Computational Geometry

Ausarbeitung fr das Praktikum

Studienarbeit von
Maximilian Hempe
Roland Wilhelm

Dozent: Dr. Fischer
Hochschule Mnchen
Master Informatik
6. Juli 2013

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingsverzeichnis

1 Einleitung

Diese Studienarbeit beschreibt das Praktikum zur Vorlesung Computational Geometry. Die Studenten sollen darin den Umgang mit Mathematischen Problemen in Zusammenhang mit Computern oder Robotern lernen. Dafr sollen relativ einfache Probleme mit einer beliebigen Programmiersprache und selbst erstellten Algorithmen gelst werden. Im Praktikum werden nur die Aufgaben und Testdaten zur Verfgung gestellt, weitere Werkzeuge gibt es nicht.

Auf dieser Basis werden im Verlauf mehrere Aufgaben erarbeitet. Diese vertiefen die Themen der Vorlesung und gehen auf spezielle Sachverhalte intensiver ein. Ziel ist es meistens Schnittpunkte, ihre Anzahl und Flchen zu bestimmen. Das Praktikum zeigt, dass es beliebig komplex werden kann triviale Mathematische Probleme in performante Algorithmen zu konvertieren.

2 Aufgabe 1 - Anzahl von Schnittpunkten berechnen

In dieser Aufgabe soll eine Datei mit Liniensegmenten eingelesen werden und die Liniensegmente anschließend auf Schnittpunkte überprüft werden. Ein Liniensegment besitzt im Gegensatz zu einer Geraden einen Start- und einen Endpunkt. Diese Punkte sind je durch X- und Y-Koordinate definiert. Eine Zeile in der einzulesenden Datei enthält pro Zeile die beiden Koordinaten von Beginn und Ende des Segments. Ein Schnittpunkt ist dann vorhanden, wenn die Segmente mindestens einen gemeinsamen Punkt besitzen.

Der Algorithmus sollte zu Beginn möglichst einfach sein, deshalb wird jede Linie mit jeder anderen Linie auf einen Schnittpunkt getestet. Dadurch entsteht eine abstrakte Laufzeit von $O(n^2)$.

Um den Algorithmus testen zu können gab es drei verschiedene Files, die sich voralem in der Anzahl der Input-Datensätze unterscheiden. Eine Datei beinhaltet 1000 Segmenten, die Zweite 10.000 Segmente und die Dritte 100.000 Segmente. Die tatsächliche Laufzeit verlängert sich bei Verzehnfachung des Inputs um ca. das 100 Fache.

2.1 Einlesen der Daten

2.2 Ablauf des Algorithmus

Nachdem die Segmente eingelesen sind, wird die Funktion zur Berechnung der sich schneidenden Linien aufgerufen. Darin enthalten ist die Zeitmessung, diese wird direkt am Funktionsbeginn gestartet und vor der Rückgabe beendet.

Jede Linie muss immer nur ein Mal mit jeder anderen Linie auf mindestens einen gemeinsamen Punkt getestet werden. Deshalb wird der Test in einer verschachtelten Schleife so realisiert, dass die innere Schleife immer nur nachfolgende Linien abfrägt. Doppelte Abfragen, wie Linie 3 mit Linie 4 und Linie 4 mit Linie 3, werden somit verhindert.

```
1  for(unsigned int i = 0; i < m_lines.size(); i++) {
2    for(unsigned int j = i+1; j < m_lines.size(); j++) {
3
4      if(m_lines[i]->is_intersection(m_lines[j]) == true) {
5        m_intersected_lines_nr++;
6      }
7    }
8  }
```

Listing 1: Schleifenkostrukt zur Schnittpunktsuche

Ergibt der Test auf einen Schnittpunkt ein true, wird eine Membervariable inkrementiert. Der Algorithmus terminiert wenn das Array, das die Segmente beinhaltet, komplett geprüft wurde. Das Ergebnis ist die Anzahl der Schnittpunkte und wird durch die Membervariable festgehalten.

