BACHELORARBEIT

Zur Verwendung von Python als Programmiersprache in einem einführenden Kurs in die algorithmische Geometrie am Beispiel des Problems der Vereinfachung von Polygonzügen

Name: Mandy Katharina Behr [23214]

Abgabedatum: 26.11.2020

Erstprüfer: Prof. Dr. Andreas Spillner

Zweitprüfer: Prof. Dr. Dr. Michael Schenke

HoME
HOCHSCHULE
MERSEBURG
University of
Applied Sciences

Abstract

Es soll ein Programm in Python geschrieben werden, das in dem Modul „Algorithmische Geometrie“ zur Anwendung kommen soll. Auch für Studierende ohne Erfahrung in dieser Programmiersprache soll das Programm es ermöglichen, diese Aufgaben zu bearbeiten und innerhalb von etwa 10 Wochen wöchentlich einen Baustein zur Abgabe im Praktikum parat zu haben. In den Bausteinen wird die Vereinfachung von Polygonzügen betrachtet. Mit Hilfe eines dazu passenden Algorithmus wird diese Aufgabe bewältigt und am Beispiel der Vereinfachung von GPS-Tracks zur Anwendung gebracht.

Aufgabenstellung

Die Aufgabe dieser Bachelorarbeit besteht darin für Studierende, die das Modul algorithmische Geometrie belegen und noch keine Erfahrung mit Python haben, schrittweise Bausteine für ein Programm zu entwickeln, sodass am Ende die Vereinfachung von Polygonzügen z.B. für Daten von OpenStreetMap durchführbar wird. Die Bausteine sollten so gestaltet sein, dass sich Teilnehmer des Moduls in einem Zeitraum von etwa 10 Wochen jede Woche einen Baustein erarbeiten. Des Weiteren umfasst die Aufgabenstellung folgende Punkte:

1. Aufgabe der Vereinfachung von Polygonzügen und Anwendungsbeispiele dafür

2. Beispiele für Kennzahlen, wie gut eine Vereinfachung ist

3. Verfahren zur Vereinfachung von Polygonen - Überblick

4. Schrittweiser Aufbau eines Programms zur Vereinfachung von Polygonzügen (als Hilfsmittel in der Lehre)

4.1 Grundbausteine (Polygonzug, Kennzahlen, Algorithmen)

4.2 Schaffung der Möglichkeit Polygone aus OpenStreetMap (OSM) zu laden und zu vereinfachen

5. Experimente zu Algorithmen und Kennzahlen an ausgewählten Daten aus OSM

Inhaltsverzeichnis

Vorwort zur verwendeten Sprachform

In der vorliegenden Bachelorarbeit werden die üblichen maskulinen Ausdrücke, in Fällen wie Nutzer oder Teilnehmer, verwendet. Dies soll jedoch keine Benachteiligung anderer Geschlechter darstellen, sondern als sprachliche Vereinfachung in geschlechtsneutraler Form gelesen werden.

1 Einleitung

In den letzten Jahren hat sich Python bei Programmierern als favorisierte Programmiersprache etabliert. Sie zeichnet sich durch eine verständliche und übersichtliche Art aus, die sie auch bei Anfängern beliebt macht. Denn in dieser Programmiersprache wird beispielsweise auf Semikolons verzichtet und die Zugehörigkeit in Schleifen oder Funktionen wird durch das Einrücken der Zeile(n) visualisiert. Solche Eigenschaften helfen Flüchtigkeitsfehler zu minimieren oder die Zeit des Debuggens zu verringern.

Im Rahmen des Moduls zur algorithmischen Geometrie sind typischerweise Programmieraufgaben zu lösen. Da die verwendeten Verfahren sehr komplex sind, viele Sonderfälle betrachtet werden müssen und umfangreiche Datenstrukturen Verwendung finden, können die Aufgaben des Praktikums auf Studierende ohne ausreichend Kenntnisse in der verwendeten Programmiersprache, mit einem sehr hohen Einarbeitungsaufwand verbunden sein. Die Aufgabe dieser Arbeit besteht darin, für Studierende, die noch keine Erfahrung mit Python haben, einen guten Einstieg in diese Programmiersprache zu bieten. Dafür ist es notwendig ein Programm zu entwickeln, das sich auf ein algorithmisches Problem bezieht, das recht anschaulich ist. Aus diesem Grund betrachte ich die Vereinfachung von Polygonzügen, da sie wenig Vorwissen aus der algorithmischen Geometrie erfordert.

Bei einem Polygonzug handelt es sich um eine Reihe von Punkten, in der jeder Punkt mit dem Nachfolgenden durch eine Strecke verbunden ist. In der Abbildung 1 ist ein solcher Polygonzug zu sehen. Er wurde durch mein Programm erstellt und ausgeplottet. Einen solchen zu vereinfachen bedeutet, unnötige Abweichungen zu entfernen, aber dabei die allgemeine Form beizubehalten. Verwendung findet diese Vereinfachung in den Geoinformationssystemen, kurz GIS. GPS-Tracks oder Karten wenden diese an, da nicht jede minimale Krümmung des Weges für die Wegbeschreibung notwendig ist.

