...
49                          coordinates(index_D,:),...
50                          coordinates(index_C,:)) > tol
51
52          % C liegt im Umkreis von BAD oder D im Umkreis von ABC
53
54          % finde entsprechende Kanten
55          index_AB = not_delaunay(1);
56          index_BC = triangles_with_edges(index_ABC, pos_B_in_ABC);
57          index_CA = triangles_with_edges(index_ABC, pos_C_in_ABC);
58          index_DA = triangles_with_edges(index_BAD, ...
59                                          succ(succ(pos_D_in_BAD)));
60          index_DB = triangles_with_edges(index_BAD, pos_D_in_BAD);
61
62          % --------------- edges_with_triangles anpassen ----------------
63          % AC
64          if edges_with_triangles(index_CA, 3) == index_ABC
65            edges_with_triangles(index_CA, 4) = 2;
66          else
67            edges_with_triangles(index_CA, 6) = 2;
68          end
69
70          % AD
71          if edges_with_triangles(index_DA, 3) == index_BAD
72            edges_with_triangles(index_DA, 3:4) = [index_ABC 3];
73          else
74            edges_with_triangles(index_DA, 5:6) = [index_ABC 3];
75          end
76
77          % DB
78          if edges_with_triangles(index_DB, 3) == index_BAD
79            edges_with_triangles(index_DB, 4) = 2;
80          else
81            edges_with_triangles(index_DB, 6) = 2;
82          end
83
84          % BC
85          if edges_with_triangles(index_BC, 3) == index_ABC
86            edges_with_triangles(index_BC, 3:4) = [index_BAD 3];
87          else
88            edges_with_triangles(index_BC, 5:6) = [index_BAD 3];
89          end
90
91          % ersetze AB durch CD
92          edges_with_triangles(index_AB, :) = [
93            max(index_C, index_D), min(index_C, index_D),...
94              index_ABC, 1, index_BAD, 1 ];
95
96          % --------------------- Dreiecke anpassen ----------------------
97          triangulation(index_ABC, :) = [
98            index_D index_C index_A ];
99
```

```
100         triangles_with_edges(index_ABC, :) = [
101           index_AB index_CA index_DA ];
102
103         triangulation(index_BAD, :) = [
104           index_C index_D index_B ];
105
106         triangles_with_edges(index_BAD, :) = [
107           index_AB index_DB index_BC ];
108
109         % ------------------- Kantenlängen anpassen --------------------
110         edges_length(index_AB) = length_edge(coordinates(index_C,:),...
111                                              coordinates(index_D,:));
112
113         % ------------------- Kanten neu markieren ---------------------
114         edge_marker(index_AB) = 0;
115         edge_marker([index_CA index_BC index_DA index_DB]) = 1;
116
117       else
118         % Kante lokal Delaunay
119         edge_marker(not_delaunay(1)) = 0;
120       end
121     end
122
123     not_delaunay = find(edge_marker,1);
124   end
125
126 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
7	Definiere eine Fehlertoleranz um Rundungsfehler in den Zeilen 43-50 zu vermeiden.
11	Finde die erste markierte Kante.
ab 13	Betrete `while`-Schleife, solange eine Kante markiert ist.
15-16	Ist die Kante eine Kante der Nebenbedingung, gehört sie nur zu einem Dreieck. Sie ist nach Definition 2.3 lokal Delaunay - entferne die Markierung.
ab 17	Die Kante ist keine Nebenbedingungskante und muss überprüft werden.
19-20	A sei der Punkt mit dem größeren Index der ersten Kante aus `not_delaunay`, B der Punkt mit dem kleineren Index.
23-33	In einem der beiden Dreiecke ABC und BAD kommt die Kante AB vor, im anderen die Kante BA. Finde das Dreieck, in welchem die Punkte A und B in der Reihenfolge AB vorkommen - dies ist das Dreieck ABC. Speichere die Position von A im Dreieck ABC.
35-37	Bestimme in ABC die Positionen von B (Nachfolger von A) und C (Nachfolger von B) sowie in BAD die Position von D (Nachfolger von A in BAD).
43-50	Überprüfe, ob D im Umkreis von ABC liegt oder C im Umkreis von BAD. Tritt einer der Fälle ein, ist die Umkreisbedingung verletzt. Die Kante AB muss demnach durch die Kante BC ersetzt werden. Andernfalls geht es weiter in Zeile 117.

Zeilen	Funktionalität
55	Die Kante AB hat den Index `not_delaunay(1)`. Diese Wertzuweisung dient der besseren Übersicht im Quellcode.
56-60	Bestimme die Indizes der übrigen vier Kanten. Nutze dafür, dass diese in `triangles_with_edges` die gleiche Position haben wie ihr Anfangspunkt in `triangulation`.
64-89	Passe die Positionen der Kanten AC, DB, AD und BC in den Dreiecken an. Die Kanten AD und BC gehören nach dem *edge-flip* zum jeweils anderen Dreieck - passe hier auch den Dreiecksindex an. Die Indizierung erfolgt nach dem in Abbildung 2.13 dargestellten Schema.
92-94	Überschreibe die Kante AB durch die neue Kante CD.
97-107	Verändere sowohl in `triangulation` als auch in `triangles_with_edges` die Einträge so, dass ABC durch DCA, BAD durch CDB überschrieben wird.
110-111	Im Allgemeinen ist die Länge von CD ungleich der von AB - bestimme daher diese Kantenlänge neu. Die Längen der übrigen beteiligten Kanten ändern sich nicht.
114-115	Entferne die Markierung der Kante CD (angesprochen über den Index der alten Kante AB) und markiere die übrigen Kanten.
117-120	Hat die Kante AB in den Zeilen 48-55 beide Umkreisbedingungen erfüllt, ist sie lokal Delaunay. Entferne die Markierung.
123	Finde die nächste markierte Kante. Gibt es keine mehr, wird in Zeile 15 die `while`-Schleife abgebrochen.

Beispiel 2.20 Durch Ausführen des *edge-flip* ergibt sich die in der folgenden Abbildung 2.14 dargestellte Änderung der Triangulierung aus Beispiel 2.17.

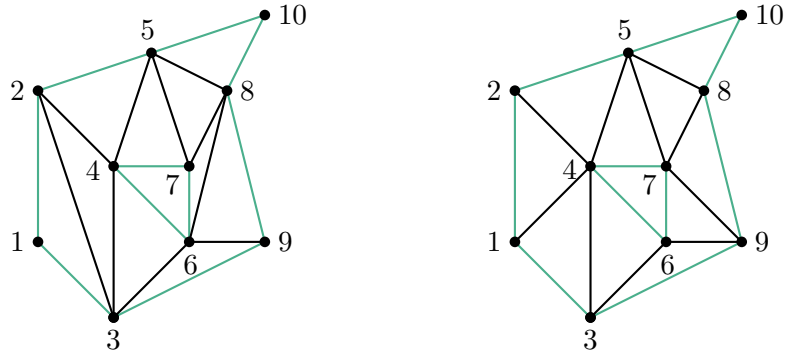


Abbildung 2.14: Triangulierung vor und nach *edge-flip*

Wir vergleichen die Belegungen der Variablen `triangulation`, `edges_with_triangles` und `triangles_with_edges` jeweils vor und nach dem *edge-flip*.

Wir beginnen mit der Triangulierung selbst. Die linke Ausgabe der Variablen ist die vor dem *edge-flip* und gehört zum linken Bild von Abbildung 2.14. Die rechte Ausgabe der Variablen zeigt die Situation nach dem *edge-flip*. Diese Triangulierung zeigt auch das rechte Bild der gleichen Abbildung. Die veränderten Einträge sind in den Variablen farbig hervorgehoben.

```
triangulation = [ 1  3  2
                  3  4  2
                  4  5  2
                  3  6  4
                  4  7  5
                  6  8  7
                  7  8  5
                  3  9  6
                  6  9  8
                  8 10  5 ]
```

```
triangulation = [ 4  1  3
                  1  4  2
                  4  5  2
                  3  6  4
                  4  7  5
                  9  7  6
                  7  8  5
                  3  9  6
                  7  9  8
                  8 10  5 ]
```

Aus Abbildung 2.14 erkennen wir, dass in der neuen Triangulierung die Kante `4-1` die alte Kante `3-2` und die Kante `9-7` die alte Kante `8-6` ersetzt. Dies spiegelt sich auch in der Belegung der Variablen `edges_with_triangles` wieder.

Aus Platzgründen ist der Variablenname durch `ewt` abgekürzt. Auch hier sind die Änderungen hervorgehoben: Die roten Zeilen geben die ausgetauschten Kanten an. In blau markierten Kanten wurde sich die Position der Kante im Dreieck neu bestimmt. Diese kann vor und nach dem *edge-flip* gleich sein. Die grün markierten Kanten haben beim *edge-flip* das zugehörige Dreieck gewecheselt.

Dabei wird auch ersichtlich, dass für jede geflippte Kante zwei andere Kanten einem anderen Dreieck zugewiesen werden. Außerdem ändern sich im Allgemeinen je geflippter Kante in zwei anderen Kanten die Positionen im Dreieck.

```
ewt = [ 2  1  1  3  0  0
        3  1  1  1  0  0
        3  2  1  2  2  3
        4  2  2  2  3  3
        5  2  3  2  0  0
        4  3  2  1  4  3
        6  3  4  1  8  3
        9  3  8  1  0  0
        5  4  3  1  5  3
        6  4  4  2  0  0
        7  4  5  1  0  0
        7  5  5  2  7  3
        8  5  7  2 10  3
       10  5 10  2  0  0
        7  6  6  3  0  0
        8  6  6  1  9  3
        9  6  8  2  9  1
        8  7  6  2  7  1
        9  8  9  2  0  0
       10  8 10  1  0  0 ]
```

```
ewt = [ 2  1  2  3  0  0
        3  1  1  2  0  0
        4  1  1  1  2  1
        4  2  2  2  3  3
        5  2  3  2  0  0
        4  3  1  3  4  3
        6  3  4  1  8  3
        9  3  8  1  0  0
        5  4  3  1  5  3
        6  4  4  2  0  0
        7  4  5  1  0  0
        7  5  5  2  7  3
        8  5  7  2 10  3
       10  5 10  2  0  0
        7  6  6  2  0  0
        9  7  9  1  6  1
        9  6  8  2  6  3
        8  7  9  3  7  1
        9  8  9  2  0  0
       10  8 10  1  0  0 ]
```

Zuletzt betrachten wir die Belegungen der Variablen `triangles_with_edges` vor und nach dem *edge-flip*. Die veränderten Dreiecke sind wieder farbig hervorgehoben. Sinnvollerweise ändert sich hier - ebenso wie in `triangulation` - stets eine gerade Anzahl an Einträgen.

```
triangles_with_edges = [ 2  3  1
                         6  4  3
                         9  5  4
                         7 10  6
                        11 12  9
                        16 18 15
                        18 13 12
                         8 17  7
                        17 19 16
                        20 14 13 ]
```

```
triangles_with_edges = [ 3  2  6
                         3  4  1
                         9  5  4
                         7 10  6
                        11 12  9
                        16 15 17
                        18 13 12
                         8 17  7
                        16 19 18
                        20 14 13 ]
```

3 Delaunay-Verfeinerung

In diesem Kapitel werden wir die nach *planesweep* und *edge-flip* bestehende Triangulierung weiter optimieren. Da im Allgemeinen die Punkte auf dem Rand des Gebiets relativ dicht sitzen, im Innern aber keine oder nur wenig Punkte vorhanden sind, sind die meisten Dreiecke sehr spitzwinklig. Dies werden wir durch das Einfügen zusätzlicher Punkte mit dem *refinement*-Algorithmus beheben.

Als Quellen wurden [Ed2001] Kapitel 2, [MG2005] Kapitel 5 sowie [Sh1997] Kapitel 3 verwendet.

3.1 Theoretische Grundlagen

Unser Ziel ist es, eine möglichst *optimale* Triangulierung zu erhalten. Unsere Kriterien für diese Optimalität sind die Folgenden.

(i) Keine Kante soll länger sein als $h_{\max}$.

(ii) Kein Winkel soll kleiner sein als $\alpha_{\min}$. Eine Ausnahme bilden hier Winkel, die sich auf Grund der Nebenbedingung nicht vergrößern lassen.

Wir benötigen also einen Algorithmus, mit dem wir

(i) zu lange Kanten verkürzen und

(ii) zu spitze Winkel nach Möglichkeit *aufsperren*, also vergrößern, können.

Das Verkürzen zu langer Kanten durch Einfügen neuer Punkte ist offensichtlich kein Problem.

Um zu kleine Winkel vergrößern zu können, müssen wir zunächst in jedem Dreieck den kleinsten Winkel bestimmen. Dabei benutzen wir die Tatsache, dass der kleinste Winkel in einem Dreieck gegenüber der kürzesten Seite liegt. Wir können davon ausgehen, dass der kleinste Winkel immer echt kleiner als $\frac{\pi}{2}$ ist.

Die Berechnung des Winkels erfolgt mit Hilfe des folgenden Satzes.

Satz 3.1 (Cosinussatz) *Gegeben sei das Dreieck ABC mit den Bezeichnungen aus der folgenden Abbildung 3.1.*

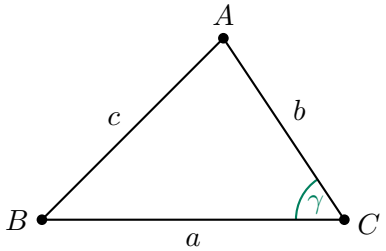


Abbildung 3.1: Bezeichnungen für den Cosinussatz

Dabei seien a, b und c die Längen der entsprechenden Kanten und $\gamma < \frac{\pi}{2}$. Dann gilt

$$c^2 = a^2 + b^2 - 2ab \cdot \cos\gamma$$
$$\text{und damit} \quad \cos\gamma = \frac{a^2 + b^2 - c^2}{2ab}.$$

Beweis: Zeichne in das Dreieck zusätzlich die Höhe ein. Die Bezeichnungen sind in Abbildung 3.2 gegeben.