2.3 Test auf Schnittpunkte

Der Test ob es einen Schnittpunkt gibt erfolgt geschachtel und wird durch die Funktion ccw$Punkt, Punkt, Punkt$ unterstützt. Die Funktion bestimmt die Fläche, die durch die drei übergebenene Punkte aufgespannt wird. Je nach Vorzeichen der Fläche kann nun bestimmt werden, ob das Dreieck rechtsdrehend, linksdrehend oder flach ist. Flach bedeutet in diesem Fall, dass die drei Punkte auf einer Geraden liegen. Durch die Drehrichtung des Dreiecks kann bestimmt werden auf welcher Seite der Linie der dritte Punkt liegt.

```
1  int Line::ccw_max(Point &a_p, Point &a_q, Point &a_r){
2
3    //x und y koordinaten der Punkte
4    double result;
5
6    result = (a_p.get_y()*a_r.get_x())-(a_q.get_y()*a_r.get_x());
7    result += (a_q.get_x()*a_r.get_y())-(a_p.get_x()*a_r.get_y());
8    result += (a_p.get_x()*a_q.get_y())-(a_p.get_y()*a_q.get_x());
9
10   if (result < 0.0)
11     return -1;
12   else if(result == 0.0)
13     return 0;
14   else
15     return +1;
16 }
```

Listing 2: ccw-Funktion - test der Drehrichtung

2.3.1 Kolinearität

Zuerst werden die beiden Linien auf Kolinierität geprüft, das würde bedeuten, dass kein Dreieck, das aus den vier Punkten konstruiert wird eine Fläche aufspannt. Falls die Linie ein Punkt ist, wird nun vereinfacht geprüft, ob dieser Punkt auf der anderen Strecke liegt. Ist das der Fall, hat man bereits einen Schnittpunkt gefunden, falls nicht gibt es für diesen Fall keinen Schnittpunkt.

```
1 ...
2 if ( ccw_max(m_start, m_end, a_line->m_start) == 0 && ccw_max(m_start, m_end, a_line->m_end)
      == 0) {
3   if ( m_start == m_end ) {
4
```

```
5     //Punkt liegt auf Linie??
6     if( ( m_start > a_line->m_start && m_start < a_line->m_end )
7       || (m_start < a_line->m_start && m_start > a_line->m_end)){
8       return true;
9     }
10    else
11      return false;
12  ...
```

Listing 3: Schnittpunkttest von kolinearen Linien - Sonderfall Punkt

Ist die Linie regulär, hat sie also einen vom Startpunkt verschiedenen Endpunkt, muss ein Überlappungstest durchgeführt werden. Hierfür wird der Verktor aus Start- und Endpunkt um 90 Grad und -90 Grad gedreht, sodass zwei zusätzliche Punkte über der Linie entstehen. Nun werden neue Dreiecke erstellt, Dreieck 1 mit Startpunkt, Startpunkt der anderen Linie und dem generiertem neuen Punkt p und Dreieck 2 mit Endpunkt, dem Startpunkt der anderen Linie und dem anderen erzeugten Punkt q. Verlaufen die beiden Dreiecke in gleicher Richtung, überdecken sich die Segmente und es gibt einen Schnittpunkt. Falls die zwei Dreiecke nicht gleichdrehend sind, werden zwei neue Dreiecke mit dem Endpunkt der anderen Linie anstelle der Startpunktes erstellt und diese wiederum gestestet. Sind auch diese Dreiecke nicht gleichdrehend, gibt es keinen Schnittpunkt.