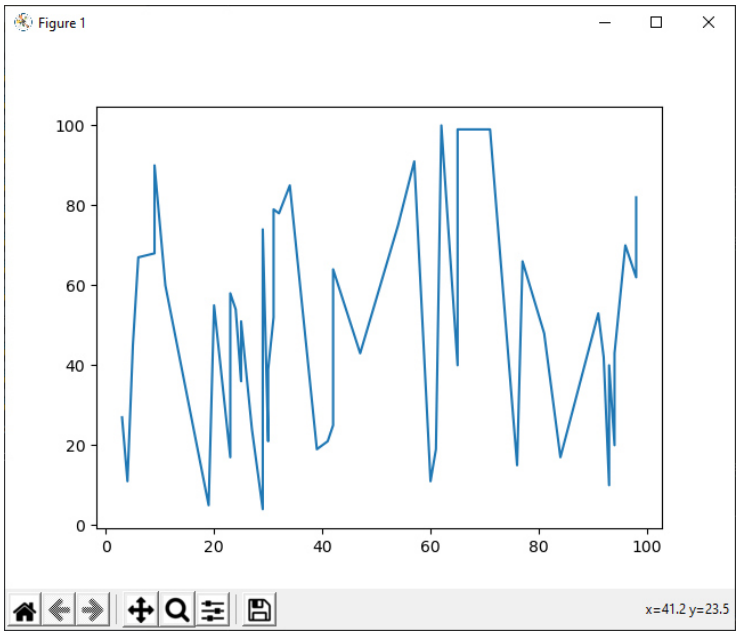

Abbildung 1: Polygonzug

Die Implementierung eines Verfahrens zur Vereinfachung eines solchen Polygonzugs ist für ein Praktikum nicht innerhalb einer Woche zu realisieren. Damit ein solches Programm auch für Studierende mit geringer Erfahrung in Python entwickelbar ist, soll jede Woche ein gewisser Teil dieser Gesamtaufgabe bearbeitet werden. Die in den Teilaufgaben bearbeiteten Programmteile werden in dieser Arbeit „Bausteine“ genannt. Um ein Konzept für die Aufgabenteilung zu schaffen, ist es nötig, zu Beginn den passenden Algorithmus zu finden, der angemessen leicht zu verstehen ist und sich gut in einem Programm umsetzen lässt.

2 Algorithmen und Kennzahlen

Es existiert eine große Anzahl an Algorithmen für die Vereinfachung von Polygonzügen. Um den passenden Algorithmus zu finden, benötige ich einige Voraussetzungen. Der ausgewählte Algorithmus sollte nicht zu komplex sein und zusätzlich in Praktikumsaufgaben von etwa 10 Wochen aufgeteilt werden können. Aber was genau wird eigentlich mit diesen Algorithmen vereinfacht? Was zeichnet Polygonzüge aus?

Bei einem Polygonzug handelt es sich um eine zweidimensionale Reihe an Verbunden von Punkten durch Strecken zwischen Punktepaaren. Ein gutes Beispiel dafür ist ein GPS-Track. Es werden, während der Bewegung in regelmäßigen Abständen, Standorte bzw. Punkte gespeichert, die am Ende mit dem gelaufenen Weg dazwischen, als Strecke dargestellt, die zurückgelegte Route bilden. Ein solcher GPS-Track beinhaltet generell schon eine Art von Vereinfachung, da nicht jeder einzelne Schritt gespeichert wird, sondern in gleichmäßigen Intervallen Speicherungen vorgenommen werden. Um einen Polygonzug im Nachhinein zu vereinfachen, sind Algorithmen vonnöten.

In dem Paper „Performance Evaluation of Line Simplification Algorithms for Vector Generalization“ [2] werden einige davon einem Vergleich unterzogen. Inwieweit sich das Endergebnis vom Original unterscheidet sowie welche Rechenzeit es dafür in Anspruch nimmt, bilden die Kriterien dieses Vergleichs. Das Resultat dieses Papers zeigt, „[…] that the Douglas-Peucker algorithm produced the most accurate generalization.“ Aber „[…] the Lang algorithm and the Zhao-Saalfeld algorithm would be of greater practical benefit.“ Diese letzte Aussage begrenzte meine Auswahl auf diese beiden Algorithmen. Um mich auf einen Algorithmus festzulegen, benötigt es ein grundlegendes Verständnis für die Funktionsweise der Algorithmen.

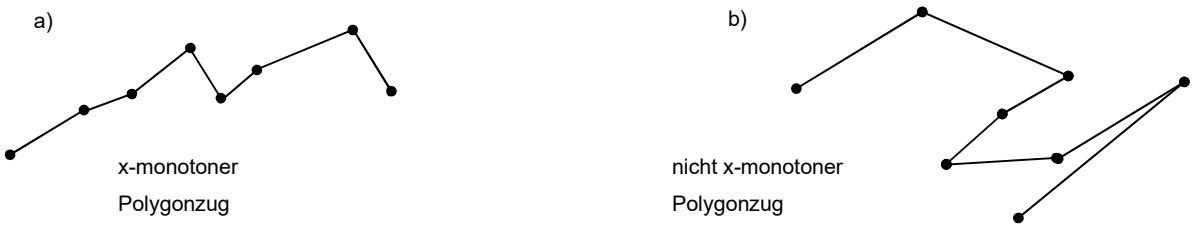

Abbildung 2: a) x-monotoner b) nicht x-monotoner Polygonzug

Zum leichteren Verständnis beschreiben die folgenden Abbildungen die Algorithmen anhand eines x-monotonen Polygonzugs. Das heißt, die Punkte sind nach x-Koordinaten sortiert, weshalb keine x-Koordinate eines Punktes kleiner sein kann als die seines Vorgängers. Wie auch in Abbildung 2 ersichtlich ist, sind die x-Werte der vierte und fünfte Punkt des nicht x-monotonen Polygonzugs geringer als die des Vorgängers. Das kann bei einem x-monotonen Polygonzug nicht vorkommen.