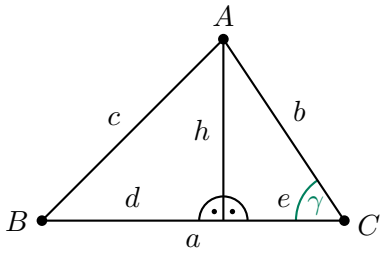


Abbildung 3.2: Bezeichnungen für den Beweis des Cosinussatz

Dabei bezeichnen auch d, e und h die Längen der entsprechenden Strecken. Der Cosinus des Winkels γ berechnet sich durch

$$\cos\gamma = \frac{e}{b}, \quad \text{was äquivalent ist zu} \quad e = b \cdot \cos\gamma.$$

Nun gilt

$$d^2 = (a - e)^2 = a^2 - 2ae + e^2.$$

Weiter gilt mit dem Satz von Pythagoras

$$h^2 = b^2 - e^2$$
$$\text{und damit} \quad c^2 = d^2 + h^2 = a^2 - 2ae + e^2 + b^2 - e^2$$
$$= a^2 + b^2 - 2ab \cdot \cos\gamma. \qquad \square$$

Um zusätzliche Rechenfehler zu vermeiden, werden wir im Folgenden stets mit dem Wert $\cos\gamma$ arbeiten, statt den Winkel explizit zu berechnen. Im für uns interessanten Intervall der Winkel zwischen 0 und $\frac{\pi}{2}$ ist der Cosinus positiv und streng monoton fallend. Ist $\alpha_{\min}$ der gewünschte Minimalwinkel, so ist das Kriterium für einen zu spitzen Winkel statt $\gamma < \alpha_{\min}$ nun

$$\cos\gamma > \cos\alpha_{\min}.$$

Nachdem wir zu spitze Winkel so identifizieren können, benötigen wir einen Mechanismus um sie zu vergrößern.

Idee 3.2 Der Umkreismittelpunkt M eines spitzen Dreiecks ABC liegt oft außerhalb desselben. Wird der Umkreismittelpunkt als neuer Punkt in die Triangulierung einfügt, so ist der Umkreis von ABC offensichtlich nicht mehr leer. Der *edge-flip* wird diese Kante dementsprechend verändern.

Beispiel 3.3 (Einfügen des Umkreismittelpunkts) Gegeben sei das in Abbildung 3.3 gezeigte Dreieck ABC. Der markierte Winkel α sei kleiner als $\alpha_{\min}$. Um den Winkel aufzusperren, fügen wir den Umkreismittelpunkt M als neuen Punkt ein und führen anschließend einen *edge-flip* durch.
Nach dem Satz 2.10 über die Winkelmaximierung ist in den beiden entstehenden Dreiecken der kleinste Winkel größer als α - im Idealfall sogar größer als $\alpha_{\min}$.

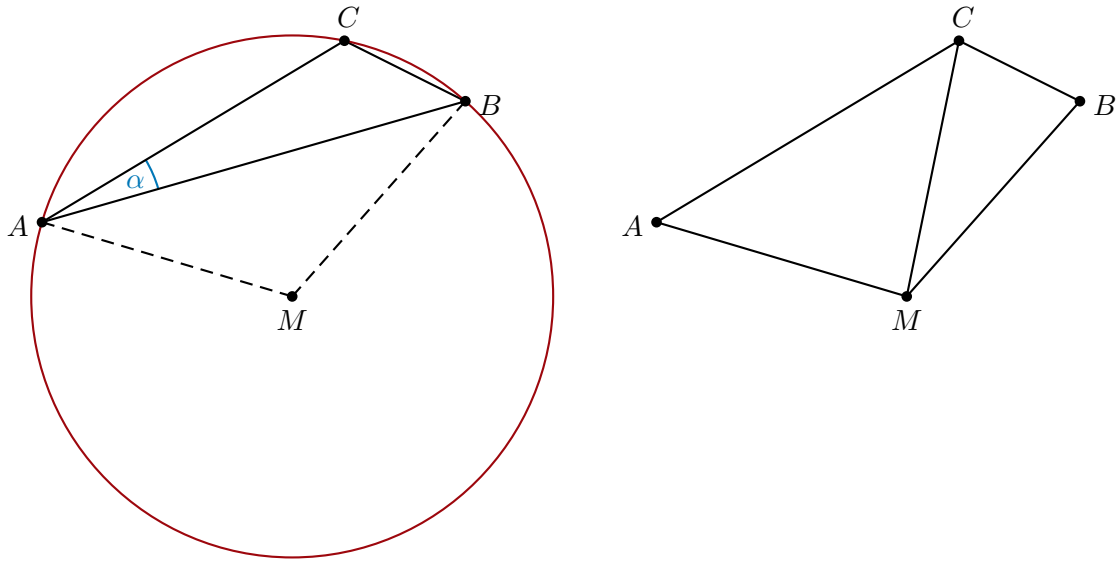


Abbildung 3.3: Einfügen des Umkreismittelpunkts zum Aufsperren des kleinsten Winkels

Das Einfügen des Umkreismittelpunkts muss nicht unbedingt zum Erfolg führen.

Problem 3.4 Ist der größte Winkel des Dreiecks kleiner als $\frac{\pi}{2}$, so liegt der Umkreismittelpunkt innerhalb des Dreiecks. Wird dieser Punkt eingefügt, werden alle Winkel des Dreiecks verkleinert.

Problem 3.5 Der Umkreismittelpunkt kann jenseits einer Kante der Nebenbedingung liegen. Der *edge-flip* kann dadurch den zu kleinen Winkel nicht aufsperren.

Wir benötigen in beiden Fällen eine alternative Vorgehensweise.

Idee 3.6 Halbiere wie in Abbildung 3.4 die längste Kante des Dreiecks, statt den Umkreismittelpunkt einzufügen.

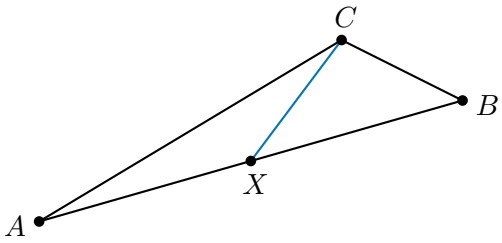


Abbildung 3.4: Halbieren der längsten Kante in spitzem Dreieck

Durch dieses Vorgehen können die beiden eben festgestellten Probleme umgangen werden. Der zu spitze Winkel α wird im nächsten Schritt vergrößert werden.

Weiter stellen wir fest: Kann durch das Einfügen des Umkreismittelpunktes M von ABC der kleinste Winkel aufgesperrt werden, so liegt M offensichtlich nicht im Innern von ABC. Statt dessen gibt es ein Dreieck PQR, in dessen Innern M liegt. Dabei kann der Punkt M sehr nah an einem der Eckpunkte oder an einer der Kanten von PQR liegen. Wird der Punkt also einfach eingefügt, so kann dadurch ein sehr spitzes Dreieck entstehen.

Beispiel 3.7 (Problem beim Einfügen des Umkreismittelpunkts) Wir betrachten die beiden in der folgenden Abbildung 3.5 dargestellten Dreiecke ABC und BAD. Der markierte Winkel α ist zu klein und soll durch Einfügen des Umkreismittelpunkts M von ABC aufgesperrt werden. Offensichtlich ist der dabei entstehende kleinste Winkel im Dreieck ADM deutlich kleiner als α.

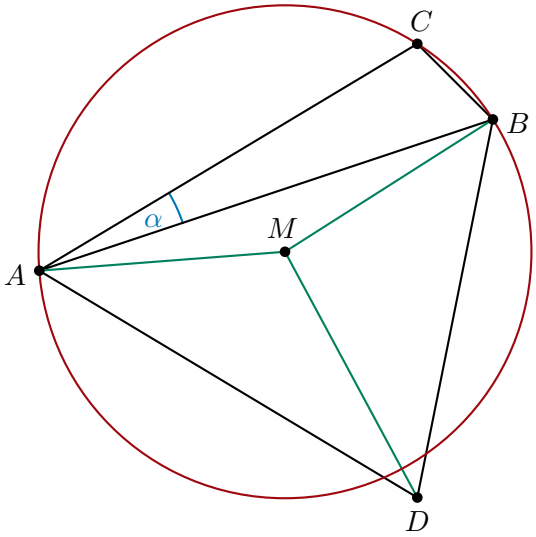


Abbildung 3.5: Entstehen eines neuen spitzen Dreiecks beim Einfügen von M

Statt nun also den Umkreismittelpunkt M von ABC einzufügen, nehmen wir als neuen Punkt den Umkreismittelpunkt M' von BAD, also des Dreiecks welches M beinhaltet. Da dieser Punkt per Definition von allen Ecken des Dreiecks BAD gleich weit entfernt ist, kann der Punkt nicht unverhältnismäßig nah an einer der Ecken liegen. Dies wird in Abbildung 3.6 dargestellt.

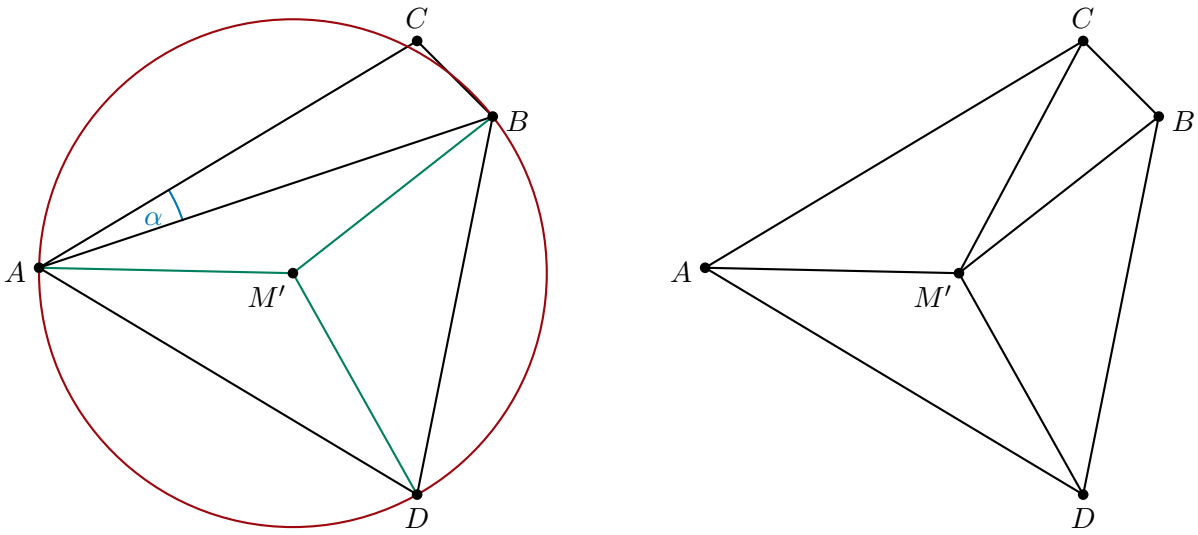


Abbildung 3.6: alternatives Einfügen von M' vor und nach dem *edge-flip*

Wir werden also analog zu diesem Beispiel wie folgt vorgehen: Liegt der Umkreismittelpunkt M von ABC in einem Dreieck PQR, so fügen wir statt M den Umkreismittelpunkt M' von PQR als neuen Punkt in die Triangulierung ein.
Dabei fügen wir sinnvollerweise den Umkreismittelpunkt von PQR nur dann ein, wenn er bezüglich der Nebenbedingungskanten von ABC aus sichtbar ist.

Beispiel 3.8 Ein Fall, in dem dies nicht gilt, ist in der folgenden Abbildung 3.7 dargestellt. Dabei gehören die grünen Kanten zur Nebenbedingung. Der Umkreismittelpunkt des grauen Dreiecks ist rot markiert.

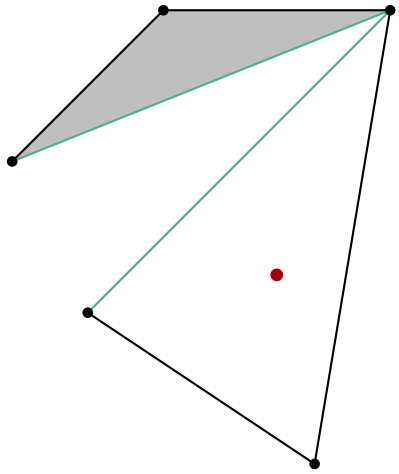


Abbildung 3.7: Umkreismittelpunkt von ABC nicht sichtbar

Diesen Fall werden wir so behandeln, als gäbe es für den Umkreismittelpunkt M von ABC kein umgebendes Dreieck.

Es sei also im Folgenden der Umkreismittelpunkt von ABC stets sichtbar bezüglich der Nebenbedingung zu allen Punkten aus dem Inneren von ABC.

Für die Lage von M' unterscheiden wir zwei Fälle.

(i) M' liegt nicht in PQR.
Wir fügen M' nicht als neuen Punkt in die Triangulierung ein. Statt dessen halbieren wir die längste Kante von PQR. Andernfalls kann es passieren, dass wir uns sukzessive so weit vom Anfangsdreieck ABC weg bewegen, dass das Einfügen eines neuen Punktes keine Auswirkung mehr auf ABC hat.

(ii) M' liegt in PQR.
Wir können demnach M' als neuen Punkt in die Triangulierung einfügen, um den kleinsten Winkel des ursprünglichen Dreiecks ABC aufzusperren. Liegt M' allerdings sehr nah an einer Kante von PQR, so entsteht durch Einfügen von M' ein Dreieck mit voraussichtlich noch spitzerem Winkel.