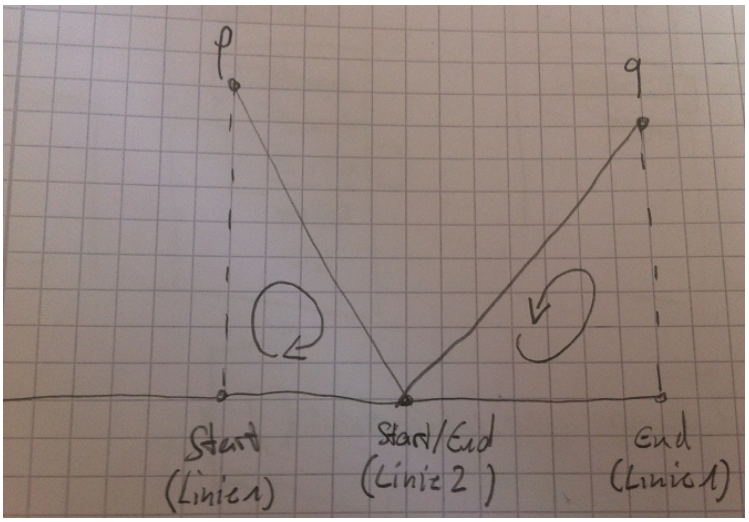

Abbildung 1: Struktur des Tests bei Kolinearität

```
1  ...
2  else { //überlappungstest (line <-> line oder line <-> punkt)
3        //p über m_start - drehung um -90°, q über m_end - drehung um 90° des gegengesetzten
           Vektor
4     Point p(m_end.get_y()-m_start.get_y(), m_end.get_x()-m_start.get_x()),
5           q(m_start.get_y()-m_end.get_y(), m_end.get_x()-m_start.get_x());
6
7     //Start-Punkt auf der Linie (inkl Ränder)
8     if( ccw_max(m_start, a_line->m_start, p)*ccw_max(m_end,q,a_line->m_start) >= 0 ){
```

```
9       return true;
10    }
11    //End-Punkt auf der Linie (inkl Ränder)
12    else if(ccw_max(m_start, a_line->m_end, p)*ccw_max(m_end,q,a_line->m_end) >= 0){
13      return true;
14    }
15    else
16      return false;
17  }
18 }
19 ...
```

Listing 4: Schnittpunkttest von kolinearen Linien

2.3.2 Normaler Schnittpunkt

Falls die beiden Strecken nicht auf der selben Gerade liegen, muss lediglich überprüft werden ob die beiden Punkte des einen Segments je auf verschiedenen Seiten der anderen Linie liegen. Dies kann wiederum mit der ccw-Funktion getestet werden. Die Betrachtung wird aus Sicht beider Linien gemacht, sonst würde eine seitlich versetzte Linie als Schnittpunkt gezählt.

```
1 ...
2 else if ( (ccw_max(m_start, m_end, a_line->m_start)*ccw_max(m_start, m_end, a_line->m_end)) <
        0
3       && ccw_max(a_line->m_start, a_line->m_end, m_start)*ccw_max(a_line->m_start, a_line->
            m_end, m_end) < 0 ) {
4
5   //->Schnittpunkt!!
6   return true;
7 }
8 return false;
```

Listing 5: Schnittprüfung mit ccw-Funktion

Sind beide Tests negativ, gibt es keinen Schnittpunkt. Es kann also ein sonst generell gültiger Else-Fall erstellt werden, der false zurück liefert.

3 Aufgabe 2

4 Aufgabe 3 - Line Sweep

Der Line Sweep Algorithmus ist ein Versuch die Schnittpunkt suche zu beschleunigen. Die Laufzeit wird nun outputsensitiv, also abhängig von der Anzahl der gefundenen Schnittpunkte. Es wird eine fiktive vertikale Linie erzeugt, die sich Punkt für Punkt durch die Linien arbeitet. Bei der Art wird zwischen Anfangs-, End- und Schnittpunkt unterschieden, je nach Art des Punktes werden eine andere Aktionen durchgeführt. Schnittpunkte können immer nur zwischen benachbarten Segmenten auftreten. Der Algorithmus terminiert, wenn alls Punkte bzw. Events bearbeitet wurden.

Es gibt beim Line Sweep allerdings einige Funktionseinschränkungen, die die Gleichzeitigkeit von Ereignissen betreffen. Diese beherrscht er nur mit sehr aufwendig zu implementierenden Mechanismen. Um die Aufgabe nicht unnötig komplexer zu gestalten werden diese nicht eingebaut und darauf geachtet folgende Regeln einzuhalten:

- X-Koordinaten von Schnitt- und Endpunkten sind paarweise verschieden
- Länge der Segmente >0
- nur echte Schnittpunkte
- keine Linien parallel zur Y-Achse
- keine Mehrfachschnittpunkte
- keine überlappenden Segmente

4.1 Initialisierung

Die Sweep Line besteht aus mehreren Queues, die den aktuellen und zukünftigen Zustand verwalten. Die erste Queue ist die Eventqueue, hier werden die Punkte abgelegt und mit der Information versehen, auch welcher Linie sie liegen. Bei Schnittpunkten werden die beiden Linien vermerkt, die sich dort schneiden.

```
1 class Event {
2 private:
3   MyEventtype m_type;
4   Point m_punkt;
5   Line* m_seg;
6   Line* m_seg2; //falls schnittpunkt
7 ...
```