2.1 k-te Punkt Routine

Bevor die Grundideen zu den bereits erwähnten Algorithmen von Lang und Zhao-Saalfeld beschrieben werden, beginne ich mit dem einfachsten Algorithmus, der k-ten Punkt Routine, die allerdings oft (siehe auch Abschnitt 2.4) nicht zu einem optimalen Ergebnis führt. Die Routine verarbeitet die einzelnen Punkte des originalen Polygonzugs, behält den Startpunkt und danach bleibt nur noch jeder k-te Punkt im vereinfachten Polygonzug erhalten.

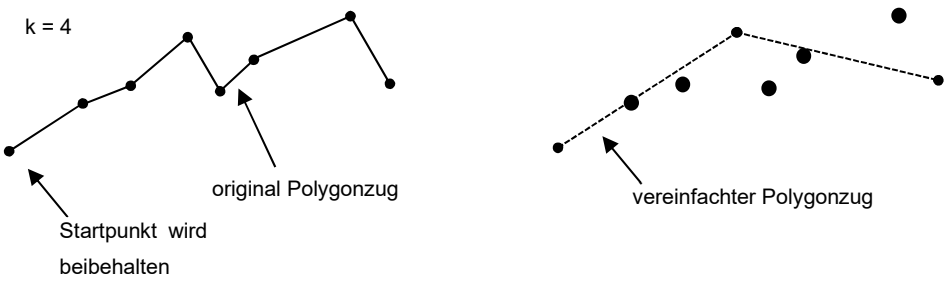

Abbildung 3: k-te Punkt Routine

Das Ziel einer Vereinfachung eines Polygonzugs ist es, mit weniger Punkten die gleiche Form abzubilden. Diese Routine ist jedoch für die Formerhaltung weniger gut geeignet. Wie schon in der Abbildung 3 zu sehen, ist der vereinfachte Polygonzug wesentlich detailarmer. Während das Original acht Punkte besitzt, enthält die Vereinfachung nur drei Punkte und ähnelt mit ihren zwei Strecken dem ursprünglichen Polygonzug kaum noch.

2.2 Zhao-Saalfeld Algorithmus

Der Zhao-Saalfeld Algorithmus teilt den Polygonzug in Suchbereiche auf. Diese werden in diesem Algorithmus sleeves genannt. Der erste sleeve beginnt bei dem ersten Punkt des Polygonzugs. Die erste Strecke des sleeves gibt die allgemeine Richtung an. Wenn eine Strecke in einem gewissen Maß von dieser Richtung abweicht und die zuvor selbst festgelegte Toleranz überschreitet, gehört der Endpunkt dieser Strecke nicht mehr zum sleeve. In der Abbildung 4 weicht der fünfte Punkt zu weit von dieser Toleranz ab und befindet sich nicht mehr in der Reichweite des sleeves, sodass dieser nicht mehr zum ersten sleeve gehört. In diesem sleeve werden dann alle Punkte bis auf den ersten und den letzten Punkt eliminiert. Der zweite sleeve entsteht mit dem letzten Punkt des ersten sleeves bis zum nächsten Richtungswechsel. Dieser Ablauf setzt sich bis zum Ende fort.

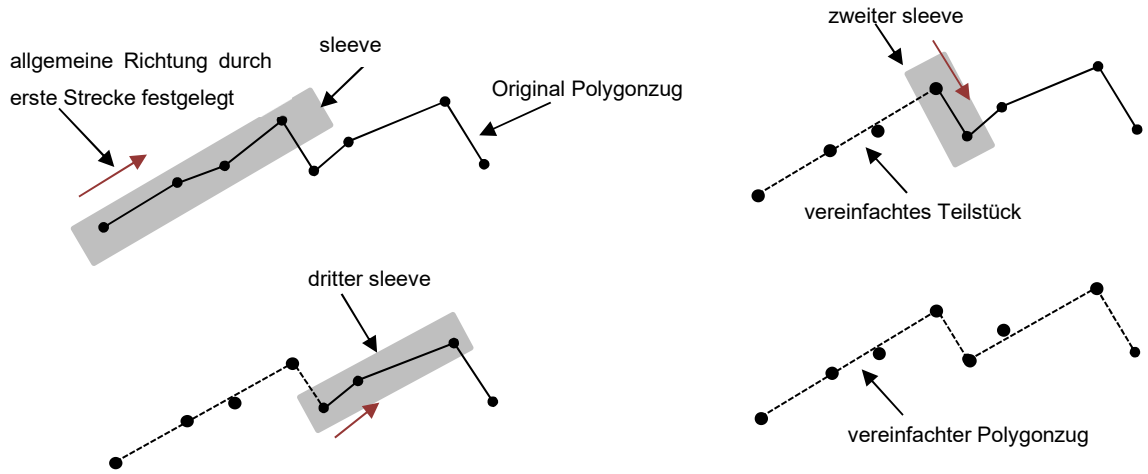

Abbildung 4: Zhao-Saalfeld Algorithmus

2.3 Lang Algorithmus

Der Lang Algorithmus teilt den Polygonzug auch auf, aber jeder Bereich besteht aus einer bestimmten Anzahl an Punkten. Zu Beginn wird eine Punkteanzahl s festgelegt, aus der jeder Suchbereich besteht. In der Abbildung 5 besteht ein Suchbereich aus 4 Punkten. Ist der erste Bereich gebildet, wird eine Strecke vom ersten Punkt P_0 zum