Wir müssen dem Fall (ii) noch einmal besondere Aufmerksamkeit schenken.

Beispiel 3.9 In der folgenden Abbildung 3.8 liegt der Umkreismittelpunkt M' sehr nah an der Kante PQ. Offensichtlich ist das Dreieck PQM' sehr spitzwinklig.

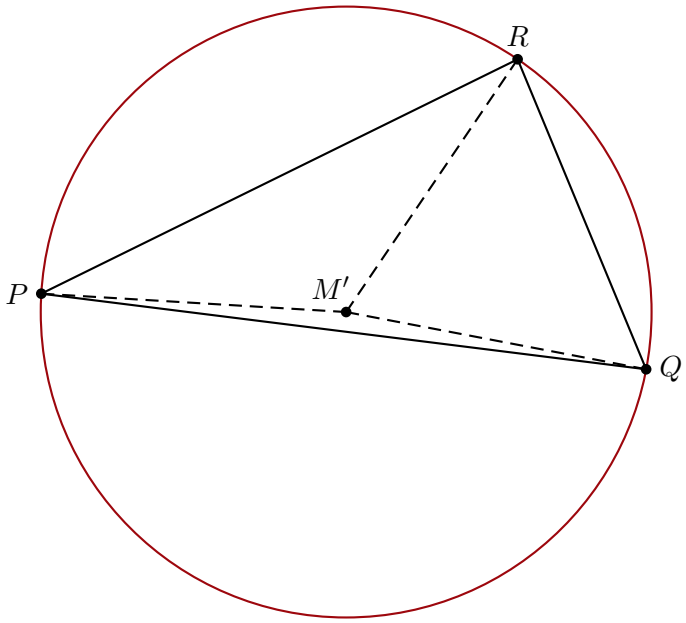


Abbildung 3.8: Umkreismittelpunkt sehr nah an einer Kante

Liegt die Kante, die dem Umkreismittelpunkt am nächsten ist, im Innern des Dreiecks, können wir den Umkreismittelpunkt als neuen Punkt einfügen. Ist diese Kante dagegen eine Kante der Nebenbedingung, so können wir den zu kleinen entstandenen Winkel offensichtlich nicht mehr durch den *edge-flip* aufsperren. Wir halbieren statt dessen in diesem Fall die entsprechende Kante.
Um einen derartigen Fall zu charakterisieren, verwenden wir den folgenden Spezialfall des Umkreiswinkelsatzes 2.7. Veranschaulicht wird dieser in Abbildung 3.9.

Satz 3.10 (Satz des Thales) *Alle Winkel im Halbkreisbogen sind rechte Winkel.*

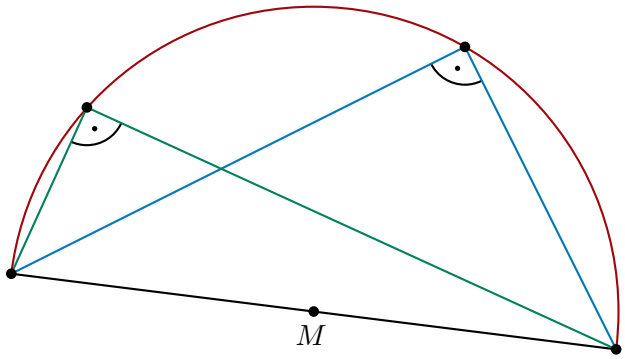


Abbildung 3.9: rechtwinklige Dreiecke im Halbkreisbogen um M

Wir können also folgern: Hat der größte Winkel eins Dreiecks PQR nahezu die Größe $\frac{\pi}{2}$, so liegt auch der Umkreismittelpunkt von PQR sehr nah an der längsten Kante. Ist die längste Kante von PQR eine Kante der Nebenbedingung, so erzeugen wir damit ein sehr spitzes Dreieck.

Mit dieser Überlegung passen wir unser Vorgehen ein weiteres Mal an: Liegt der Umkreismittelpunkt M eines Dreiecks ABC wie in Beispiel 3.7 in einem anderen Dreieck PQR, so bestimmen wir mit Hilfe des Cosinussatzes 3.1 einen Innenwinkel des Dreiecks PQR. Da wir überprüfen wollen, ob ein Winkel etwa die Größe $\frac{\pi}{2}$ hat, betrachten wir hierbei den größten Winkel φ - dieser liegt gegenüber der längsten Kante. Es können die folgenden Fälle auftreten.

(i) φ ist deutlich kleiner als $\frac{\pi}{2}$. Ein Beispiel hierfür finden wir im Dreieck BAD aus Abbildung 3.6. In diesem Fall liegt der Umkreismittelpunkt M' von PQR innerhalb des Dreiecks selbst und nicht zu nah an einer Kante. Wir fügen daher M' als neuen Punkt ein.

(ii) φ liegt zwischen $\frac{\pi}{2} - \varepsilon$ und $\frac{\pi}{2}$ mit einer Toleranz ε. Dies wird etwa in Abbildung 3.8 illustriert. Der Umkreismittelpunkt M' liegt zwar innerhalb des Dreiecks, aber sehr nah an der längsten Kante von PQR. Ist diese Kante eine Nebenbedingungskante, werden wir sie also wie bereits in Beispiel 3.9 beschrieben halbieren.

(iii) φ ist größer als $\frac{\pi}{2}$. Dies wird beispielsweise in Abbildung 3.3 gezeigt. Der Umkreismittelpunkt M' liegt außerhalb von PQR. Für diesen Fall hatten wir bereits beschlossen, die längste Kante zu halbieren.

Die Fälle (ii) und (iii) lassen sich in ihrer Behandlung offensichtlich zusammenfassen. Wir werden also das im Folgenden beschriebene Vorgehen anwenden, um einen Winkel im Dreieck ABC zu vergrößern.

Pseudocode 3.11 (Aufsperren zu kleiner Winkel)

---

bestimme den Umkreismittelpunkt M von ABC
if M liegt in einem von ABC verschiedenen Dreieck PQR **then**
  if größter Winkel von PQR ist größer als $\frac{\pi}{2} - \varepsilon$ **then**
    halbiere die längste Kante von PQR
  else
    füge den Umkreismittelpunkt M' von PQR als neuen Punkt ein
  end if
else
  halbiere die längste Kante von ABC
end if

Bevor wir die im Algorithmus verwendete Größe ε bestimmen, benötigen wir eine kurze Bemerkung zur Notation.

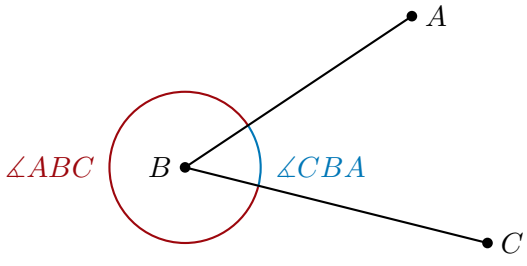


Abbildung 3.10: Winkelbezeichnung

Der Winkel in einem Punkt wird stets in der Form ∡ *Endpunkt - Scheitelpunkt - Endpunkt* bezeichnet. Dabei umschließen die Endpunkte den Scheitelpunkt gegen den Uhrzeigersinn.

Mit dieser Bezeichnung betrachten wir nun das folgende Beispiel.

Beispiel 3.12 Dazu betrachten wir erneut das Dreieck aus Beispiel 3.9. Wir ergänzen dabei die Kante PQ durch einen Punkt R', der auf der Kante PR liegt, zu einem rechtwinkligen Dreieck PQR'. Außerdem sei M' der Umkreismittelpunkt von PQR - er ergänzt QR zu einem gleichschenkligen Dreieck.

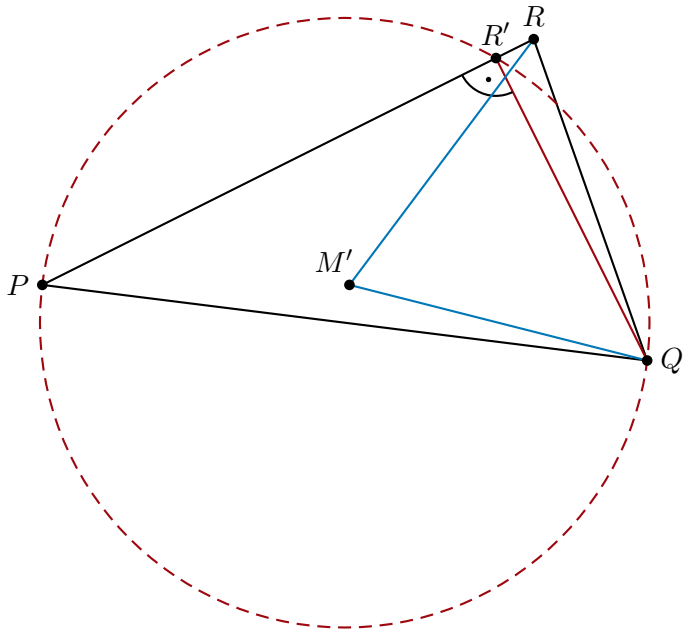


Abbildung 3.11: Abweichung vom rechten Winkel

In Abbildung 3.11 ist anschaulich klar, dass die beiden Winkel $\measuredangle RQR'$ und $\measuredangle M'QP$ gleich groß. Außerdem ist dies der Winkel, der den Winkel $\measuredangle R'RQ$ zu $\frac{\pi}{2}$ ergänzt.

Für unsere Wahl von ε ergeben sich die folgenden beiden Möglichkeiten.

(i) Ist PQ eine Randkante, so kann ein zu klein eingefügter Winkel nicht durch den *edge-flip* aufgesperrt werden. Wir fügen also den Umkreismittelpunkt M' von PQR nur dann ein, wenn der entstehende Winkel $\measuredangle M'QP$ mindestens die Größe $\alpha_{\min}$ hat. Dies ist nach Beispiel 3.12 genau dann der Fall, wenn der größte Winkel des Dreiecks höchstens die Größe $\frac{\pi}{2} - \alpha_{\min}$ hat. Andernfalls ist der entstehende Winkel zu klein und wir halbieren statt dessen die Kante PQ.

(ii) Ist PQ keine Randkante, so fügen wir den Umkreismittelpunkt M' ein, falls er innerhalb des Dreiecks liegt. Dies ist dann der Fall, wenn der größte Winkel kleiner als $\frac{\pi}{2}$ ist. Ist der größte Winkel gleich $\frac{\pi}{2}$, so liegt der Umkreismittelpunkt in der Mitte der längsten Kante. Da das Halbieren dieser Kante unsere Alternativhandlung ist, erlauben wir auch diesen Fall. Wir wählen also ε als 0.

Mit diesen Überlegungen konkretisieren wir noch einmal das Vorgehen zum Aufsperren zu kleiner Winkel.

Pseudocode 3.13 (Aufsperren zu kleiner Winkel, Version 2)

bestimme den Umkreismittelpunkt M von ABC
if M liegt in einem von ABC verschiedenen Dreieck PQR **then**
  if längste Kante von PQR Randkante **then**
    $\varepsilon = \alpha_{\min}$
  else
    $\varepsilon = 0$
  end if
  if größter Winkel von PQR ist größer als $\frac{\pi}{2} - \varepsilon$ **then**
    halbiere die längste Kante von PQR
  else
    füge den Umkreismittelpunkt M' von PQR als neuen Punkt ein
  end if
else
  halbiere die längste Kante von ABC
end if

3.2 Implementierung

Wir betrachten nun die Implementierung des sogenannten *refinement*-Algorithmus. Dieser soll sämtliche Überlegungen aus dem vorangegangen Theorieteil realisieren.

Wie bereits zu Beginn von Abschnitt 3.1 festgestellt, lässt sich der Algorithmus in zwei unterschiedliche Schritte aufteilen.

Verkürzen der Kanten

Im ersten Schritt werden alle zu langen Kanten so lange halbiert, bis die erste Bedingung erfüllt ist. Dabei wird immer die längste Kante geteilt, um eine möglichst regelmäßige Triangulierung zu ermöglichen.
Da jede Kante eine endliche Länge hat, muss nach endlich vielen Schritten jede Kante kurz genug sein.

Vergrößern der Winkel

Im nächsten Schritt sollen alle Winkel, die kleiner als $\alpha_{\min}$ sind, vergrößert werden. Auch hier ist es sinnvoll, eine bestimmte Reihenfolge einzuhalten: Es wird stets der kleinste Winkel vergrößert. Durch das Vergrößern von Winkeln entstehen keine Kanten die länger sind als bereits vorhandene. Daher ist eine wiederholte Ausführung des ersten Schrittes nicht nötig.

Es kann passieren, dass der von uns geforderte Minimalwinkel nicht erreicht werden kann. Wir speichern für diesen Fall die Triangulierung und die Koordinaten nach dem Verkürzen der Kanten. So ist meist bereits eine gute Triangulierung gegeben. In diesem Fall wollen wir außerdem die Größe des kleinsten vorkommenden Winkels ausgeben. So ist es dem Anwender überlassen, ob der bestehende kleinste Winkel groß genug ist.

Das Vorgehen des *refinement*-Algorithmus sieht wie folgt aus.

Pseudocode 3.14 (*refinement*)

---

while zu lange Kanten vorhanden **do**
    halbiere längste Kante
    führe *edge-flip* aus
end while
speichere die aktuelle Triangulierung
while zu kleiner Winkel vorhanden **do**
    vergrößere kleinsten Winkel
    führe *edge-flip* aus
end while

3.2.1 verwendete Hilfsfunktionen

Wie bereits in den letzten Kapiteln benötigen wir für die Implementierung des *refinement* wieder einige Hilfsfunktionen. Diese werden im kommenden Abschnitt gezeigt und erläutert.