Listing 6: Attribute der Klasse Event

In der zweiten Liste werden die Segmente verwaltet, die sich aktuell mit der Seep Line schneiden. An einem Startpunkt wird die damit verbundene Linie an der richtigen Stelle in diese Queue eingefügt und auf Schnittpunkte mit ihren Nachbarn überprüft. Erreicht die Sweep Line einen Endpunkt, wird die entsprechende Linie entfernt und die neuen Nachbarn auf einen Schnittpunkt getestet. Wird ein Schnittpunkt behandelt, müssen die beiden Segmente auf der dieser Schnittpunkt liegt die Positionen tauschen. Gefundene Schnittpunkte werden in die Output Liste und als neues Event in die Eventqueue eingefügt. Hier ist besonders darauf zu achten, dass die Schnittpunkte an der richtigen Position einsortiert werden.

```
class Sweep {
private:
  list<Event> eventqueue;
  list<Line> segmentqueue;
  Event *ereignis;
  list<Event> output;
...
```

Listing 7: Attribute der Klasse Sweep

Um die Eventqueue zu initialisieren wurde eine neue Funktion in die Klasse LineFile eingefügt. Diese fügt alls Punkte ein und definiert ob es sich um einen Start- oder Endpunkt handelt. Anschließend wird die Liste durch eine in der STL enthaltende Funktion nach der X-Koordinate sortiert.

```
void LineFile::sweepiniteventqueue(){
  for(unsigned int i = 0; i < m_lines.size(); i++) {
    m_sweep.addevent(m_lines[i]->getstart(),m_lines[i]);
    m_sweep.addevent(m_lines[i]->getend(), m_lines[i]);
  }

}
...
void Sweep::addevent(Point* a_point, Line* a_line){
  //a_point ist startpunkt
  if ( a_point == a_line->getstart() ){
    eventqueue.push_front( Event(a_point, a_line, STARTPUNKT) );
  }
  //a_point ist endpunkt
  else {
    eventqueue.push_front( Event(a_point, a_line, ENDPUNKT) );
  }
  print_eventqueue();
}
```

Listing 8: Initialisierung der Sweep Line

4.2 Behandlung von Events

Da die Eventqueue immer nach aufsteigenden X-Koordinaten sortiert sein muss, kann der aktuelle X-Wert der Sweep Line durch die X-Koordinate des ersten Elements in der Eventqueue bestimmt werden. Wird die Sweep gestartet, wählt sie das erste Event aus der Queue und führt je nach Art des Punktes die entsprechende Aktion aus. Anschließend wird das Event aus der Queue entfernt und wieder das erste ausgewählt. Dieser Vorgang wir solange wiederholt, bis keine Events mehr in der Eventqueue enthalten sind. Dann können die Schnittpunkte aus der Outputqueue ausgelesen werden. Da der Vorgang des Auswählens und Löschen von Events bei allen Punkten gleich ist, wird dieser im Folgenden nicht weiter erwähnt.

```
1  void Sweep::calcinters(){
2  bool test = false;
3  list<Event>::iterator neweve;
4
5    eventqueue.sort();
6    while( (test = eventqueue.empty()) == false) {
7      neweve = eventqueue.begin();
8      ereignis = &(*neweve);
9
10     switch(ereignis->gettype()){
11       case STARTPUNKT:
12         leftendpoint( );
13         break;
14
15       case ENDPUNKT:
16         rightendpoint( );
17         break;
18
19       case INTERSECTION:
20         treatintersection( );
21         break;
22
23       default:
24         exit(1);
25         break;
26     }
27     delevent(ereignis);
28   }
29 }
```

Listing 9: Schleife der Sweep Line

4.2.1 Startpunkte

Wie schon kurz beschrieben, wird bei der Behandlung eines Startpunktes das neue Segment in die Segmentqueue eingefügt. Es ist wichtig, dass es an der richtigen Stelle eingefügt wird, da nur zwischen Nachbarn ein Schnittpunkt liegen kann. Ist

die Nachbarschaft nicht korrekt sortiert, kommt es zu schwer nachvollziehbaren Fehlern.

```
Line* Sweep::addseg(Line* a_seg) {
  list<Line>::iterator newseg;

  if ( segmentqueue.empty() ) {
    segmentqueue.push_front(*a_seg);
    return a_seg;
  }
  else {
    for ( newseg = segmentqueue.begin(); newseg != segmentqueue.end(); ++newseg ) {
      if ( newseg->get_yvalue(getxposition()) > a_seg->get_yvalue(getxposition()) ) {
        segmentqueue.insert (newseg, *a_seg);
        return a_seg;
      }
    }
  }
  segmentqueue.insert (newseg, *a_seg);
  segmentqueue.unique();
  return a_seg;
}
```