letzten Punkt P_s des Bereiches gezogen. Der Abstand der mittleren Punkte zu dieser Strecke wird mit einer ebenfalls festgelegten Toleranz t verglichen. Diese Toleranz darf allerdings nicht zu klein gewählt werden, da ansonsten möglicherweise keine Vereinfachung mehr vorliegt. Um diese Abstände auszurechnen, wird das Lot des betrachteten Punktes zur Strecke gebildet und die Länge dieses Lotes bestimmt. Ist auch nur einer dieser Abstände größer als die Toleranz, wird der letzte Punkt P_s aus dem Bereich ausgeschlossen, die Strecke wird nun vom ersten Punkt zum neuen Endpunkt P_{n-1} gezogen und die Abstände werden neu berechnet. Sollte ein Abstand der mittleren Punkte erneut größer sein als die Toleranz, wird P_{s-1} aus dem Bereich eliminiert und die darauf folgenden Schritte erneut wiederholt. Sind die Abstände alle kleiner als die Toleranz, werden nun alle mittleren Punkte entfernt und ein neuer Bereich wird betrachtet: vom endgültig letzten Punkt P_x (das x zeigt hier, dass das Ende des ersten Suchbereichs unbekannt ist) des vorherigen Bereichs bis zum neuen Endpunkt P_{x+s}. Im zweiten Suchbereich in der Abbildung 5 wird der letzte Punkt aus dem Suchbereich entfernt, da die Toleranz überschritten wurde. So werden nun nur noch drei Punkte betrachtet und der Endpunkt des Suchbereichs ist der sechste Punkt des Polygonzugs.

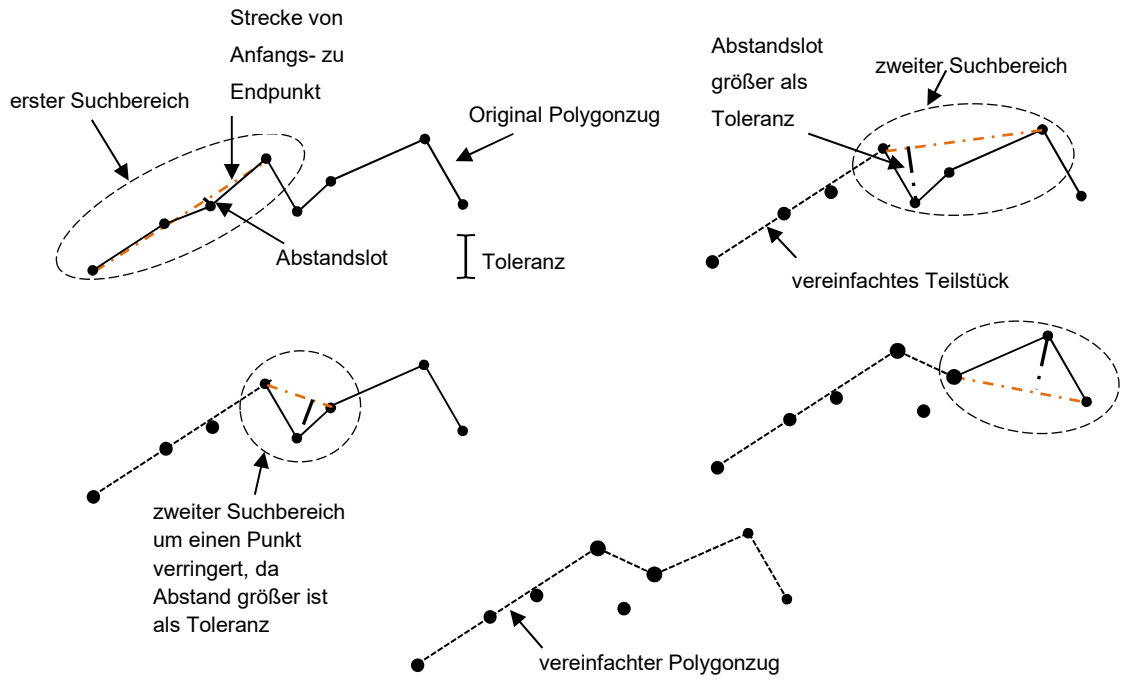

Abbildung 5: Lang Algorithmus

2.4 Maximum Distorsion

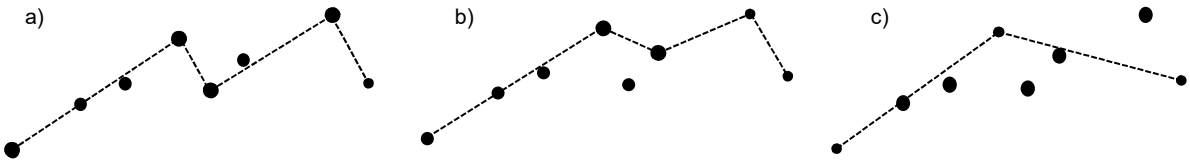

Abbildung 6: Vergleich von Algorithmen (a)Zhao-Saalfeld Algorithmus, b) Lang Algorithmus, c) k-te Punkt Routine)

Wie zuvor in Kapitel 2.3 erwähnt, sind die Algorithmen von Zhao-Saalfeld und Lang nicht so genau wie der Douglas-Peucker Algorithmus. Dagegen kann allerdings die k-te Punkt Routine dazu führen, dass der Original Polygonzug in seiner vereinfachten Version seine Form verliert. Wir sehen also im Vergleich der vereinfachten Polygonzüge in Abbildung 6, dass einige Algorithmen die Form des Originals trotz geringerer Punkteanzahl besser beibehalten als andere. Je weniger Punkte in der Vereinfachung vorhanden sind, desto mehr weicht der vereinfachte Polygonzug vom Original ab. Doch wie wird die Abweichung bestimmt, wenn es nicht genügt nach Augenmaß zu gehen?

