Hochschule München
Fakultät für Informatik und Mathematik

Praktikumsaufgabe 1

in der Vorlesung

Computational Geometry

Bestimmung von Schnittpunkten aus einem gegebenen Satz von Strecken

Team:	Christopher Hinz, Tobias Gruber
Studiengruppe:	Master Informatik
Studiensemester:	1. Semester
Schwerpunkt:	Embedded Computing

11.04.2022

Sommersemester 2022

Inhaltsverzeichnis

1 Problemstellung

In dem Tar-File 'strecken.tgz' (s.u.) befinden sich Dateien mit jeweils 4 Koordinaten pro Zeile. Diese stellen jeweils die x- und y-Koordinaten eines Start- bzw. Endpunkts einer Strecke dar. Die Datei soll eingelesen und die Anzahl der sich schneidenden Strecken durch paarweises Testen herausgefunden werden. Zusätzlich wird die pro Datei aufgewendete Zeit gemessen. Es ist zu begründen, warum die Anzahl der vom Programm jeweils gefundenen Schnittpunkte korrekt ist.

2 Einführung

Im Rahmen der Vorlesung Computational Geometry lautet die Aufgabe für das erste Praktikum, die Anzahl der Schnittpunkte aus einem Satz von Strecken zu ermitteln. Dafür werden 3 Dateien zur Verfügung gestellt, die jeweils Datensätze zu 1001, 10.001 und 100.001 Strecken enthalten.
Die Dateien enthalten jeweils 4 Koordinaten pro Zeile, welche jeweils die x- und y-Koordinaten eines Start- bzw. Endpunkts einer Strecke darstellen. Die Dateien sollen jeweils eingelesen und die Anzahl der sich schneidenden Strecken durch paarweises Testen ermittelt werden. Zusätzlich wird die aufgewendete Zeit zur Ermittlung der Anzahl der Schnittpunkte für jede Datei gemessen.
Im folgenden Abschnitt wird darauf eingegangen, wie die Grundstruktur der Problemlösung aufgebaut ist und wie die mathematische Strategie zur Lösung des Problems in C++ formuliert werden kann. Anschließend werden die Ergebnisse der Datenauswertung vorgestellt Es ist zu begründen, warum die Anzahl der vom Programm jeweils gefundenen Schnittpunkte korrekt ist.

3 Methoden

3.1 Abschätzung der zu erwartenden Größenordnung des Ergebnisses

Um einschätzen zu können, inwiefern die Ergebnisse sinnvoll sind, sollen an dieser Stelle einige Vorüberlegungen getroffen werden.

Welche Anzahl an Schnitten sind zu erwarten bzw. in welchem Wertebereich sollten sich die Ergebnisse bewegen?
Die minimal zu erwartende Anzahl an Schnitten wäre für 1.000, 10.0000 und 100.000 Strecken jeweils null. Dies ist somit die untere Grenze des zu erwartenden Wertebereichs. Die maximale Anzahl an Schnitten würden in dem Fall auftreten, dass sich alle Strecken gegenseitig schneiden. Dann wäre nämlich für jede Strecke ein Schnittpunkt mit allen anderen Strecken gegeben. Das Problem ist in der Mathematik auch als Sektglasproblem bekannt. Die Berechnung erfolgt in Anlehnung an die Formel

für die Dreieckszahlen. Für eine Anzahl von n Strecken ergeben sich somit k Schnittpunkte wie folgt:

$$k = \frac{(n-1) \cdot n}{2} \tag{1}$$

Damit wären für 1.000 Strecken maximal 499.500 (für 10.000 maximal 49.995.000, für 100.000 maximal 4.999.950.000) zu erwarten. Dies ist somit die obere Grenze des zu erwartenden Wertebereichs.