Listing 10: Einfügen von Segmenten in die Sweep Line

Anschließend werden die beiden Nachbarn des gerade eingefügten Segments gesucht. Falls diese existieren, wird wie in Aufgabe 1 getestet ob ein Schnittpunkt zwischen dem Segment und seinem Nachbarn existiert. Existiert ein Schnittpunkt, werden erst anschließend die genauen Koordinaten bestimmt, dadurch soll die Laufzeit optimiert werden. Für den berechneten Schnittpunkt wird am Ende ein Event erzeugt, das sowohl in die Outputqueue als auch in die Eventqueue einsortiert wird. Dieser Vorgang wird anschließend für den zweiten Nachbarn wiederholt, soweit dieser existiert.

```
...
if ( neighbourA != NULL ) {
    if( aktseg->is_intersection_max(neighbourA) == true ) {
      interp1 = aktseg->intersectionpoint(neighbourA);
      Event Inter1(interp1,aktseg,neighbourA);
      addinter( Inter1 );
      addevent( Inter1 );
    }
  }
...
```

Listing 11: Schnittpunktprüfug bei Startpunkten

4.2.2 Endpunkte

Handelt es sich beim aktuell behandelten Event um einen Endpunkt, muss das entsprechende Segment aus der Queue entfernt werden und die anschließend neuen

Nachbarsegmente auf Schnittpunkte überprüft werden. Um diesen Vorgang durchführen zu können, muss man zuerst die Nachbarn der Linie bestimmen. Das suchen von Elementen wird immer mit der STL-Funktion find() durchgeführt. Nun kann das Segment entfernt werden. Existieren beide Nachbarn, kann versucht werden den Schnittpunkt zu bestimmen. Falls ein Schnittpunkt zwischen den Nachbarn besteht, wird dieser wie beim Startpunkt beschrieben in die Output- und Eventqueue eingefügt. Man muss nun allerdings darauf achten, dass man diesen Schnittpunkt möglicherweise schon berechnet hat. Es dürfen also nur Punkte in die Eventqueue eingefügt werden, deren X-Koordinate größer ist als die aktuelle Position der Sweep Line.

```
1  void Sweep::rightendpoint(){
2    Line *segA=NULL, *segB=NULL;
3    Point interp1;
4
5    segA = getneighbour_high(ereignis->get_line());
6    segB = getneighbour_low(ereignis->get_line());
7
8    delseg(ereignis->get_line());
9    if ( segA != NULL && segB != NULL) {
10     if ( segA->is_intersection_max(segB) == true ) {
11       interp1 = segA->intersectionpoint(segB);
12       Event Inter1(interp1, segB, segA);
13       if(Inter1.get_x() > getxposition()) {
14         addevent( Inter1 );
15       }
16       addinter( Inter1 );
17     }
18   }
19 }
```

Listing 12: Behandlung von Endpunkten

4.2.3 Schnittpunkte

Die Behandlung von Schnittpunkten klingt vorerst sehr einfach ist allerdings relativ komplex umzusetzten. Es müssen die Segmente, die sich am Schnittpunkt schneiden, in der Segmentqueue vertauscht werden und anschließend je mit ihrem neuen Nachbarn auf einen Schnittpunkt überprüft werden. Hierfür müssen zunächst beide Segmente in der Queue gefunden werden und man muss spezifizieren welche Line in der Queue einen früheren Platz belegt. Nun werden die beiden Nachbarn des Schnittpärchens bestimmt. Das Vertauschen der beiden Schnittsegmente übernimmt nun die STL-Funktion iter_swap$iterator, iterator$. Nun werden die beiden Segmente je mit ihrem neuen Nachbarsegment auf einen Schnittpunkt geprüft. Existiert ein Schnittpunkt, wird dieser, wie aus den anderen Events bekannt, als Event in Eventqueue und Outputqueue eingefügt. Auch hier gilt die Voraussetzung, dass nur Events