Um die Abweichung exakter zu bestimmen, muss diese berechnet werden. Die Maximum Distorsion ist eine dieser Berechnungen, die zur Bestimmung der Abweichung verwendet werden kann. Sie wird berechnet, indem der größte Abstand zwischen den Punkten und den Strecken der einzelnen Bereiche gesucht wird. Egal wie der Polygonzug vereinfacht wird, resultieren daraus stets mehrere Strecken entstehen, die mindestens einen Punkt weniger als das Original besitzen. Die Abweichung kann daher nur zwischen den neu gebildeten Strecken und den Punkten im Original entstehen. Aus diesem Grund wird der Abstand vom Punkt zur Strecke berechnet. Der Abstand wird mit folgender Gleichung „Punkt -Gerade“ aus „Formeln + Hilfen“ [1] auf Seite 57 berechnet:

$$dist = \frac{|\overrightarrow{ea} \times \overrightarrow{mpa}|}{|\overrightarrow{ea}|} \; ; \quad \overrightarrow{mpa} = \overrightarrow{mp} - \vec{a}$$

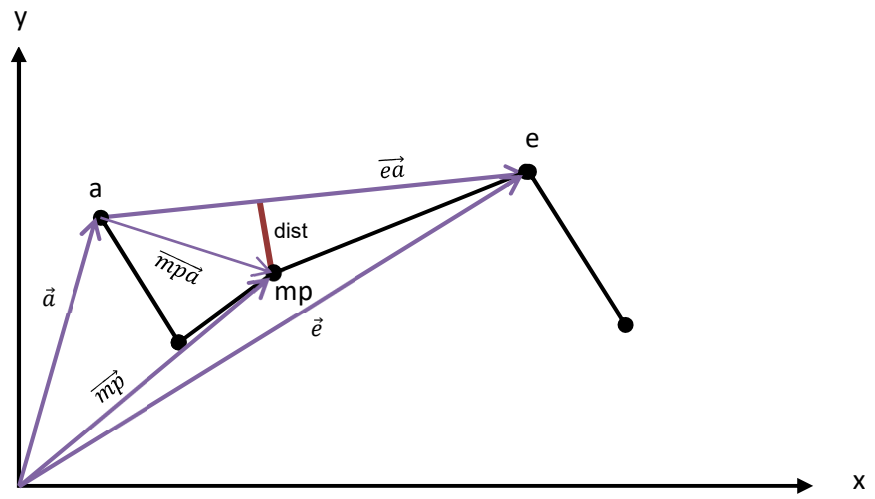

Abbildung 7: Visualisierung der Berechnung des Abstandes von einem der mittleren Punkte zur Strecke zwischen Anfangs- und Endpunkt

In dieser Gleichung gibt es drei wichtige Punkte zu betrachten: den Anfangspunkt des betrachteten Suchbereichs „a“, den Endpunkt des Suchbereichs „e“ und den Punkt „mp“, dessen Abstand zur Strecke berechnet werden soll. Die Abbildung 7 zeigt einen Polygonzug in schwarz mit vier Punkten innerhalb des Suchbereiches. Der betrachtete Punkt zwischen „e“ und „a“ ist mit „mp“ betitelt. Das Lot von diesem Punkt bis zur Strecke, das hier mit „dist“ bezeichnet wird, gilt es zu bestimmen. Diese Gleichung wird im Kapitel 3 erneut Erwähnung finden. Um daraus die maximale Abweichung des gesamten Polygonzugs zu erhalten, muss die vorhergehende Gleichung für jeden Punkt, den der Polygonzug besitzt, durchgeführt werden. Dies wird in der zweiten Gleichung durch das „$dist_i$“ dargestellt. Die Berechnung von $dist_i$ erfolgt dabei mit den Punkten aus dem originalen Polygonzug, durch welche die Abstände zu der Strecke des vereinfachten Polygonzugs ermittelt werden. Der dadurch berechnete Abstand wird mit dem bisher gespeicherten Maximum verglichen. Sollte der neue Abstand größer sein, wird das Maximum durch diesen ersetzt. Dieser Vorgang wird bis zum Ende des Polygonzugs durchgeführt und das Maximum, das zuletzt gespeichert ist, die maximale Abweichung darstellt.

$$dist_{max} = max_{0 \leq i \leq n} [dist_i]$$

2.5 Shape Dissimilarity

Die Berechnung der Shape Dissimilarity eines Polygonzugs ist ebenfalls eine Möglichkeit, um die Abweichung des vereinfachten Polygonzugs vom originalen Polygonzug zu quantifizieren. Dabei wird der Anstieg jeder Strecke des originalen Polygonzugs zwischen zwei aufeinander folgenden Punkten bestimmt. Vom vereinfachten Polygonzug wird dasselbe berechnet. Die Beträge der Differenzen, zwischen den Anstiegen des vereinfachten und des originalen Polygonzugs, werden addiert. Es erfolgt eine gewichtete Summierung. Die Wichtung wird berechnet, indem die Länge des i-ten Abstandes A_i durch die Summe der Längen aller Abstände A_1 bis A_k geteilt wird. In der Abbildung 8 ist k = 7, so wird also $L(A_i)$ durch $L(A_1)+ L(A_2)+ L(A_3)+ L(A_4)+ L(A_5)+ L(A_6)+ L(A_7)$ geteilt.