Hilfsfunktion 3.15 (`split_encroached_edge`)
Wir nennen eine Kante *beeinträchtigt* - oder *encroached* - falls sie entweder zu lang ist oder beim Winkel Vergrößern ein Punkt eingefügt werden soll, der zu nah an der Kante liegt. In beiden Fällen soll die Kante durch Einfügen ihres Mittelpunktes halbiert werden. Dabei unterscheiden wir, ob die Kante zu einem oder zu zwei Dreiecken gehört.

Es sei zunächst die beeinträchtigte Kante AB eine Nebenbedingungskante und gehöre damit nur zu einem Dreieck ABC.

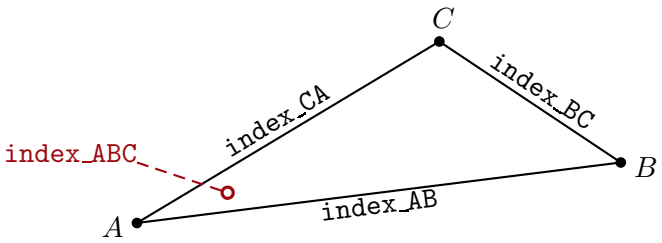


Abbildung 3.12: Ausgangssituation `split_encroached_edge` mit einem Dreieck

Wird die Kante AB halbiert, wird dabei offensichtlich eine neue Kante sowie ein neues Dreieck erzeugt. Es sei X der eingefügte Punkt.

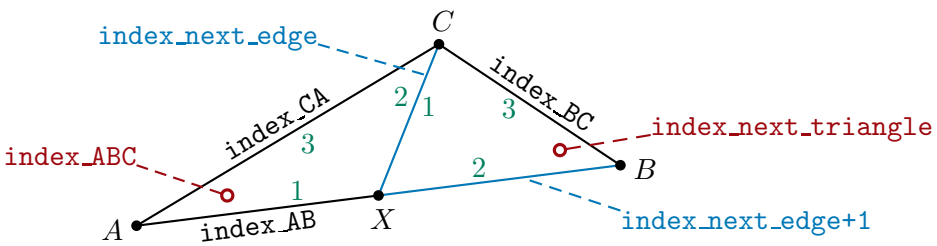


Abbildung 3.13: Indizierung `split_encroached_edge` mit einem Dreieck

In den beiden neuen Dreiecken wird die Reihenfolge der Punkte auf $A - X - C$ und $C - X - B$ festgelegt. Dabei ersetzt AXC das alte Dreieck ABC, das Dreieck CXB wird neu in die Triangulierung eingefügt.

Da die Kante AB eine Nebenbedingungskante war, gilt dies nun auch für die Kanten XA und XB. Da X als neuer Punkt den größten Index hat, steht er bei der Kantenbezeichnung stets vorn. Die Indizierung der Kanten und ihre Positionen in den entsprechenden Dreiecken sind in Abbildung 3.13 abzulesen.

Im Folgenden betrachten wir nun das Halbieren einer Kante AB, die zu zwei Dreiecken ABC und BAD gehört.

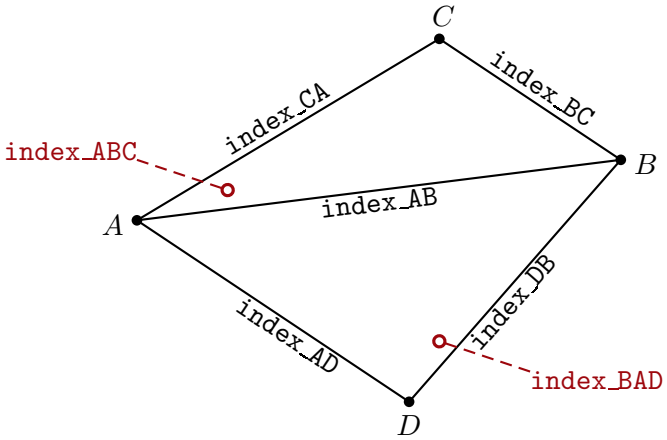


Abbildung 3.14: Ausgangssituation `split_encroached_edge` mit zwei Dreiecken

Halbieren wir in diesem Fall die Kante AB durch Einfügen ihres Mittelpunktes X, so entstehen zwei neue Dreiecke und drei neue Kanten. Auch hier legen wir die Punktreihenfolge in den Dreiecken stets so fest, dass der neu eingefügte Punkt X der zweite Punkt in jedem Dreieck ist. Es ergeben sich also die Dreiecke AXC, CXB, BXD und DXA.
Sämtliche verwendeten Indizes und Kantenpositionen sind in Abbildung 3.15 abzulesen.

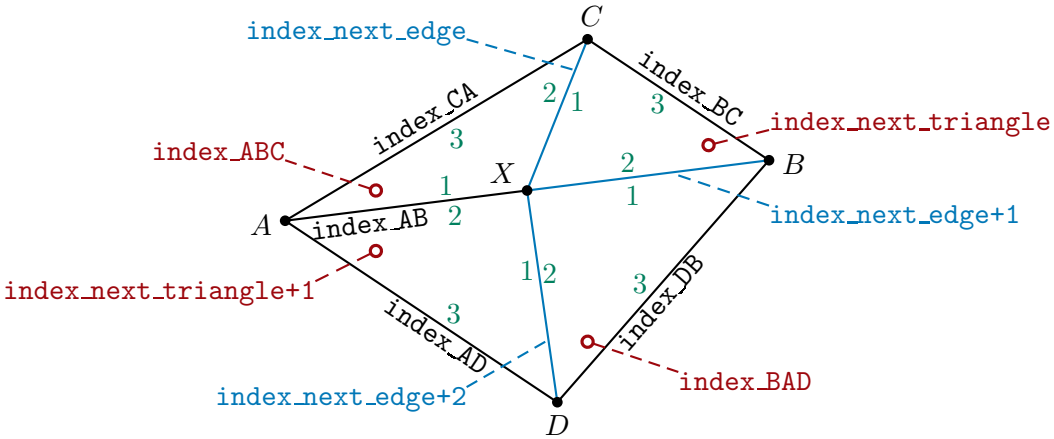


Abbildung 3.15: Indizierung `split_encroached_edge` mit zwei Dreiecken

Wir werden nun das Programm betrachten, das diesen Vorgang implementiert.

Programm 3.1: split_encroached_edge.m

```
function [coordinates, triangulation, edges_with_triangles,...
    triangles_with_edges, index_next_point, index_next_edge,...
    index_next_triangle, edge_marker, edges_length]...
  = split_encroached_edge(coordinates, triangulation, ...
    edges_with_triangles, triangles_with_edges, edge_marker,...
    edges_length, index_next_point, index_next_edge,...
    index_next_triangle, index_encroached_edge)

  % halbiert die beeinträchtigte Kante AB und passt alle
  % Variablen an

  succ = [2 3 1];

  index_ABC = edges_with_triangles(index_encroached_edge, 3);
  index_BAD = edges_with_triangles(index_encroached_edge, 5);

  % bestimme in ABC die Positionen der Punkte A, B und C
  index_A = find(triangulation(index_ABC, :) == ...
                 edges_with_triangles(index_encroached_edge, 1));

  if triangulation(index_ABC, succ(index_A)) == ...
                   edges_with_triangles(index_encroached_edge, 2)
    index_B = succ(index_A);
    index_C = succ(index_B);
  else
    index_B = index_A;
    index_C = succ(index_B);
    index_A = succ(index_C);
  end

  % füge neuen Punkt zu (Kantenmittelpunkt von AB)
  coordinates(index_next_point, :) = ...
      (coordinates(triangulation(index_ABC, index_B),:) + ...
       coordinates(triangulation(index_ABC, index_A),:)) ./ 2;

  if ~index_BAD % beeinträchtigte Kante ist nur Teil eines Dreiecks
    % teile das Dreieck ABC auf in AXC und CXB

    % --------------- aktualisiere edges_with_triangles ----------------
    % XC einfügen
    edges_with_triangles(index_next_edge, :) = [...
        index_next_point, triangulation(index_ABC, index_C), ...
        index_ABC, 2, index_next_triangle, 1 ];

    % XB einfügen (das ist auch eine Randkante)
    edges_with_triangles(index_next_edge+1, :) = [...
        index_next_point, triangulation(index_ABC, index_B), ...
        index_next_triangle, 2, 0, 0 ];

    % ersetze AB durch XA
    edges_with_triangles(index_encroached_edge, :) = [ ...
        index_next_point, triangulation(index_ABC, index_A), ...
        index_ABC, 1, 0, 0 ];
```

```
54
55      % Kantenindize von BC und CA bestimmen
56      index_BC = triangles_with_edges(index_ABC, index_B);
57      index_CA = triangles_with_edges(index_ABC, index_C);
58
59      % BC aktualisieren
60      if edges_with_triangles(index_BC, 3) == index_ABC
61        edges_with_triangles(index_BC, 3:4) = [index_next_triangle, 3];
62      else
63        edges_with_triangles(index_BC, 5:6) = [index_next_triangle, 3];
64      end
65
66      % CA aktualisieren
67      if edges_with_triangles(index_CA, 3) == index_ABC
68        edges_with_triangles(index_CA, 4) = 3;
69      else
70        edges_with_triangles(index_CA, 6) = 3;
71      end
72
73      % ----------------------- Kanten markieren  ------------------------
74      edge_marker([index_next_edge, index_BC, index_CA]) = 1;
75
76      % ------------------ bestimme neue Kantenlängen --------------------
77      % AB wird halbiert zu XA, XB hat die gleiche Länge
78      edges_length(index_encroached_edge) = ...
79                    edges_length(index_encroached_edge)/2;
80      edges_length(index_next_edge+1) = ...
81                    edges_length(index_encroached_edge);
82
83      % berechne die Länge von XC neu
84      edges_length(index_next_edge) = ...
85         length_edge(coordinates(index_next_point,:), ...
86                    coordinates(triangulation(index_ABC, index_C),:));
87
88      % -------------------- Dreiecke aktualisieren ----------------------
89      % CXB einfügen
90      triangulation(index_next_triangle, :) = [...
91            triangulation(index_ABC, index_C), ...
92            index_next_point, ...
93            triangulation(index_ABC, index_B) ];
94
95      triangles_with_edges(index_next_triangle, :) = [ ...
96            index_next_edge, ...
97            index_next_edge + 1, ...
98            index_BC ];
99
100     % ABC zu AXB ändern
101     triangulation(index_ABC, :) = [ ...
102           triangulation(index_ABC, index_A), ...
103           index_next_point, ...
104           triangulation(index_ABC, index_C) ];
105
106     triangles_with_edges(index_ABC, :) = [ ...
107           index_encroached_edge, ...
```

```
108           index_next_edge, ...
109           index_CA ];
110
111     % ---------- zugefügt wurden zwei Kanten und ein Dreieck -----------
112     index_next_edge = index_next_edge + 2;
113     index_next_triangle = index_next_triangle + 1;
114
115   else % beeinträchtigte Kante liegt zwischen zwei Dreiecken
116
117     % -------------- bestimme in BAD die Position von D ----------------
118     index_D = succ(triangulation(index_BAD, :) == ...
119                    triangulation(index_ABC, index_A));
120
121     % -------- teile ABC und BAD auf in AXC, CXB, DXA und BXD ----------
122
123     % --------------- aktualisiere edges_with_triangles ----------------
124     % XC einfügen
125     edges_with_triangles(index_next_edge, :) = [...
126             index_next_point, triangulation(index_ABC, index_C), ...
127             index_ABC, 2, index_next_triangle, 1 ];
128
129     % XB einfügen
130     edges_with_triangles(index_next_edge+1, :) = [...
131             index_next_point, triangulation(index_ABC, index_B), ...
132             index_next_triangle, 2, index_BAD, 1 ];
133
134     % XD einfügen
135     edges_with_triangles(index_next_edge+2, :) = [ ...
136             index_next_point, triangulation(index_BAD, index_D), ...
137             index_BAD, 2, index_next_triangle+1, 1 ];
138
139     % ersetze AB durch XA
140     edges_with_triangles(index_encroached_edge, :) = [ ...
141             index_next_point, triangulation(index_ABC, index_A), ...
142             index_ABC, 1, index_next_triangle+1, 2 ];
143
144     % Kantenindize von BC, CA, AD und DB bestimmen
145     index_BC = triangles_with_edges(index_ABC, index_B);
146     index_CA = triangles_with_edges(index_ABC, index_C);
147     index_AD = triangles_with_edges(index_BAD, succ(succ(index_D)));
148     index_DB = triangles_with_edges(index_BAD, index_D);
149
150     % BC aktualisieren
151     if edges_with_triangles(index_BC, 3) == index_ABC
152       edges_with_triangles(index_BC, 3:4) = [index_next_triangle, 3];
153     else
154       edges_with_triangles(index_BC, 5:6) = [index_next_triangle, 3];
155     end
156
157     % CA aktualisieren
158     if edges_with_triangles(index_CA, 3) == index_ABC
159       edges_with_triangles(index_CA, 4) = 3;
160     else
161       edges_with_triangles(index_CA, 6) = 3;
```

```
162     end
163
164     % AD aktualisieren
165     if edges_with_triangles(index_AD, 3) == index_BAD
166       edges_with_triangles(index_AD, 3:4) = [index_next_triangle+1, 3];
167     else
168       edges_with_triangles(index_AD, 5:6) = [index_next_triangle+1, 3];
169     end
170
171     % DB aktualisieren
172     if edges_with_triangles(index_DB, 3) == index_BAD
173       edges_with_triangles(index_DB, 4) = 3;
174     else
175       edges_with_triangles(index_DB, 6) = 3;
176     end
177
178     % ------------- Kanten markieren, ggfs als Randkanten --------------
179     edge_marker([index_encroached_edge, index_next_edge,...
180         index_next_edge+1, index_next_edge+2, index_AD,...
181         index_DB, index_BC, index_CA])                      = 1;
182
183     % ------------------ bestimme neue Kantenlängen --------------------
184     % XA ist halb so lang wie AB
185     edges_length(index_encroached_edge) = ...
186                   edges_length(index_encroached_edge)/2;
187
188     %XC
189     edges_length(index_next_edge) = ...
190         length_edge(coordinates(index_next_point,:),...
191                     coordinates(triangulation(index_ABC, index_C),:));
192
193     %XB ist genauso lang wie XA
194     edges_length(index_next_edge+1) = ...
195                   edges_length(index_encroached_edge);
196
197     %XD
198     edges_length(index_next_edge+2) = ...
199         length_edge(coordinates(index_next_point,:),...
200                     coordinates(triangulation(index_BAD, index_D),:));
201
202     % -------------------- Dreiecke aktualisieren ----------------------
203     % CXB einfügen
204     triangulation(index_next_triangle, :) = [...
205           triangulation(index_ABC, index_C), ...
206           index_next_point, ...
207           triangulation(index_ABC, index_B) ];
208
209     triangles_with_edges(index_next_triangle, :) = [ ...
210           index_next_edge, ...
211           index_next_edge + 1, ...
212           index_BC ];
213
214     % DXA einfügen
215     triangulation(index_next_triangle+1, :) = [ ...
```

```
216           triangulation(index_BAD, index_D), ...
217           index_next_point, ...
218           triangulation(index_ABC, index_A) ];
219
220     triangles_with_edges(index_next_triangle+1, :) = [ ...
221           index_next_edge + 2, ...
222           index_encroached_edge, ...
223           index_AD ];
224
225     % BAD zu BXD ändern
226     triangulation(index_BAD, :) = [ ...
227           triangulation(index_ABC, index_B), ...
228           index_next_point, ...
229           triangulation(index_BAD, index_D) ];
230
231     triangles_with_edges(index_BAD, :) = [ ...
232           index_next_edge + 1, ...
233           index_next_edge + 2, ...
234           index_DB ];
235
236     % ABC zu AXB ändern
237     triangulation(index_ABC, :) = [ ...
238           triangulation(index_ABC, index_A), ...
239           index_next_point, ...
240           triangulation(index_ABC, index_C) ];
241
242     triangles_with_edges(index_ABC, :) = [ ...
243           index_encroached_edge, ...
244           index_next_edge, ...
245           index_CA ];
246
247     % ---------- zugefügt wurden zwei Kanten und ein Dreieck -----------
248     index_next_edge = index_next_edge + 3;
249     index_next_triangle = index_next_triangle + 2;
250   end
251   index_next_point = index_next_point + 1;
252
253 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
14-15	Bestimme die Indizes der beiden Dreiecke ABC und BAD. Ist die Kante nur Teil eines Dreiecks, so ist `index_BAD` gleich null.
18-29	Bestimme die Positionen von A, B und C in ABC.
32-34	Der neue Punkt ist der Mittelpunkt der Kante AB.
ab 36	Die Kante gehört nur zu einem Dreieck. Andernfalls weiter in Zeile 119.
41-48	Füge die Kanten XC und XB ein. Die Einträge werden nach Abbildung 3.13 gewählt.
51-53	Überschreibe die Kante AB durch XA.
56-57	Bestimme aus `triangles_with_edges` die Indizes der Kanten BC und CA.

Zeilen	Funktionalität
60-64	Die Kante BC gehört nun zu einem anderen Dreieck - aktualisiere Dreiecksindex und Position.
67-71	In der Kante CA muss lediglich die Position im Dreieck angepasst werden.
74	Markiere die Kanten CA, XC und BC, um sie erneut vom *edge-flip* überprüfen zu lassen. Die Kanten XA und XB sind nur Teil eines Dreiecks und damit per Definition lokal Delaunay. Sie müssen nicht überprüft werden.
78 - 81	Die Kante AB wird halbiert - anschließend sind die Kanten XA und XB jeweils halb so lang wie AB.
84-86	Die Länge der Kante XC muss neue berechnet werden.
90-98	Füge das Dreieck CXB neu ein.
101-109	Überschreibe das alte Dreieck ABC durch AXB.
112-113	Es wurden zwei Kanten und ein Dreieck zugefügt. Die entsprechenden Indizes müssen für das Einfügen des nächsten Punktes inkrementiert werden.
ab 115	Ab hier erfolgt die eben beschriebene Aufteilung der Kante AB für den Fall, dass AB Teil von zwei Dreiecken ist. Dabei gehen wir analog vor. Die Einträge der Variablen lassen sich mit Abbildung 3.15 erschließen.
	Drei Unterschiede seien dabei noch hervorgehoben.
118-119	Die Position des Punktes D aus dem Dreieck BAD muss noch bestimmt werden.
179-181	Da keine der beteiligten acht Kanten mehr eine Randkante sein muss, werden alle Kanten für einen erneuten Durchlauf des *edge-flip* markiert.
248-249	Es wurden eine Kante und ein Dreieck mehr eingefügt als im ersten Fall. Die entsprechenden Variablen müssen daher anders inkrementiert werden.
251	In beiden Fällen wurde jeweils ein Punkt neu eingefügt. Inkrementiere auch hier die entsprechende Variable.

Hilfsfunktion 3.16 (`get_circumcentre`)
Im Theorieteil dieses Kapitels haben wir festgestellt, dass wir durch Einfügen des Umkreismittelpunktes von ABC gegebenenfalls dessen kleinsten Winkel aufsperren können. Wir benötigen Daher eine Funktion, die zu gegebenen Eckpunkten A, B und C diesen Umkreismittelpunkt X berechnet.

Die Eckpunkte seien gegeben durch

$$A = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, \qquad B = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \quad \text{und} \quad C = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}.$$

Gesucht sind die Koordinaten eines Punktes $X = (x_1,\ x_2)^T$, der für einen nicht zu bestimmenden Radius r die folgenden Gleichungen erfüllt.

$$|A - X| = |B - X| = |C - X| = r$$

Wir betrachten stellvertretend die Gleichung

$$|A - X| = r$$

Damit ergibt sich

$$\begin{aligned} & (x_1 - a_1)^2 + (x_2 - a_2)^2 = r^2 \\ \Leftrightarrow \quad & x_1^2 - 2a_1x_1 + a_1^2 + x_2^2 - 2a_2x_2 + a_2^2 = r^2 \\ \Leftrightarrow \quad & a_1^2 + a_2^2 - a_1 \underbrace{v_1}_{=2x_1} - a_2 \underbrace{v_2}_{=2x_2} - \underbrace{v_3}_{=r^2} = 0 \end{aligned}$$

und analog auch für die anderen Gleichungen

$$\begin{aligned} b_1^2 + b_2^2 - b_1v_1 - b_2v_2 - v_3 = 0 \\ c_1^2 + c_2^2 - c_1v_1 - c_2v_2 - v_3 = 0 \end{aligned}$$

Insgesamt erhalten wir demnach das folgende lineare Gleichungssystem.

$$\begin{pmatrix} a_1 & a_2 & 1 \\ b_1 & b_2 & 1 \\ c_1 & c_2 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} a_1^2 + a_2^2 \\ b_1^2 + b_2^2 \\ c_1^2 + c_2^2 \end{pmatrix} \tag{3.1}$$

Hieraus wiederum ergibt sich für den gesuchten Punkt

$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} v_1/2 \\ v_2/2 \end{pmatrix}.$$

Programm 3.2: Bestimmen des Umkreismittelpunkts von ABC