3.2 Strategie zur Problemlösung

3.3 Verifizierung des Codes

4 Ergebnisse

4.1 Ergebniss der Verifizierung

4.2 Ergebnisse der Schnittstellenberechnung und der Zeitmessung

Das Programm verwendet die Funktion `read_dat()` um die Daten aus den Dateien einzulesen und über `pack_koords()` die Datensätze zu Strecken zusammenzufassen. Jede Strecke wird anschließend gegen alle anderen Strecken geprüft, ob sie diese schneidet, oder nicht. Zur Feststellung, ob sich die zwei Strecken schneiden wird ausgewertet, ob die Punkte p_1, p_2 der ersten Strecke zu den Punkten p_1, p_2 der zweiten Strecke im oder gegen den Uhrzeigersinn angeordnet sind. Dafür wird folgende Formel verwendet, die im Ergebnis den Betrag des Kreuzproduktes aus den Richtungsvektoren der drei Punkte enthält:

$$ccw(p,q,r) := \begin{vmatrix} p_1 & p_2 & 1 \\ q_1 & q_2 & 1 \\ r_1 & r_2 & 1 \end{vmatrix} = (p_1 q_2 - p_2 q_1) + (q_1 r_2 - q_2 r_1) + (p_2 r_1 - p_1 r_2) \tag{2}$$

Ist das Produkt der ccw-Funktion der ersten Strecke mit den Endpunkten der zweiten Strecke, sowie das Produkt der ccw-Funktion der zweiten Strecke mit den Endpunkten der ersten Strecke kleiner oder gleich 0, so kann unter der Annahme, dass die Strecken nicht kollinear zueinander sind davon ausgegangen werden, dass sich die Strecken schneiden.
Sind die Strecken Kollinear angeordnet, so ergeben beide Kreuzprodukte 0. In diesem Fall muss noch überprüft werden, ob sich die Strecken überlappen. Bei Überlappung der Strecken lässt sich mindestens einer der Endpunkte der zweiten Strecke über ein Vielfaches zwischen 0 und 1 des ersten Streckenvektors addiert mit einem Endpunkt der ersten Strecke darstellen. Da zuvor bereits festgestellt wurde, dass die Strecken kollinear sind, kann der Skalar zum Richtungsvektor der zweiten Strecke über das Auflösen der parametrischen Darstellung durch die x-Komponenten der Ortsvektoren aufgelöst werden.

$$\lambda = \frac{x_r - x_p}{x_q - x_p} \tag{3}$$

Liegt der Wert von λ zwischen 0 und 1, so überlappen sich die Funktionen. Nimmt λ den Wert 0 oder 1 an, schneiden sich die Strecken in jeweils einem der Endpunkte.

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>


struct point{
    double x;
    double y;
};

struct line{
    point p1;
    point p2;
};

void read_dat(char* filename, std::vector<line>& target){
    line temp;
    std::ifstream file;
    file.open(filename);
    double k1, k2, k3, k4;
    if(!file.is_open()){
        std::cout << "Could not open file\n";
    }
    while(file >> k1 >> k2 >> k3 >> k4){
        temp.p1.x = k1;
        temp.p1.y = k2;
        temp.p2.x = k3;
        temp.p2.y = k4;
        target.push_back(temp);
    }
    file.close();
}

double ccw(point p, point q, point r){
    double res = (p.x*q.y - p.y*q.x)+ (q.x*r.y - q.y*r.x) + (p.y*r.x  - p.
   x*r.y);
    return res;
}

bool line_intersect_check(line l1, line l2){
    bool retval = false;
    double ccw_res1 = ccw(l1.p1, l1.p2, l2.p1) * ccw(l1.p1, l1.p2, l2.p2);
    double ccw_res2 = ccw(l2.p1, l2.p2, l1.p1) * ccw(l2.p1, l2.p2, l1.p2);
    if(ccw_res1 <= 0.0 && ccw_res2 <= 0.0){
        retval = true;
        if(ccw_res1 == 0.0 && ccw_res2 == 0.0){
            double lambda1 = (l2.p1.x-l1.p1.x)/(l1.p2.x-l1.p1.x);
            double lambda2 = (l2.p2.x-l1.p1.x)/(l1.p2.x-l1.p1.x);
            if ((lambda1 < 0.0 || lambda1 > 1.0) && (lambda2 < 0.0 ||
```