eingefügt werden dürfen, deren X-Koordinate größer ist als die aktuelle Position der Sweep Line.

```
void Sweep::treatintersection(){
  Line *segA=ereignis->get_line(), *segB=ereignis->get_line2(), *neighbourA=NULL, *neighbourB
      =NULL;
  Point interp1, interp2;
  list<Line>::iterator it_A = find(segmentqueue.begin(),segmentqueue.end(), *segA);
  list<Line>::iterator it_B = find(segmentqueue.begin(),segmentqueue.end(),*segB);

  if ( *it_B == *(++it_A) ){
    neighbourA = getneighbour_high(segB);
    neighbourB = getneighbour_low(segA);
    //Swap Elements
    //segB vor segA einordnen und altes Element segB löschen
    iter_swap(--it_A, it_B);

  }
  else if ( *it_B == *(--(--it_A)) ) { //iterator wurde in if() verändert
    neighbourA = getneighbour_low(segB);
    neighbourB = getneighbour_high(segA);
    iter_swap(++it_A, it_B);

  }

  if ( neighbourA != NULL ) {
      if ( segA->is_intersection_max(neighbourA) == true ) {
        interp1 = segA->intersectionpoint(neighbourA);
        Event Inter1(interp1, neighbourA, segA);
        if(Inter1.get_x() > getxposition()) {
          addevent( Inter1 );
        }
        addinter( Inter1 );
      }
    }
  if ( neighbourB != NULL ) {
      if ( segB->is_intersection_max(neighbourB) == true ) {
        interp2 = segB->intersectionpoint(neighbourB);
        Event Inter2(interp2, neighbourB, segB);
        if(Inter2.get_x() > getxposition()) {
          addevent( Inter2 );
        }
        addinter( Inter2 );
      }
    }
}
```

Listing 13: Behandlung von Schnittpunkten

5 Maximaler Kreis in einer konvexen Hlle

In dieser Aufgabe 4 werden fr zwei vorgegebene konvexe Polygone der grtmgliche einbeschreibbare Kreis mit Linearer Programmierung berechnet. Die vorgegebenen Polygone sind in den Dateien *Polygon.txt* und *testpolygon.txt* abgespeichert. Nachdem das Problem beschrieben wurde, wird auf Basis der Testdatei *testpolygon.txt* ein lineares Programm definiert und hergeleitet. Dieses lineare Programm wird schlussendlich auf das Polygon in der Datei *Polygon.txt* angewendet und das Ergebnis dargestellt.

5.1 Herleitung Problem

Gegeben ist ein konvexes Polygon $P \subseteq \mathbb{R}^2$. Bei dem Problem in dieser Aufgabe suchen wir nach dem grtmglichen Kreis $K \subseteq \mathbb{R}^2$, der vollstndig in P enthalten ist; das heit, es gilt $K \subseteq P$ wobei der Radius von K maximal ist. Damit der Kreis K mit Mittelpunkt $m := (m_x, m_y)$ und Radius r komplett in P enthalten ist, mssen folgende Bedingungen erfllt sein:

- Der Punkt m hat mindestens den Abstand r von allen Strecken.
- Der Punkt m liegt innerhalb des konvexen Polygons P.

Um den Abstand r des Mittelpunktes m von einer Strecke P_i zu berechnen, wird die folgende Formel genutzt.

$$r = N_{0i} * [m - P_i]$$

Dabei beinhaltet die Matrix N_{0i} in der Zeile i den jeweiligen Normaleneinheitsvektor fr die Strecke P_i.

Der Kreis K liegt also genau dann vollstndig in P, wenn folgende Ungleichung fr alle Strecken erfllt ist.

$$N_{0i} * [m - P_i] \geq r, i = 1, ..., n$$

5.2 Lsung durch lineare Programmierung

Das Ziel ist es den grten Kreis zu finden. Wir definieren ein lineares Programm mit den Variablen m_x, m_y und r, indem die Variable r unter der Nebenbedingung

$$N_{0i} * [m - P_i] \geq r, i = 1, ..., n$$

maximiert wird. Die entsprechende Zielfunktion wird definiert als

$$f := \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Zur Lsung des lineare Problems wird das Programm **MATLAB** (MATrix LABoratory) verwendet. Nachfolgend werden nur die wichtigsten Programmteile dargestellt und erklrt, dass komplette Programm **incircle.m** kann in der mitgelieferten Datei unter **matlab/incircle.m** begutachtet und getestet werden. Die jeweiligen konvexen Polygone werden durch das Programm als Matrizen in den **Workspace** importiert und stehen anschlieend im **Command Window** zur Verfgung. Die Matrizen fr die konvexen Polygonen P haben dabei folgende Strukturen:

$$P = \begin{bmatrix} 0 & 0 \\ 10 & 0 \\ 10 & 10 \\ 0 & 10 \end{bmatrix}$$

Die jeweiligen Zeilen der Matrix bilden den Startpunkt einer Strecke ab. Die Anzahl der Zeilen ist zugleich die Anzahl der zur Verfgung stehenden Strecken indem der Kreis eingebettet werden soll. Die Spalte 1 gibt die x-Werte und die Spalte 2 die y-Werte eines Startpunktes wieder. Um aus den einzelnen Punkten ein konvexes Polygon zu erhalten, werden die Startpunkte miteinander verbunden; also der Punkt in der Zeile wird mit dem Punkt in der Zeile 2 verbunden u. s. w. , der letzte Punkt (hier Zeile 4) wird mit dem Punkt in der Zeile 1 verbunden.