```
1  function [circumcentre] = get_circumcentre(a,b,c)
2    % bestimmt den Umkreismittelpunkt des Dreiecks abc
3
4    % Aufstellen des LGS Av = y
5    A = [a(1) a(2) 1; ...
6       b(1) b(2) 1; ...
7       c(1) c(2) 1];
8    y = [a(1)^2 + a(2)^2; ...
9       b(1)^2 + b(2)^2; ...
10      c(1)^2 + c(2)^2];
11
12    x = A\y;
13
14    circumcentre = [x(1)/2, x(2)/2];
15 end
```

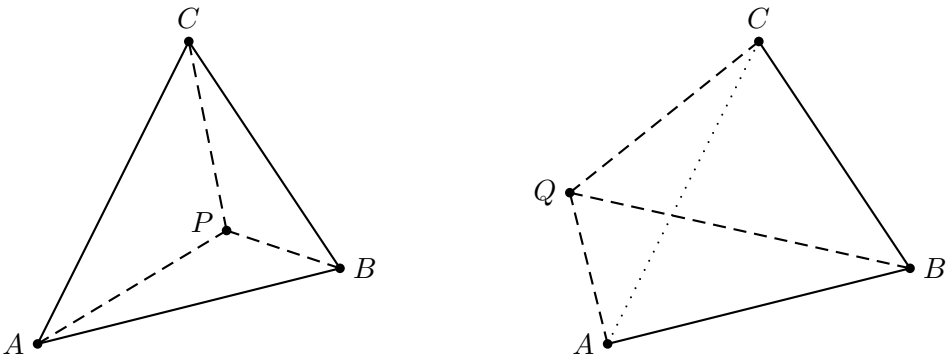
Erläuterung des Quellcodes

Zeilen	Funktionalität
5-10	Definieren der Matrix sowie der rechten Seite des linearen Gleichungssystems aus Gleichung 3.1.
12	Löse das LGS $Av = y$.
14	Bestimmen der Koordinaten des Umkreismittelpunkts aus der Lösung v.

Hilfsfunktion 3.17 (`find_encircling_triangle`)
Wird zum Aufsperren eines spitzen Winkels der Umkreismittelpunkt des zugehörigen Dreiecks eingefügt, so liegt dieser sinnvollerweise nicht im Dreieck selbst. Um den Punkt in die Triangulierung einzufügen, wird der Index des umgebenden Dreiecks benötigt.
Zu einem gegebenen Punkt $P = (p_1,\ p_2)^T$ soll also ein Dreieck ABC gefunden werden, so dass P im Innern von ABC oder auf einer der Kanten liegt.

Allgemein gilt: Ein Punkt P liegt genau dann in einem Dreieck ABC, wenn die Dreiecke PCA, PAB und PBC die gleiche Orientierung haben.

Beispiel 3.18 Im der folgenden Abbildung sind im linken Bild die Dreiecke PCA, PAB und PBC positiv orientiert - der Punkt P liegt im Dreieck ABC. Im rechten Bild sind die Dreiecke QAB und QBC positiv, das Dreieck QCA dagegen negativ orientiert. Der Punkt Q liegt nicht im Dreieck ABC.



Analog zu Vorgehen bei Hilfsfunktion 1.27 lässt sich die Orientierung wieder über das Berechnen einer Determinante bestimmen.

Um das Berechnen der drei benötigten Determinanten nicht für jedes Dreieck der Triangulierung ausführen zu müssen, schließt man am Anfang alle Dreiecke aus, bei denen

- die größte x-Koordinate eines Eckpunkts kleiner oder die kleinste x-Koordinate eines Eckpunkts größer ist als p_1 oder
- die größte y-Koordinate eines Eckpunkts kleiner oder die kleinste y-Koordinate eines Eckpunkts größer ist als p_2.

Programm 3.3: find_encircling_triangle.m

```
function [index_triangle, pos_point] ...
  = find_encircling_triangle(coordinates, triangulation,...
      index_next_triangle, P)

  indice = 1:index_next_triangle-1;

  I = find ((coordinates(triangulation(indice,1),1)>=P(1)  | ...
             coordinates(triangulation(indice,2),1)>=P(1)  | ...
             coordinates(triangulation(indice,3),1)>=P(1)) & ...
```

```
10            (coordinates(triangulation(indice,1),1)<=P(1)  | ...
11             coordinates(triangulation(indice,2),1)<=P(1)  | ...
12             coordinates(triangulation(indice,3),1)<=P(1)));
13
14  if isempty(I)
15    index_triangle = -1;
16    pos_point = 0;
17    return;
18  end
19
20  J = (coordinates(triangulation(I,1),2)>=P(2)  | ...
21       coordinates(triangulation(I,2),2)>=P(2)  | ...
22       coordinates(triangulation(I,3),2)>=P(2)) & ...
23      (coordinates(triangulation(I,1),2)<=P(2)  | ...
24       coordinates(triangulation(I,2),2)<=P(2)  | ...
25       coordinates(triangulation(I,3),2)<=P(2));
26
27  I = I(J);
28  cnt_triangles = size(I,1);
29
30  including_triangles = zeros(cnt_triangles,1);
31  next_triangle = 1;
32
33  for i = 1: cnt_triangles
34    det_trian_1 = det([P(1),P(2),1; ...
35                       coordinates(triangulation(I(i),2), :),1; ...
36                       coordinates(triangulation(I(i),3), :),1]);
37    det_trian_2 = det([P(1),P(2),1; ...
38                       coordinates(triangulation(I(i),3), :),1; ...
39                       coordinates(triangulation(I(i),1), :),1]);
40    det_trian_3 = det([P(1),P(2),1; ...
41                       coordinates(triangulation(I(i),1), :),1; ...
42                       coordinates(triangulation(I(i),2), :),1]);
43
44    if sign(det_trian_1) >= 0 && sign(det_trian_2) >= 0 ...
45       && sign(det_trian_3) >= 0
46        including_triangles(next_triangle) = I(i);
47        next_triangle = next_triangle + 1;
48    end
49  end
50
51  including_triangles(next_triangle:end) = [];
52
53  if isempty(including_triangles)
54    index_triangle = -1;
55    pos_point = 0;
56    return;
57  end
58
59  if size(including_triangles, 1) == 1
60    % Punkt liegt nur in einem Dreieck ~> Inneres
61    index_triangle = including_triangles;
62    pos_point = 1;
63  elseif size(including_triangles, 1) == 2
```

```
64      % Punkt liegt in zwei Dreiecken ~> Kante; gib erstes Dreieck aus
65      index_triangle = including_triangles(1);
66      pos_point = 2;
67    else
68      % Punkt liegt in mehr als zwei Dreiecken ~> Ecke; gib erstes
69      % Dreieck aus
70      index_triangle = including_triangles(1);
71      pos_point = 3;
72    end
73
74  end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
5	Es müssen nur die Zeilen von `triangulation` überprüft werden, in denen bisher Dreiecke eingetragen wurden.
7-12	Finde alle Dreiecke, die einen Eckpunkt mit x-Koordinate kleiner p_1 und einen mit x-Koordinate größer p_1 haben.
14-18	Erfüllt kein Dreieck die Bedingung, so gibt es kein umgebendes Dreieck.
20-25	Finde aus allen in den Zeilen 7-12 gefundenen Dreiecken diejenigen heraus, die eine äquivalente Bedingung für die y-Koordinate erfüllen.
34-42	Berechne für jedes Dreieck, das alle Bedingungen erfüllt, die jeweiligen Determinanten.
44-48	Sind alle Determinanten positiv (Dreieck hat positive Orientierung) oder gleich Null (Punkt liegt auf einer Kante oder, sind zwei Determinanten gleich Null, in einer Ecke), so beinhaltet das Dreieck den Punkt.
53-72	Beinhaltet kein Dreieck den Punkt, so gib -1 zurück. Andernfalls wird der Index des Dreiecks zurückgegeben. Die Variable `pos_point` enthält die Anzahl der umgebenden Dreiecke. Liegt der Punkt im Innern eines Dreiecks, so gibt es nur ein umgebendes Dreieck; liegt er auf einer Kante, ist er Teil zweier Dreiecke; liegt er auf einer Ecke, ist er mindestens Teil dreier Dreiecke.