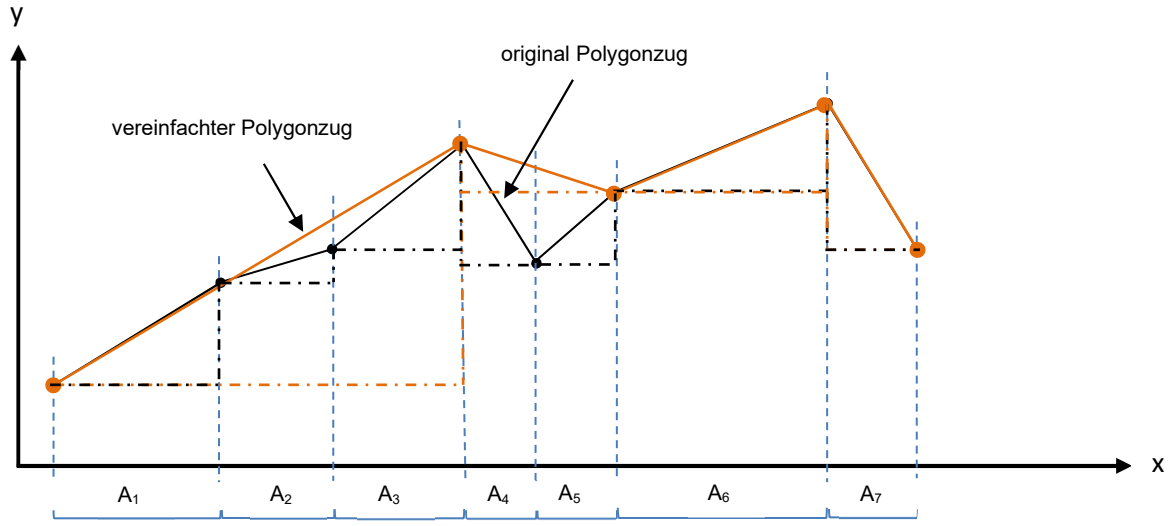

Abbildung 8: shape dissimiarity

3 Programmierung

Bevor ein Programm geschrieben werden kann, sollte sich jeder Studierende sicher sein, welche Funktionen es am Ende beinhalten und welche Anforderungen es erfüllen soll. Aus diesem Grund sollte eine Vorüberlegung der erste Schritt sein. Auch ich habe so mit der Erarbeitung meiner Vorgehensweise angefangen.

Meine Aufgabenstellung setzt voraus, dass ich mein Programm in Bausteine aufteile, damit jede Woche im Praktikum für algorithmische Geometrie einer dieser Bausteine bearbeitet werden kann. Das Programm sollte in ungefähr 10 Wochen bearbeitet werden, was also einer Anzahl von 10 Bausteinen entspricht. Da diese Arbeit auch den Python unerfahrenen Studierenden den Einstieg erleichtern soll, ergaben sich die ersten zwei Bausteine wie folgt: Zuerst wird eine festgelegte Anzahl an Punkten mit zufälligen Koordinaten erzeugt und darauf werden die Punkte ausgeplottet und mit Strecken fortlaufend verbunden, sodass ein Polygonzug entsteht und dargestellt werden kann. Wenn der Polygonzug vorhanden ist, benötige ich einen Algorithmus zur Vereinfachung.

Wie im Kapitel 2 schon beschrieben, entschied ich mich für die Algorithmen von Zhao-Saalfeld und Lang. Da nur ein Algorithmus in der Programmierung Anwendung finden soll, betrachtete ich die Funktionsweise beider und überlegte mir die Aufteilung für die restlichen Bausteine. (Die Aufteilung kann von der späteren Version abweichen.)

Tabelle 1: Vergleich überlegte Bausteinaufteilung von Zhao-Saalfeld und Lang

Reihenfolge	Zhao-Saalfeld	Lang
1.	Punkte erzeugen und nach x-Koordinaten sortieren	
2.	Punkte ausplotten und miteinander durch Strecken verbinden	
3.	Subtraktion der y-Werte (P_1-P_2, P_2-P_3, …) bei Wechsel im Vorzeichen anhalten	Punkteanzahl für die Suchbereiche auswählen, Toleranz festlegen, ersten Suchbereich abgrenzen und Gerade zwischen Anfangs- und Endpunkt bilden
4.	ersten sleeve erstellen (P1 bis Punkt vor Vorzeichenänderung bei der Subtraktion) und Punkte in der Mitte eliminieren	Abstand zwischen Punkten innerhalb von Anfangs- und Endpunktes und der Strecke messen und mit Toleranz vergleichen

5.	Iteration des Verfahrens in einer Schleife	letzten Punkt aus dem Bereich ausschließen bei Toleranzüberschreitung eines Punktes und erneute Geradenbildung
6.	-	Eliminieren der Punkte zwischen Anfangs- und Endpunkt des Suchbereiches und Speichern des Anfangs- und Endpunkts für den vereinfachten Polygonzug
7.	-	Bilden des nächsten Bereichs vom Endpunkt des alten Suchbereichs bis zum Punkt addiert mit der Anzahl an Punkten für Suchbereich
8.	-	erneute Geradenbildung und siehe 5. & 6., Wiederholung des Prozesses und Endausgabe des Ergebnisses
9.	-	Daten aus GIS auslesen und vereinfachen

Da diese Vorüberlegung vor der eigentlichen Programmierung und des Schreibens dieser Arbeit stattfand, entspricht der Schritt 3 nicht der Erklärung des Zhao-Saalfeld Algorithmus. Die Subtraktion der y-Werte lässt nur erkennen, ob ein Punkt die Richtung ändert oder nicht. Dabei wird allerdings nicht beachtet, dass trotz minimaler Abweichung von der Richtung - vorgegeben durch die erste Strecke - der sleeve noch beibehalten wird. Dennoch stützte ich mich im Folgenden auf diesen Vergleich.