Das Listing 4 zeigt die Erstellung einer konvexen Hlle. Durch den MATLAB Befehl `convhulln(xy)` wird aus den vorgegebenen Strecken (xy) eine konvexe Hlle erzeugt und die entsprechenden Indizes zurckgegeben. Dadurch knnen dann in Zeilen 2 und 3 die jeweiligen Start- und Endpunkte einer Strecke bestimmt werden.

```
1  ecken = convhulln(xy);
2  A = xy(ecken(:,1),:);
3  B = xy(ecken(:,2),:);
```

Listing 14: Erstellen einer konvexen Hlle mit MATLAB

Nachdem die Start- und Endpunkte fr alle verfgbaren Strecken berechnet worden sind, werden fr diese in den Zeilen 1 - 3 der dazugehrige Normaleneinheitsvektor berechnet. Um sicherzustellen, dass alle Normaleneinheitsvektoren zum Mittelpunkt m zeigen, findet in den Zeilen 3 - 6 eine berprfung statt. Hier werden aus allen Startpunkten der x- und y-Mittelwert berechnet und als sicherer Mittelpunkt M_0

abgespeichert. Ist bei der anschlieenden Subtraktion ein negativer Normaleneinheitsvektor vorhanden, wird dieser um 180 Grad gedreht.

```
1  N_p = (B - A) * [0 1; -1 0];
2  Betrag = sqrt(sum(N_p.^2,2));
3  N_p = N_p./[Betrag, Betrag];
4  M_0 = mean(A,1);
5  index = sum(N_p.*bsxfun(@minus, M_0, A), 2) < 0;
6  N_p(index,:) = -N_p(index, :);
```

Listing 15: Berechnung des Normaleneinheitsvektor mit MATLAB

Nachdem alle erforderlichen Daten berechnet worden sind, wird nun unter Einhaltung der Nebenbedingung, dass Ungleichungssystem aufgestellt. Mit dem MATLAB Befehl **linprog** kann ein Ungleichungssystem in der Form

$$A * x \leq b$$

gelst werden. Durch Umformung der Nebenbedingung

$$N_{0i} * [m - P_i] \geq r$$

erhlt man das angepasste Ungleichungssystem

$$-N_0 * m + r \leq -N_0 * P$$

fr das Programm MATLAB. Das Listing 5 berechnet in den Zeilen 1 und 2 jeweils die linke Seite sowie rechte Seite. In den Zeilen 3 und 4 wird die dazugehrige Zielfunktion definiert.

```
1  b = -sum(N_p.*A, 2);
2  A = [-N_p.*ones(strecken_nr, 2), ones(strecken_nr, 1)];
3  f = zeros(3, 1);
4  f(3) = -1;
```

Listing 16: Erstellung der Ungleichung mit MATLAB

Die erzeugten Parameter werden in der ersten Zeile dem Befehl *linprog* bergeben, dieses liefert uns dann in den entsprechenden Rckgabewerten die Lsung des linearen Problems zurck.

```
1  [result, fval, exitflag, output] = linprog(f, A, b);
2  C = result(1:2)';
3  R = result(3);
```

Listing 17: Starten der Berechnung mit MATLAB

5.3 Ergebnisse

Im Abschnitt 5.2 wurde das entwickelte MATLAB-Programm schrittweise vorgestellt und erklrt. Hier werden nun die entsprechenden Lsungen fr die Dateien *Polygon.txt* und *testpolygon.txt* vorgestellt.

Wird das Polygon in der Datei *testpolygon.txt* dem MATLAB-Skript *incircle* bergeben, werden die Werte $m = (5.0, 5.0)$ fr den Mittelpunkt und $r = 5.0$ fr den Radius des Kreises berechnet. Die Abbildung 2 bildet das Ergebnis grafisch ab.