Hilfsfunktion 3.19 (`get_smallest_angle`)
Wir müssen überprüfen können, ob noch zu kleine Winkel in der Triangulierung vorhanden sind. Dafür benötigen wir eine Funktion, die uns zu jedem Dreieck den kleinsten Winkel bestimmt.

Der kleinste Winkel liegt dabei immer gegenüber der kürzesten Kante des Dreiecks. Den Winkel - beziehungsweise dessen Cosinus - berechnen wir über den Cosinussatz 3.1. Zusätzlich zum kleinsten Winkel speichern wir für jedes Dreieck noch, an welcher Position dieser Winkel liegt.

Programm 3.4: get_smallest_angle.m

```
1  function [angle_marker] = get_smallest_angle(edges_length, ...
2      triangles_with_edges, cnt_triangles)
3    % bestimmt cos des kleinsten Winkels in jedem Dreieck sowie dessen
4    % Position (also Pos des Punktes, wo er vorliegt)
5
```

```
6   succ =[2 3 1];
7
8   angle_marker = zeros(cnt_triangles, 2);
9
10  edges_length = edges_length(triangles_with_edges);
11
12  [length_AB, index_A] = min(edges_length(:,:), [], 2);
13  index_B = succ(index_A)';
14  angle_marker(:,2) = succ(index_B);
15
16  length_BC = diag(edges_length(:, index_B));
17  length_CA = diag(edges_length(:, angle_marker(:, 2)));
18
19  % Sei die kürzeste Kante AB, dann ist der kleinste Winkel gamma
20  angle_marker(:,1) = (length_BC .* length_BC ...
21                     + length_CA .* length_CA ...
22                     - length_AB .* length_AB) ...
23                     ./ (2*length_BC.*length_CA);
24
25  end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
8	Initialisiere die Variable `angle_marker`. Zu jedem Dreieck wird in der ersten Spalte der Cosinus des kleinsten Winkels, in der zweiten Spalte die Position des kleinsten Winkels gespeichert.
10	Überschreibe `edges_length` so, dass zu k Dreiecken eine $k \times 3$-Struktur entsteht. Diese hat an Position `(i,j)` die Länge der `j`-ten Kante im `i`-ten Dreieck gespeichert.
12	Suche zeilenweise das Minimum in `edges_length`, also die Länge der kürzesten Kante und ihre Position. Diese Kante sei AB, ihre Position ist auch die von A. `index_A` beinhaltet nun zu jedem der k Dreiecke die Position der kürzesten Kante.
13-14	Bestimme die Positionen von B (Nachfolger von A) und C (Nachfolger von B, Position des kleinsten Winkels).
16	Bestimme die Längen der Kanten BC. `edges_length(:, index_B)` gibt dabei eine $k \times k$ Matrix aus. In der `i`-ten Spalte dieser Matrix steht die `index_B(i)`-te Spalte von `edges_length`. Gesucht ist aus der ersten Spalte der erste Eintrag, der zweite aus der zweiten Spalte und so weiter. Wir benötigen demnach die Diagonale der erzeugten Matrix.
17	Die Längen der Kanten CA werden analog bestimmt.
20-23	Der kleinste Winkel wird anhand des Cosinussatzes 3.1 bestimmt.

Hilfsfunktion 3.20 (`insert_circumcentre`)
Wir benötigen eine Funktion, die den Umkreismittelpunkt eines Dreiecks einfügt, wenn dieser im Dreieck selbst liegt.

Es sei also X der Umkreismittelpunkt des Dreiecks ABC. Dann ergeben sich durch Einfügen des Punktes zwei neue Dreiecke. Die Indizierung wird in der folgenden Abbildung 3.16 beschrieben.

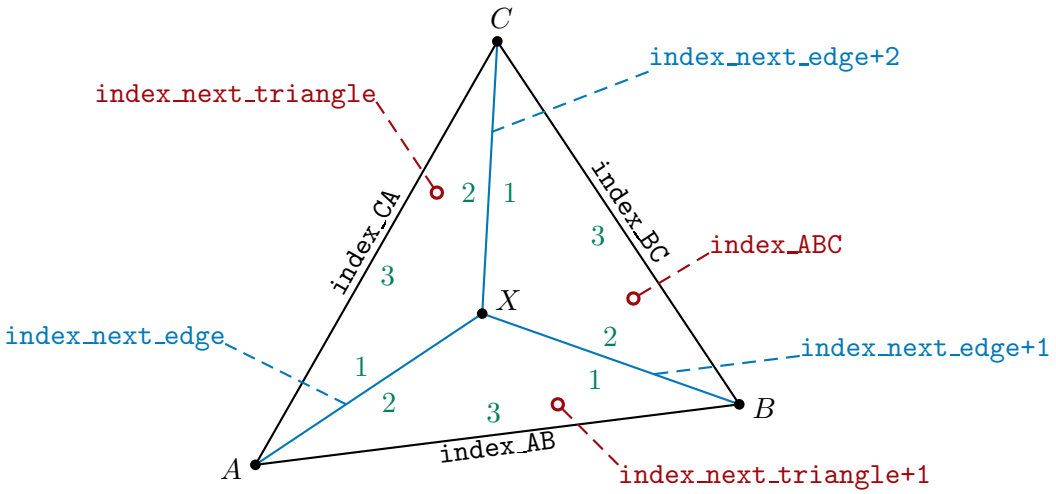


Abbildung 3.16: Indizierung beim Einfügen des Umkreismittelpunktes

Programm 3.5: insert_circumcentre.m

```
function [coordinates, triangulation, index_next_point, ...
    index_next_edge, index_next_triangle, edge_marker, edges_length, ...
    edges_with_triangles, triangles_with_edges] ...
  = insert_circumcentre(coordinates, triangulation, index_ABC, ...
    index_next_point, index_next_edge, index_next_triangle, ...
    edge_marker, edges_length, edges_with_triangles, ...
    triangles_with_edges)

  % fügt den Umkreismittelpunkt, der im Innern des Dreiecks ABC liegt,
  % als neuen Punkt ein

  index_A = triangulation(index_ABC, 1);
  index_B = triangulation(index_ABC, 2);
  index_C = triangulation(index_ABC, 3);

  index_AB = triangles_with_edges(index_ABC, 1);
  index_BC = triangles_with_edges(index_ABC, 2);
  index_CA = triangles_with_edges(index_ABC, 3);

  coordinates(index_next_point, :) ..
    = get_circumcentre(coordinates(index_A,:), ...
                       coordinates(index_B,:), ...
                       coordinates(index_C,:));


  % ---------------------- Kanten aktualisieren -----------------------
  % XA einfügen
  edges_with_triangles(index_next_edge,:) = [ ...
    index_next_point, index_A, index_next_triangle, 1, ...
                               index_next_triangle+1, 2];

  % XB einfügen
  edges_with_triangles(index_next_edge+1,:) = [ ...
```

```
34      index_next_point, index_B, index_next_triangle+1, 1, index_ABC, 2];
35
36    % XC einfügen
37    edges_with_triangles(index_next_edge+2,:) = [ ...
38      index_next_point, index_C, index_ABC, 1, index_next_triangle, 2];
39
40    % AB aktualisieren
41    if edges_with_triangles(index_AB, 3) == index_ABC
42      edges_with_triangles(index_AB, 3:4) = [index_next_triangle+1, 3];
43    else
44      edges_with_triangles(index_AB, 5:6) = [index_next_triangle+1, 3];
45    end
46
47    % BC aktualisieren
48    if edges_with_triangles(index_BC, 3) == index_ABC
49      edges_with_triangles(index_BC, 4) = 3;
50    else
51      edges_with_triangles(index_BC, 6) = 3;
52    end
53
54    % CA aktualisieren
55    if edges_with_triangles(index_CA, 3) == index_ABC
56      edges_with_triangles(index_CA, 3:4) = [index_next_triangle, 3];
57    else
58      edges_with_triangles(index_CA, 5:6) = [index_next_triangle, 3];
59    end
60
61
62    % ------------------------ Kanten markieren --------------------------
63    edge_marker([index_AB, index_BC, index_CA, index_next_edge, ...
64      index_next_edge+1, index_next_edge+2]) = 1;
65
66
67    % ------------------- bestimme neue Kantenlängen ---------------------
68    % XA
69    edges_length(index_next_edge) = length_edge(circumcentre, ...
70      coordinates(index_A, :));
71
72    % XB
73    edges_length(index_next_edge+1) = length_edge(circumcentre, ...
74      coordinates(index_B, :));
75
76    % XC
77    edges_length(index_next_edge+2) = length_edge(circumcentre, ...
78      coordinates(index_C, :));
79
80
81    % --------------------- Dreiecke aktualisieren ----------------------
82    % AXC einfügen
83    triangulation(index_next_triangle, :) = [...
84      index_A, index_next_point, index_C ];
85
86    triangles_with_edges(index_next_triangle, :) = [...
87      index_next_edge, index_next_edge+2, index_CA ];
```

```
88
89    % BXA einfügen
90    triangulation(index_next_triangle+1, :) = [...
91      index_B, index_next_point, index_A ];
92
93    triangles_with_edges(index_next_triangle+1, :) = [...
94      index_next_edge+1, index_next_edge, index_AB ];
95
96    % CXB statt ABC einfügen
97    triangulation(index_ABC, :) = [...
98      index_C, index_next_point, index_B ];
99
100   triangles_with_edges(index_ABC, :) = [...
101     index_next_edge+2, index_next_edge+1, index_BC ];
102
103
104   % ---------- zugefügt wurden drei Kanten und zwei Dreiecke ----------
105   index_next_edge = index_next_edge + 3;
106   index_next_triangle = index_next_triangle + 2;
107   index_next_point = index_next_point + 1;
108 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
12-18	Bestimme die Indizes der Punkte A, B und C sowie die der Kanten AB, BC und CA.
20-23	Bestimme den Umkreismittelpunkt X von ABC und füge ihn als neuen Punkt ein.
27-38	Füge die Kanten XA, XB und XC ein. Sämtliche Einträge ergeben sich aus Abbildung 3.16.
40-59	Aktualisiere die Kanten AB, BC und CA.
63-64	Markiere alle Kanten, um sie durch den *edge-flip* überprüfen zu lassen.
69-78	Die Längen der Außenkanten haben sich nicht verändert. Bestimme die Längen der drei neuen Kanten.
83-101	Aktualisiere die Variablen `triangulation` und `triangles_with_edges`.
105-107	Inkrementiere die Indizes für neu einzufügende Punkte, Kanten und Dreiecke.

Hilfsfunktion 3.21 (`open_skinny_angle`)
In Pseudocode 3.13 wurde das Aufsperren zu kleiner Winkel dargestellt. Dies wird durch das folgende Programm realisiert.