```
    lambda2 > 1.0))
50                  retval = false;
51          }
52      }
53      return retval;
54  }
```

Listing 1: p1_lib.h: Bibliotheksfunktionen

```
1  #include "p1_lib.h"
2  #include <chrono>
3
4  int main(){
5      auto start = std::chrono::steady_clock::now();
6
7
8      std::vector<line> lines_vec;
9      read_dat((char*)"../strecken/s_1000_1.dat", lines_vec);
10
11
12     int intersect_counter = 0;
13     for(unsigned int i = 0; i < lines_vec.size(); ++i){
14         for(unsigned int j = i; j < lines_vec.size(); ++j){
15             if(i!=j){
16                 if(line_intersect_check(lines_vec[i], lines_vec[j])){
17                     ++intersect_counter;
18                 }
19             }
20         }
21     }
22     std::cout << "Strecken insgesamt: " << lines_vec.size() << "\n" << "
       Schnitte zweier Strecken: " << intersect_counter << "\n";
23
24     auto end = std::chrono::steady_clock::now();
25     std::cout << "Runtime: "
26     << (double)std::chrono::duration_cast<std::chrono::microseconds>(end -
       start).count()/1000
27         + std::chrono::duration_cast<std::chrono::milliseconds>(end -
       start).count() << " ms\n";
28
29     return 0;
30 }
```

Listing 2: strecken.cpp: Aufgruf der Bibliotheksfunktionen

In Listing 3 wird die Ausgabe des Programmes zur Schnittpunktezählung aufgeführt. In der Ausgabe ist zu erkennen, dass die Anzahl der Schnitte für alle Fälle (1.000, 10.000, 100.000) im zu erwartenden Wertebereich liegt. Schnittpunkte:

- 1.000 Strecken: 0 <= 11 <= 499.500
- 10.000 Strecken: 0 <= 732 <= 49.995.000
- 100.000 Strecken: 0 <= 77.126 <= 4.999.950.000

```
# s_1000_1.dat
Strecken insgesamt: 1001
Schnitte zweier Strecken: 11
Runtime: 40.965 ms
#s_10000_1.dat
Strecken insgesamt: 10001
Schnitte zweier Strecken: 732
Runtime: 3984.3 ms
#s_100000_1.dat
Strecken insgesamt: 100001
Schnitte zweier Strecken: 77126
Runtime: 394721 ms
```

Listing 3: strecken.cpp: Ausgabe Konsole

4.3 Programme zum Testen der Funktion

Um die korrekte Funktionsweise des Programmes zu verifizieren werden verschiedene Testfälle definiert. Die Testfälle decken unterschiedliche Streckenanordnungen ab und sollen dadurch die in der Vorlesung besprochenen Grenzfälle für Streckenschnitte überprüfen. Abbildung 1 zeigt die implementierten Testfälle.

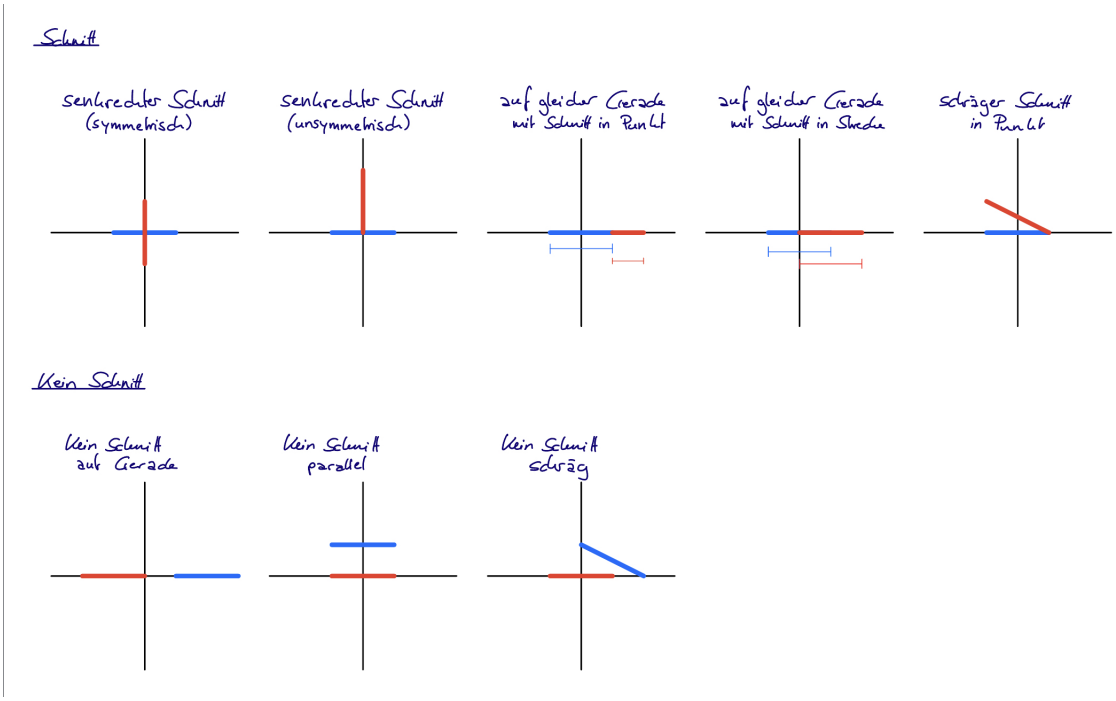

Abbildung 1: Testfälle für Streckenschnitte

Listing 4 zeigt die Implementierung der Testfälle bzw. den Aufruf der entsprechenden Bibliotheksfunktionen in C++.