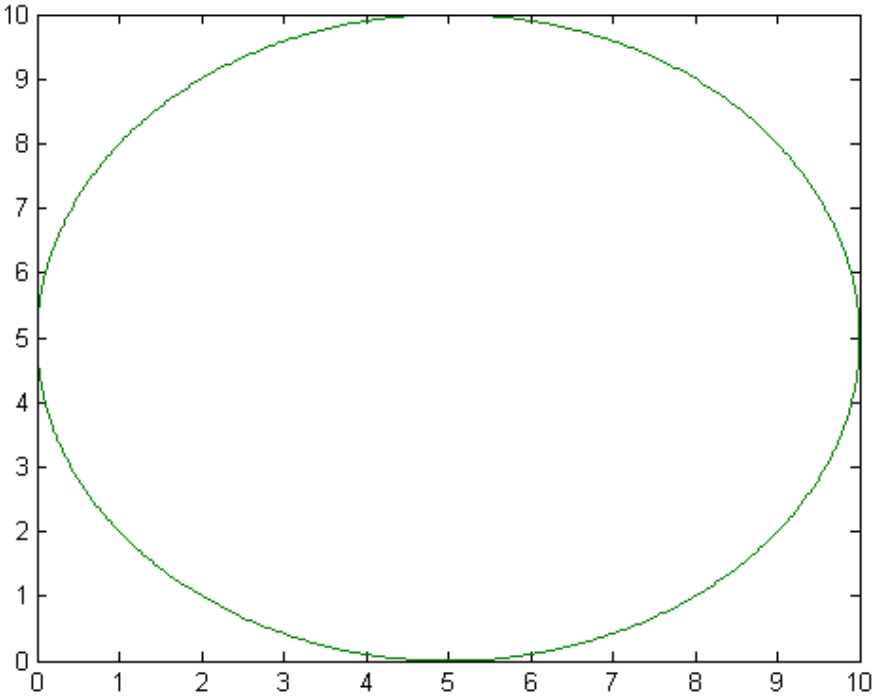

Abbildung 2: Testpolygon mit Kreis

Mit der Datei *testpolygon.txt* werden die Werte $m = (472.5705, 476.6642)$ fr den Mittelpunkt und $r = 438.5922$ fr den Radius des Kreises berechnet. Die entsprechende Grafik wird in der Abbildung 3 abgebildet. Das konvexe Polygon wird blau und der grtmgliche einbeschreibbare Kreis grn dargestellt.

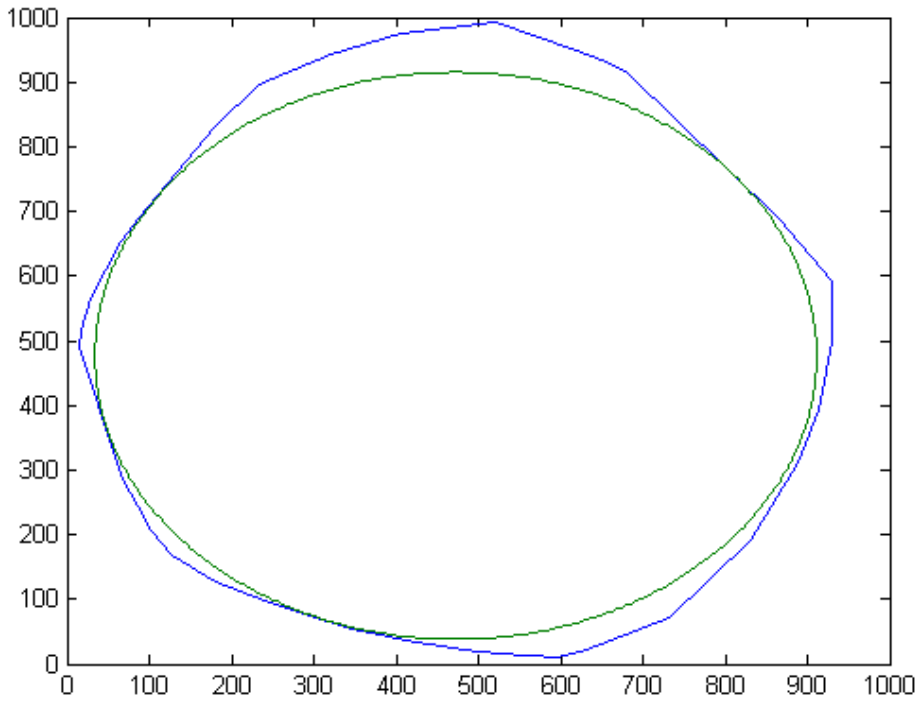

Abbildung 3: Polygon mit Kreis

6 Aufgabe 5

7 Fazit

7.1 Zusammenfassung

7.2 Lessons Learned

Literatur