Der Vergleich zeigt, dass der Lang Algorithmus besser für eine Aufteilung auf ungefähr 10 Wochen geeignet scheint. Ich entschied mich gegen den Zhao-Saalfeld Algorithmus, da auch mit ein oder zwei zusätzlichen Bausteinen, sollten sich einige Vorüberlegungen als komplexer erweisen als erwartet, wäre nicht ausreichend Material vorhanden. Aus diesem Grund habe ich mich dafür entschieden, den Lang Algorithmus zu verwenden. Nun kann das eigentliche Programmieren beginnen. Allerdings ist es noch nötig, eine geeignete Programmierumgebung zu finden.

3.1 Auswahl des Programmierwerkzeugs

Um ein Programm zu schreiben ist es nötig, im Vorfeld folgende Entscheidung zu treffen: Genügt es den Quellcode in einem Editor zu schreiben oder sollte in einer Entwicklungsumgebung gearbeitet werden? Daraus resultiert die Frage nach dem Unterschied zwischen beiden.

Wird ein einfacher Texteditor betrachtet, bietet dieser keine außergewöhnlichen Funktionen auf. Es ist möglich, einen Quellcode zu schreiben, es ist jedoch weder möglich Kommentare vom eigentlichen Programm noch Variablen von Funktionen visuell unterscheiden. Das bedeutet, dass es kein Syntax-Highlighting gibt.

Ein Beispiel für einen fortgeschritteneren Editor ist Notepad++ [3]. In diesem Editor kann jeder mit sehr vielen Programmiersprachen schreiben, bei welchen das Syntax-Highlighting funktioniert. Damit können hier Unterscheidungen zwischen den einzelnen Komponenten eines Programms gemacht werden. Eine wichtige Funktion die Notepad++ fehlt, ist die Fehlererkennung. Sollte im Programm also ein Semikolon fehlen, kann Notepad++ diesen Fehler nicht erkennen. Des Weiteren kann dieser Editor das Programm nicht ausführen oder debuggen, und somit keine kritischen Fehler in der Funktionsweise des Programms finden.

Eine Entwicklungsumgebung dagegen hat den Vorteil, für die gewünschte Sprache all das zur Verfügung zu stellen. Sie ist dafür ausgelegt, die einzelnen Komponenten jedes Quellcodes dieser Sprache erkennbar und unterscheidbar hervorzuheben, Fehler in der Syntax zu erkennen und darauf hinzuweisen, das Programm auszuführen und gegebenenfalls zu debuggen. Der Nachteil ist allerdings, dass eine Entwicklungsumgebung seltener für mehr als eine Programmiersprache ausgelegt ist.

Für jede Person, die noch nicht so vertraut mit einer Programmiersprache ist, sollte die Antwort dennoch klar sein. Denn trotz der Tatsache, dass eine (weitere) Entwicklungsumgebung heruntergeladen werden muss, ist es für jeden Anfänger einfacher, alle notwendigen Funktionen in einem Programm zu haben. Doch welche ist die Richtige für die Programmiersprache?

Die Entscheidung, welche Entwicklungsumgebung die selbst gewählten Voraussetzungen am besten erfüllt, liegt bei jedem Einzelnen. Um ein Beispiel für eine geeignete Entwicklungsumgebung darzulegen, stelle ich meine verwendete Umgebung vor. Da es sich in diesem Fall um Python handelt, stehen dafür einige Optionen zur Auswahl. Wer sofort auf die erste Website [12] vertraut, hat mit PyCharm [10] die beliebteste Wahl getroffen. In der Computerwoche [13] wird PyCharm als „zugänglich und einfach zu bedienen“ beschrieben. Ebenso wurde mir zu Beginn von „Algorithmische Geometrie“ diese Entwicklungsumgebung empfohlen, welche ich

daraufhin das ganze Praktikum über nutzte. Auch das Programm für diese Arbeit ist damit, wie in Abbildung 9 ersichtlich ist, programmiert worden.

Zusätzlich zu den Vorteilen einer Entwicklungsumgebung fällt die Einarbeitungszeit in PyCharm gering aus. Ob bereits Erfahrungen mit einer Entwicklungsumgebung gesammelt wurden oder nicht spielt hier nur eine geringe Rolle. Sobald PyCharm geöffnet wird und ein neues Projekt angelegt ist, sieht jeder sofort, wo der Quellcode geschrieben wird. Schnell können auch die Funktion zum Ausführen oder Debuggen gefunden werden. Die einzige Schwierigkeit bestand darin zu entdecken, wo die Bibliotheken eingebunden werden. Sobald dieses Hindernis überwunden ist, sollte es keine Schwierigkeiten mehr geben, da PyCharm eine große Auswahl an Bibliotheken besitzt, die nur noch im Projekt heruntergeladen werden müssen.