```cpp
#include "p1_lib.h"

void print_result(line l1, line l2, bool b_soll){
    std::cout << std::boolalpha << "Soll: " << b_soll << ", Ist: " << line_intersect_check(l1, l2) << "\n";
}


int main(){

    // Verschiedene Schnitte

    // Strecken mit senkrechtem Schnitt (symmetrisch)
    // p1 = (0,1) nach p2 = (0,-1)
    // p1 = (1,0) nach p2 = (-1,0
    print_result(line{.p1.x = 0, .p1.y = 1, .p2.x = 0, .p2.y = -1}, line{.p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);

    // Strecken mit senkrechtem Schnitt (unsymmetrisch)
    // p1 = (0,2) nach p2 = (0,0)
    // p1 = (1,0) nach p2 = (-1,0)
    print_result(line{.p1.x = 0, .p1.y = 2, .p2.x = 0, .p2.y = 0}, line{.p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);

    // Strecken auf gleicher Gerade mit Schnitt in einem Punkt
    // p1 = (1,0) nach p2 = (2,0)
    // p1 = (1,0) nach p2 = (-1,0)
    print_result(line{.p1.x = 1, .p1.y = 0, .p2.x = 2, .p2.y = 0}, line{.p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);

    // Strecken auf gleicher Gerade mit Schnitt in einer Strecke
    // p1 = (0,0) nach p2 = (2,0)
    // p1 = (1,0) nach p2 = (-1,0)
    print_result(line{.p1.x = 0, .p1.y = 0, .p2.x = 2, .p2.y = 0}, line{.p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);

    // Strecken auf gleicher Gerade mit Schnitt in einer Strecke
    // p1 = (0,0) nach p2 = (2,0)
    // p1 = (1,0) nach p2 = (-1,0)
    print_result(line{.p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 1}, line{.p1.x = 1, .p1.y = 0, .p2.x = -1, .p2.y = 0}, true);