Programm 3.6: open_skinny_angle.m

```
1 function [coordinates, triangulation, edges_with_triangles, ...
2     triangles_with_edges, index_next_point, index_next_edge, ...
3     index_next_triangle, edge_marker, edges_length] ...
4   = open_skinny_angle(coordinates, triangulation, ...
5     edges_with_triangles, triangles_with_edges, edges_length, ...
6     index_next_point, index_next_edge, index_next_triangle, ...
7     index_skinny_triangle, edge_marker, alpha_min, h_max)
```

```
8
9   % Geg.: ABC an Pos. index_skinny_triangle
10
11  succ = [2 3 1];
12
13  tol = h_max * 1e-12;
14
15  % Der UMP von ABC liegt in ABC, falls größter Winkel mind pi/2
16  [length_AB, pos_A] = max(edges_length(triangles_with_edges(...
17                                          index_skinny_triangle,:)));
18  pos_B = succ(pos_A);
19  pos_C = succ(pos_B);
20
21  length_BC = edges_length(triangles_with_edges( ...
22                                       index_skinny_triangle, pos_B));
23  length_CA = edges_length(triangles_with_edges( ...
24                                       index_skinny_triangle, pos_C));
25
26  % überprüfe, ob für gamma größter Winkel cos(gamma) < tol
27  if (length_BC * length_BC + length_CA * length_CA - ...
28      length_AB * length_AB) / (2 * length_BC * length_CA) < tol
29    % UMP von ABC liegt in PQR
30    circumcentre = get_circumcentre(...
31      coordinates(triangulation(index_skinny_triangle, pos_A),:), ...
32      coordinates(triangulation(index_skinny_triangle, pos_B),:), ...
33      coordinates(triangulation(index_skinny_triangle, pos_C),:));
34
35    % bestimme PQR
36    [index_PQR, pos_point] = find_encircling_triangle(coordinates,...
37         triangulation, index_next_triangle, circumcentre);
38
39    if pos_point == 0
40      % UMP liegt außerhalb des Polygons
41      % halbiere die längste Kante (dh AB)
42      index_encroached_edge = triangles_with_edges(...
43                                        index_skinny_triangle, pos_A);
44
45      [coordinates, triangulation, edges_with_triangles,...
46        triangles_with_edges, index_next_point, index_next_edge,...
47        index_next_triangle, edge_marker, edges_length]...
48      = split_encroached_edge(coordinates, triangulation, ...
49        edges_with_triangles, triangles_with_edges, edge_marker,...
50        edges_length, index_next_point, index_next_edge,...
51        index_next_triangle, index_encroached_edge);
52    else
53      % UMP von ABC liegt in PQR
54
55      % es gibt umschließendes Dreieck PQR
56      % bestimme längste Kante PQ
57      [length_PQ, pos_P] = max(edges_length(triangles_with_edges( ...
58                                                  index_PQR,:)));
59
60      if is_viewable(coordinates, ...
61           edges_with_triangles(edges_with_triangles(:,5) == 0,1:2), ...
```

```
62            triangulation(index_skinny_triangle, pos_C), ...
63            circumcentre, 1)
64          if edges_with_triangles(triangles_with_edges(index_PQR,pos_P),5)
65            % PQ keine Randkante
66            epsilon = 0;
67          else
68            % PQ Randkante
69            epsilon = alpha_min;
70          end
71          % bestimme größten Winkel von PQR (gegenüber von PQ)
72          pos_Q = succ(pos_P);
73          pos_R = succ(pos_Q);
74
75          length_QR = edges_length(triangles_with_edges( ...
76                                          index_skinny_triangle, pos_Q));
77          length_RP = edges_length(triangles_with_edges( ...
78                                          index_skinny_triangle, pos_R));
79
80          % überprüfe, ob für gamma größter Winkel
81          % cos(gamma) < cos(pi/2 - epsilon)
82          cos_pi_eps = cos(pi/2 - epsilon);
83          if (length_QR * length_QR + length_RP * length_RP - ...
84             length_PQ * length_PQ) / (2 * length_QR * length_RP) ...
85             < cos_pi_eps
86
87            % UMP zu nah an Kante
88            % halbiere längste Kante PQ
89            index_encroached_edge = triangles_with_edges(index_PQR,pos_P);
90
91            [coordinates, triangulation, edges_with_triangles,...
92              triangles_with_edges, index_next_point, index_next_edge,...
93              index_next_triangle, edge_marker, edges_length]...
94             = split_encroached_edge(coordinates, triangulation, ...
95              edges_with_triangles, triangles_with_edges, edge_marker,...
96              edges_length, index_next_point, index_next_edge,...
97              index_next_triangle, index_encroached_edge);
98          else
99            % UMP als neuen Punkt einfügen
100           % füge circumcentre als neuen Punkt ein
101           [coordinates, triangulation, index_next_point, ...
102             index_next_edge, index_next_triangle, edge_marker, ...
103             edges_length, edges_with_triangles, triangles_with_edges]...
104           = insert_circumcentre(coordinates, triangulation, ...
105             circumcentre, index_PQR, index_next_point, ...
106             index_next_edge, index_next_triangle, edge_marker, ...
107             edges_length, edges_with_triangles, triangles_with_edges);
108         end
109       else
110         index_encroached_edge = triangles_with_edges(...
111                                          index_skinny_triangle, pos_A);
112
113         [coordinates, triangulation, edges_with_triangles,...
114           triangles_with_edges, index_next_point, index_next_edge,...
115           index_next_triangle, edge_marker, edges_length]...
```

```
116           = split_encroached_edge(coordinates, triangulation, ...
117             edges_with_triangles, triangles_with_edges, edge_marker,...
118             edges_length, index_next_point, index_next_edge,...
119             index_next_triangle, index_encroached_edge);
120         end
121       end
122     else
123       % UMP liegt innerhalb
124       % halbiere die längste Kante (dh AB)
125       index_encroached_edge = triangles_with_edges(...
126                                               index_skinny_triangle, pos_A);
127
128       [coordinates, triangulation, edges_with_triangles,...
129         triangles_with_edges, index_next_point, index_next_edge,...
130         index_next_triangle, edge_marker, edges_length]...
131       = split_encroached_edge(coordinates, triangulation, ...
132         edges_with_triangles, triangles_with_edges, edge_marker,...
133         edges_length, index_next_point, index_next_edge,...
134         index_next_triangle, index_encroached_edge);
135     end
136
137 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
13	Die Fehlertoleranz wird erneut in Abhängigkeit von der maximalen Netzweite berechnet.
16-17	Bestimme die längste Kante im Dreieck ABC sowie deren Position im Dreieck.
18-24	Bestimme die Positionen der Kanten BC und CA sowie deren Längen.
27-28	Berechne mit Hilfe des Cosinussatzes 3.1 den Cosinus des größten Winkels. Ist dieser kleiner als die Fehlertoleranz (abgesehen von Rundungsfehlern also kleiner oder gleich Null), so ist der größte Winkel größer als $\pi/2$. Dementsprechend liegt der Umkreismittelpunkt von ABC in einem anderen Dreieck PQR.
30-37	Berechne den Umkreismittelpunkt von ABC und finde das umgebende Dreieck PQR.
39-51	Wird kein umgebendes Dreieck gefunden, wird die längste Kante von ABC, AB, halbiert.
57-58	Bestimme die längste Kante von PQR und deren Position im Dreieck.
60-63	Überprüfe, ob der Umkreismittelpunkt von ABC von C aus sichtbar ist bezüglich der Nebenbedingungskanten. Andernfalls weiter in Zeile 109.
64-70	Bestimme ε analog zu Pseudocode 3.13.
72-78	Bestimme die Positionen der Kanten QR und RP sowie deren Längen.
82	Berechne als Vergleichswert $\cos(\pi/2 - \varepsilon)$.
83-97	Ist der Cosinus des größten Winkels kleines als der in Zeile 82 berechnete Wert, so ist der größte Winkel von PQR zu groß: Der Umkreismittelpunkt liegt entweder auf der Kante oder außerhalb des Dreiecks. Halbiere die längste Kante von PQR.

Zeilen	Funktionalität
98-108	Der Umkreismittelpunkt von PQR liegt innerhalb des Dreiecks. Füge diesen Punkt als neuen Punkt ein.
109-120	Der Umkreismittelpunkt von ABC liegt zwar in einem von ABC verschiedenen Dreieck, ist allerdings wegen der Nebenbedingung nicht sichtbar. Halbiere stattdessen die längste Kante von ABC.
123-134	Der Umkreismittelpunkt von ABC liegt im Dreieck selbst. Da durch Einfügen der kleinste Winkel verkleinert wird, wird stattdessen die längste Kante halbiert.

3.2.2 Realisierung *refinement*

Wir haben nun alle für den *refinement*-Algorithmus benötigten Hilfsfunktionen besprochen. Damit können wir die Implementierung des Pseudocode 3.14 betrachten.

Programm 3.7: delaunay_refinement.m

```
1  function [coordinates, triangulation] ...
2    = delaunay_refinement(coordinates, triangulation, edge_marker,...
3      edges_with_triangles, triangles_with_edges, edges_length, ...
4      h_max, alpha_min)
5
6    tol = h_max * 1e-14;
7    cos_alpha_min = cos(alpha_min);
8
9    % bestimme die Positionen, an denen der nächste Punkt, die nächste
10   % Kante und das nächste Dreieck eingetragen werden
11   index_next_edge = size(edges_with_triangles,1) + 1;
12   index_next_point = size(coordinates,1) + 1;
13   index_next_triangle = size(triangulation,1) + 1;
14
15   % vergrößere die verwendeten Arrays, um schnelleres Eintragen möglich
16   % zu machen
17   coordinates = [coordinates; zeros(20*index_next_point,2)];
18   edges_with_triangles = [edges_with_triangles; ...
19                           zeros(60*index_next_edge, 6)];
20   triangles_with_edges = [triangles_with_edges; ...
21                           zeros(40*index_next_triangle, 3)];
22   triangulation = [triangulation; zeros(40*index_next_triangle, 3)];
23   edges_length = [edges_length; zeros(60*index_next_edge, 1)];
24   edge_marker = [edge_marker; zeros(60*index_next_edge,1)];
25
26
27   % ------------------------ Kanten verkürzen -------------------------
28   [length_longest_edge, index_longest_edge] = max(edges_length);
29
30   % solange die längste Kante zu lang ist
31   while length_longest_edge > h_max
32
33     % Kante halbieren
34     [coordinates, triangulation, edges_with_triangles, ...
35       triangles_with_edges, index_next_point, index_next_edge, ...
36       index_next_triangle, edge_marker, edges_length]...
37     = split_encroached_edge(coordinates, triangulation, ...
```

```
38        edges_with_triangles, triangles_with_edges, edge_marker, ...
39        edges_length, index_next_point, index_next_edge, ...
40        index_next_triangle, index_longest_edge);
41
42      % edge-flip
43      [triangulation, edges_with_triangles, triangles_with_edges, ...
44        edges_length, edge_marker] ...
45      = edge_flip(coordinates, triangulation, edges_with_triangles,...
46        triangles_with_edges, edges_length, edge_marker, h_max);
47
48      % Index und Länge der jetzt längsten Kante bestimmen
49      [length_longest_edge, index_longest_edge] = max(edges_length);
50    end
51
52
53    % ------------------------ Winkel vergrößern -------------------------
54    % bestimme den cos des kleinsten Winkels jedes Dreiecks
55    angle_marker = get_smallest_angle(edges_length, ...
56                            triangles_with_edges, index_next_triangle-1);
57
58    skinniest_triangle = find(max(angle_marker(:,1)) == angle_marker(:,1));
59
60    % speichere Größe des aktuell kleinsten Winkels
61    smallest_angle_begin = acos(angle_marker(skinniest_triangle(1),1)) ...
62                           /pi*180;
63
64    % speichere Triangulierung mit verkürzten Kanten sowies verwendete
65    % Punkte und Indizes, falls Winkelverkleinerung nicht erfolgreich
66    short_triangulation = triangulation(1:index_next_triangle-1,:);
67    short_coordinates = coordinates(1:index_next_point-1,:);
68
69    % speichere die Fläche des Dreiecks mit dem kleinsten Winkel und der
70    % kürzesten Kante
71    if size(skinniest_triangle,1) > 1
72      [tmp_length, tmp_pos] = min(edges_length(triangles_with_edges( ...
73                                                skinniest_triangle,:)));
74      [~, pos] = min(tmp_length);
75      smallest_triangle = skinniest_triangle(tmp_pos(pos(1)));
76    else
77      smallest_triangle = skinniest_triangle;
78    end
79    smallest_area_start...
80      = det([coordinates(triangulation(smallest_triangle,1),:), 1;...
81             coordinates(triangulation(smallest_triangle,2),:), 1;...
82             coordinates(triangulation(smallest_triangle,3),:), 1]) / 2;
83    smallest_area = smallest_area_start;
84
85    terminated = 0;
86
87    while smallest_area > smallest_area_start / 100
88
89      % finde alle Dreiecke, deren beiden längeren Kanten
90      % Nebenbedingungskanten sind
91      boundary_triangle_marker = mark_boundary_triangles( ...
```

```
92          triangles_with_edges, edges_with_triangles, edges_length, ...
93          index_next_triangle-1);
94
95      % entferne bei all diesen Dreiecken (hier müssen wir mit evtl zu
96      % kleinem Winkel leben) den cos des kleinsten Winkels - dadurch
97      % werden diese Dreiecke nicht in Betracht gezogen
98      angle_marker(boundary_triangle_marker(:) == 1, 1:2) = 0;
99
100     skinny_triangles = find(angle_marker(:,1) > cos_alpha_min);
101     if isempty(skinny_triangles)
102       terminated = 1;
103       break
104     end
105
106     % bestimme Dreieck mit dem kleinsten Winkel
107     skinniest_triangle = find(max(angle_marker(:,1)) == ...
108                                                  angle_marker(:,1));
109
110     % falls mehrere gleich spitze Dreiecke vorhanden, nimm zuerst das
111     % mit der längsten Kante - bestimme außerdem das mit der kürzesten
112     % Kante
113     if size(skinniest_triangle,1) > 1
114       [tmp_length, tmp_pos] = min(edges_length(triangles_with_edges( ...
115                                                 skinniest_triangle,:)));
116       [~, pos] = min(tmp_length);
117       smallest_triangle = skinniest_triangle(tmp_pos(pos(1)));
118
119       [tmp_length, tmp_pos] = max(edges_length(triangles_with_edges( ...
120                                                 skinniest_triangle,:)));
121       [~, pos] = max(tmp_length);
122       skinniest_triangle = skinniest_triangle(tmp_pos(pos(1)));
123     else
124       smallest_triangle = skinniest_triangle;
125     end
126
127     smallest_area ...
128       = det([coordinates(triangulation(smallest_triangle,1),:), 1;...
129              coordinates(triangulation(smallest_triangle,2),:), 1;...
130              coordinates(triangulation(smallest_triangle,3),:), 1]) / 2;
131
132     % kleinsten Winkel vergrößern
133     [coordinates, triangulation, edges_with_triangles, ...
134       triangles_with_edges, index_next_point, index_next_edge, ...
135       index_next_triangle, edge_marker, edges_length] ...
136     = open_skinny_angle(coordinates, triangulation, ...
137       edges_with_triangles, triangles_with_edges, edges_length, ...
138       index_next_point, index_next_edge, index_next_triangle, ...
139       skinniest_triangle, edge_marker, alpha_min, h_max);
140
141     % edge-flip
142     [triangulation, edges_with_triangles, triangles_with_edges, ...
143       edges_length, edge_marker] ...
144     = edge_flip(coordinates, triangulation, edges_with_triangles,...
145       triangles_with_edges, edges_length, edge_marker, h_max);
```

```
146
147     angle_marker = get_smallest_angle(edges_length, ...
148                                 triangles_with_edges, index_next_triangle-1);
149   end
150
151   % falls der Winkel vorher nicht groß genug war
152   if smallest_angle_begin < alpha_min && ~terminated
153     fprintf('\nDer␣geforderte␣Minimalwinkel␣konnte␣nicht␣erreicht\n');
154     fprintf('werden.␣Alle␣Kanten␣wurden␣soweit␣nötig␣verkürzt.␣Der\n');
155     fprintf('kleinste␣vorkommende␣Winkel␣in␣der␣ausgegebenen\n');
156     fprintf('Triangulierung␣hat␣eine␣Größe␣von␣%f2␣Grad.\n\n', ...
157             smallest_angle_begin);
158     triangulation = short_triangulation;
159     coordinates = short_coordinates;
160   end
161
162   % lösche am Ende die nicht verwendeten Einträge in den Array
163   coordinates(index_next_point:end, :) = [];
164   triangulation(index_next_triangle:end, :) = [];
165
166 end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
6	Lege eine Fehlertoleranz fest.
7	Da stets nur die Cosinuswerte aller Winkel überprüft werden, wird auch der Cosinus des Minimalwinkels $\alpha_{\min}$ benötigt.
11-13	Bestimme jeweils den Index für den nächsten Punkt, die nächste Kante und das nächste Dreieck.
17-24	Vergrößere alle beteiligten Datenstrukturen, um schnelleres Schreiben zu ermöglichen. Dabei werden pro eingetragenen Punkt maximal drei Kanten und zwei Dreiecke eingefügt.
28	Bestimme die Länge und den Index der längsten Kante.
ab 31	Solange die längste Kante zu lang ist: Kanten verkürzen.
34-40	Rufe die Funktion zum Halbieren der längsten Kante auf.
43-46	Führe einen *edge-flip* durch, um wieder eine Delaunay-Triangulierung zu erhalten, bevor die nächste Kante verkürzt wird.
49	Finde die nächste längste Kante und ihre Länge.
55-56	Bestimme zu jedem Dreieck den Cosinus des kleinsten Winkel.
58	Bestimme das Dreieck mit dem kleinsten Winkel, also dem größten Cosinus.
61-62	Speichere die Größe des aktuell kleinsten Winkels für die Ausgabe im Falle eines Abbruchs.
66-67	Speichere die aktuelle Triangulierung sowie alle verwendeten Punkte.
71-83	Speichere die Fläche des Dreiecks mit dem kleinsten Winkel und der kürzesten Kante. Diese Fläche wird als Vergleich für das Abbruchkriterium verwendet.
85	Setze eine Variable, um zu überprüfen, ob die `while`-Schleife selbst terminiert ist.

Zeilen	Funktionalität
ab 70	Solange es zu kleine Winkel gibt und die kleinste Dreiecksfläche groß genug ist, vergrößere diese.
91-93	Bestimme alle Dreiecke, deren beide längste Kanten Randkanten sind. In diesen Dreiecken sind zu spitze Winkel durch die Nebenbedingung vorgegeben und können nicht vergrößert werden.
98	Entferne bei den eben markierten Dreiecken den Cosinus des kleinsten Winkels. Dadurch werden diese Dreiecke nicht betrachtet.
100-104	Finde alle Dreiecke, die einen zu spitzen Winkel haben.
107-108	Finde aus all diesen Dreiecken das spitzeste Dreieck.
113-125	Sind mehrere Dreiecke gleich spitz, verändere zunächst das mit der längsten Kante. Bestimme außerdem das mit der kürzesten Kante.
127-130	Berechne die Fläche des eben bestimmten kleinsten Dreiecks.
133-139	Sperre den kleinsten Winkel auf.
142-145	Führe *edge-flip* durch.
147-148	Bestimme erneut die Größen der Winkel.
152-160	War zu Beginn der kleinste Winkel zu klein und hat die `while`-Schleife nicht terminiert, so ist der geforderte Winkel $\alpha_{\min}$ nicht zu erreichen. Gebe die in den Zeilen 66-67 gespeicherten Variablen aus und drucke die Größe des kleinsten Winkels in dieser Triangulierung.
163-164	Terminiert das Winkel verkleinern, müssen gegebenenfalls noch leere Einträge am Ende von `coordinates` und `triangulation` entfernt werden, bevor diese ausgegeben werden.

4 Programm zur Delaunay-Triangulierung

In diesem letzten Kapitel werden wir schließlich das Programm zur Delaunay-Triangulierung betrachten. Dabei hat der hier vorgestellte Quellcode keine bisher nicht beschriebene Funktionalität. Er dient lediglich zum Aufrufen aller bisher besprochenen Funktionen.

4.1 Implementierung

Die Funktionsaufrufe im Programm `delaunay_triangulation` erfolgen - soweit sinnvoll - in der gleichen Reihenfolge, in der wir sie auch in der vorliegenden Arbeit betrachtet haben.

Die Funktionen sind für eine bessere Übersicht kapitelweise gruppiert. In der Erklärung des Quellcodes finden sich Verweise zu den einzelnen Funktionen.

Programm 4.1: delaunay_triangulation.m

```
1  function [coordinates, triangulation] = ...
2    delaunay_triangulation(coordinates, edges, h_max, alpha_min)
3    %Erzeugt eine Delaunay-Triangulierung des von edges umschlossenen
4    % Gebiets.
5    %
6    % Input:    coordinates     px2 Koordinaten
7    %           edges           kx2 Kanten, geschlossener Polygonzug
8    %           h_max           maximale Netzweite
9    %           alpha_min       minimaler Winkel im Bogenmaß
10   %
11   % Output:   coordinates     mx2 Punkte, m >= p
12   %           triangulation   nx3 Triangulierung
13
14
15   % ---------------------------- planesweep -----------------------------
16   % Funktionsaufruf planesweep
17   [coordinates, triangulation, boundary] =...
18         planesweep(coordinates, edges, h_max);
19
20   % sorge für korrekte Orientierung der Dreiecke
21   triangulation = assure_orientation(coordinates, triangulation);
22
23   % finde alle Dreiecke, die im Innern liegen
24   [triangle_marker] = pnpoly(coordinates, triangulation, boundary);
25
26   % entferne alle Dreiecke aus der Triangulierung, die nicht im Innern
27   % liegen
28   triangulation = triangulation(find(triangle_marker),:);
29
30
31   % ---------------------- Aufbau Datenstrukturen ----------------------
32   % erzeuge Kanten mit Dreieckszugehörigkeit
```

```
33    edges = generate_edges(triangulation);
34
35    % erzeuge Liste aller Kanten sowie die Dreiecke, in denen sie vor-
36    % kommen (inkl Position)
37    edges_with_triangles = triangle2edge(triangulation, edges);
38
39    % bestimme die Länge aller Kanten
40    edges_length = edge_length_in_triangle(coordinates, ...
41                                           edges_with_triangles);
42
43    % erzeuge Liste, die jeder Kante in jedem Dreieck den Kantenindex in
44    % edges_with_triangles zuweist
45    triangles_with_edges = edge2triangle(edges_with_triangles);
46
47
48    % ---------------------------- edge-flip -----------------------------
49    % markiere alle aktuell eingefügten Kanten
50    edge_marker = ones(size(edges_with_triangles,1),1);
51
52    % führe edge-flip durch - anschließend ist der edge-marker Nullvektor
53    [triangulation, edges_with_triangles, triangles_with_edges, ...
54        edges_length, edge_marker] ...
55    = edge_flip(coordinates, triangulation, edges_with_triangles,...
56        triangles_with_edges, edges_length, edge_marker, h_max);
57
58
59    % ---------------------------- refinement ----------------------------
60    % rufe Delaunay-Refinement auf
61    [coordinates, triangulation]...
62    = delaunay_refinement(coordinates, triangulation, edge_marker,...
63        triangle_marker, edges_with_triangles, triangles_with_edges, ...
64        edges_length, h_max, alpha_min);
65
66  end
```

Erläuterung des Quellcodes

Zeilen	Funktionalität
17-18	Erzeuge die Triangulierung durch den *planesweep*, siehe dazu Seite 23.
21	Die Orientierung aller Dreiecke wird überprüft. Anschließend werden alle Eckpunkte jeweils im positiven Sinn aufgelistet. Siehe dazu Seite 29.
24	Es wird eine Triangulierung des vom Polygonzug eingeschlossenen Gebiets gesucht. Daher werden alle Dreiecke markiert, die im Innern des Polygons liegen. Siehe dazu Seite 30.
28	Entferne alle Dreiecke aus der Triangulierung, die in Zeile 24 nicht markiert wurden.
33	Erzeuge die Datenstruktur `edges`. Diese wird benötigt um die nächste Datenstruktur zu erzeugen. Siehe dazu Seite 41.
37	Erzeuge mit `edges_with_triangles` eine Liste, die sämtliche Kanten, die Indizes der zugehörigen Dreiecke sowie die Position der Kante im Dreieck vermerkt. Siehe dazu Seite 42.

Zeilen	Funktionalität
40-41	Berechne sämtliche Kantenlängen und speichere sie als `edges_length`. Siehe dazu auch Seite 48.
45	Erzeuge die Datenstruktur `triangles_with_edges` als Zuweisung von jedem Dreieck auf die beteiligten Kanten. Siehe dazu Seite 46.
50	Markiere einmalig alle vorkommenden Kanten, so dass der *edge-flip* für sämtliche Kanten überprüft, ob sie nach Definition 2.3 lokal Delaunay sind.
53-56	Führe den *edge-flip* aus. Anschließend sind alle Kanten lokal Delaunay. Siehe dazu Seite 50.
61-64	Verfeinere die Triangulierung durch den *refinement*-Algorithmus so, dass die geforderten Optimalitätsbedingungen erfüllt sind. Siehe dazu auch Seite 87.

4.2 Fazit

Gegeben war ein endliches Gebiet, welches durch einen Polygonzug umschlossen ist. Dieses Gebiet darf dabei auch „Löcher“ und weitere, nicht im Polygonzug enthaltene Punkte beinhalten, die Teil der fertigen Triangulierung sind.

Das in den letzten Kapiteln erarbeitete Programm trianguliert dieses Gebiet derart, dass die folgenden bereits in Kapitel 1 vorgestellten Optimalitätskriterien erfüllt sind.

Anforderung 4.1 (Optimalität) Wir suchen eine Triangulierung T einer Punktemenge P unter einer Nebenbedingung N, so dass

(i) die Netzweite eine vorgegebene Länge von $h_{\max}$ nicht überschreitet und

(ii) der kleinste vorkommende Winkel mindestens eine vorgegebene Größe $\alpha_{\min}$ hat.

Wir haben während der Implementierung festgestellt, dass sich die zweite Bedingung nicht immer realisieren lässt. Es können durch die Nebenbedingung kleinere Winkel unvermeidbar sein. Außerdem kann der vorgegebene Winkel $\alpha_{\min}$ so groß gewählt worden sein, dass der Algorithmus zum Vergrößern der Winkel nicht terminiert.
Dies konnten wir mit Hilfe von Abbruchkriterien umgehen und dem Anwender des Programms statt dessen die Triangulierung ausgeben, die zumindest das erste Kriterium erfüllt.

Inhalt der CD

Auf der beigefügten CD finden sich die folgenden Dateien.

(i) Im Ordner *Ausarbeitung* finden sich diese Arbeit in PDF-Form sowie die verwendeten `.tex`-Dateien. Diese sind aus Gründen der Übersichtlichkeit kapitelweise in einzelnen Ordnern gespeichert. Teil der `.tex`-Dateien sind auch sämtliche verwendeten Abbildungen, die alle selbst angefertigt wurden.

(ii) Der Ordner *Implementierung* beinhaltet die Quellcodes der verwendeten Funktionen. In der Datei `execute.m` findet sich eine Beispielkonfiguration, mit der das Programm getestet werden kann. Einige Beispielpolygonzüge liegen im Ordner *data*.

(iii) Die beiden als PDF vorliegenden Quellen [dB2000] und [Sh1997] befinden sich im Ordner *Literaturquellen*. Die übrigen Quellen lagen in gedruckter Form vor.

Literaturverzeichnis

[dB2000] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational geometry: algorithms and applications, Springer Verlag, Berlin, 2000.

[Ed2001] H. Edelsbrunner, Geometry and Topology for Mesh Generation, Cambridge University Press, Cambridge, 2006.

[MG2005] S. Müller-Philipp, H.-J. Gorski, Leitfaden Geometrie, Vieweg Verlag, Wiesbaden, 2005.

[Sh1997] J. Shewchuk, Delaunay Refinement Mesh Generation, Technical Report CMU-CS-97-137, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1997.

Erklärung

Ich erkläre, dass ich die Arbeit selbständig angefertigt und nur die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinn nach anderen Werken, gegebenenfalls auch elektronischen Medien, entnommen sind, sind von mir durch Angabe der Quelle als Entlehnung kenntlich gemacht. Entlehnungen aus dem Internet sind durch Ausdruck belegt.

Ulm, den 29. März 2012

Anna Jostes