    // Verschiedene ohne Schnitt

    // Strecken ohne Schnitt auf Gerade
    // p1 = (-2,0) nach p2 = (0,0)
    // p1 = (1,0) nach p2 = (3,0)
    print_result(line{.p1.x = -2, .p1.y = 0, .p2.x = 0, .p2.y = 0}, line{.p1.x = 1, .p1.y = 0, .p2.x = 3, .p2.y = 0}, false);

    // Strecken ohne Schnitt parallel
```

```
46      // p1 = (-2,0) nach p2 = (0,0)
47      // p1 = (-2,1) nach p2 = (0,2)
48      print_result(line{.p1.x = -2, .p1.y = 0, .p2.x = 0, .p2.y = 0}, line{.
        p1.x = -2, .p1.y = 1, .p2.x = 0, .p2.y = 2}, false);
49
50      // Strecken ohne Schnitt schraeg
51      // p1 = (-2,0) nach p2 = (0,0)
52      // p1 = (0,1) nach p2 = (2,0)
53      print_result(line{.p1.x = -2, .p1.y = 0, .p2.x = 0, .p2.y = 0}, line{.
        p1.x = 0, .p1.y = 1, .p2.x = 2, .p2.y = 0}, false);
54
55      return 0;
56  }
```

Listing 4: testing.cpp: Testen der Bibliotheksfunktionen

Listing 5 zeigt, dass für die Testfälle die erwarteten Ergebnisse erzielt werden. Das bedeutet, dass für alle Streckenpaare, die wie in den Testfällen angeordnet sind, die Schnittstellenfunktion korrekt berechnet ob sie sich schneiden oder nicht.

```
Soll: true, Ist: true
Soll: true, Ist: true
Soll: true, Ist: true
Soll: true, Ist: true
Soll: true, Ist: true
Soll: false, Ist: false
Soll: false, Ist: false
Soll: false, Ist: false
```

Listing 5: testing.cpp: Ausgabe Konsole

Des weiteren wurde ein Python-Programm geschrieben. Dieses plottet die Strecken mit Schnittpunkten und stellt damit eine weitere Möglichkeit dar die Ergebnisse zu überprüfen. Dies Graphen werden mittels matplotlib erstellt und sehen wie nachfolgend dargestellt aus. Auf Implementierungsdetails soll an dieser Stelle verzichtet werden, da dieses Programm ausschließlich zur Überprüfung dient.

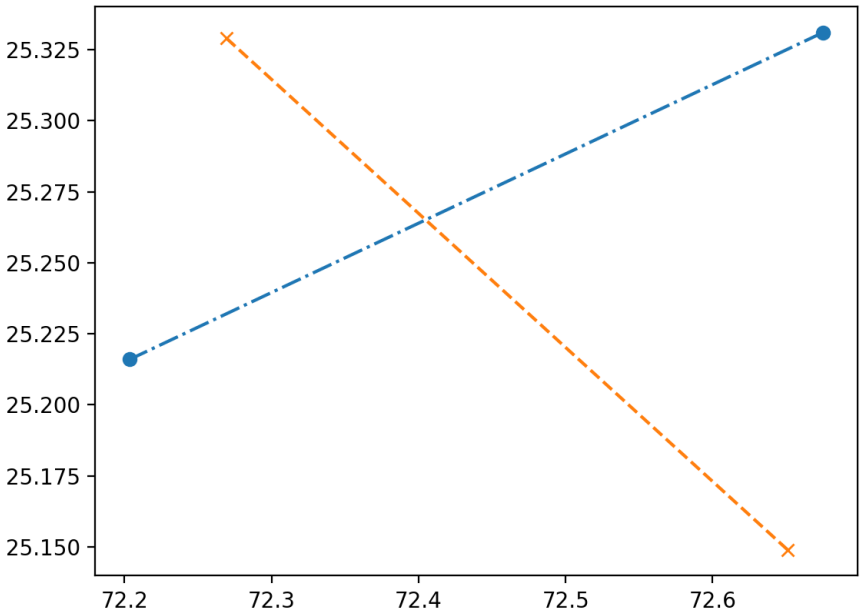

Abbildung 2: Python-Plot mit matplotlib