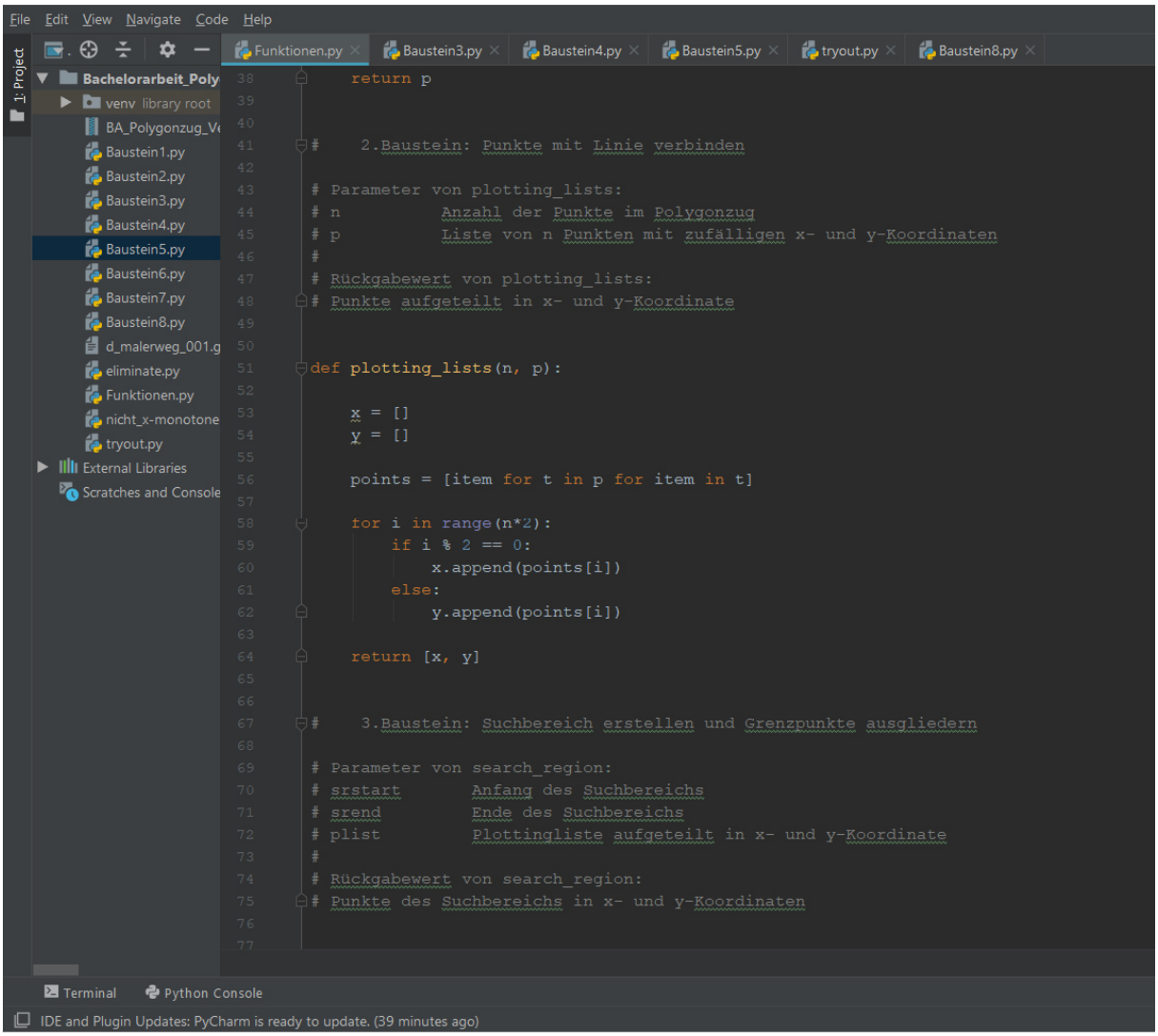

Abbildung 9: Screenshot von Quellcode in PyCharm im dunklen Design (Thema: Darcula) aus der Datei „Funktionen.py“

3.2 Grundwissen

Um einen Polygonzug zu vereinfachen, sollte jedem bekannt sein, was ein Polygonzug ist. Wie schon im Kapitel 1 beschrieben, handelt es sich bei einem Polygonzug um eine Reihe an Punkten in der jeder Punkt mit seinem Nachfolger durch eine Strecke verbunden ist. Ebenfalls sollte jeder wissen, dass eine Vereinfachung eines Polygonzugs darin resultiert, dass die vereinfachte Version eine geringere Anzahl an Punkten besitzt als das Original.

Studierende im Praktikum von „Algorithmischer Geometrie“ sollen pro Woche eine Aufgabe bearbeiten und damit einen Baustein fertigstellen. Nach der Tabelle in Kapitel 3 müsste nach neun Wochen das Programm fertig sein. Am Ende dieser Aufgaben sollte jeder die Antworten auf Fragen, wie diese, kennen:

Wie vereinfache ich ein Polygonzug? Wie berechne ich den Abstand zwischen einem Punkt und einer Strecke im Koordinatensystem? Wie erstelle ich zufällige Punkte in Python? Welche Bibliothek und Funktion benötige ich zum Ausplotten von Punkten? Wie berechne ich ein Vektorprodukt mit Python? Wie erstelle ich verallgemeinerte Funktionen für jeden einzelnen Baustein, um ihn im folgenden Baustein wieder zu verwenden?

3.3 Programm schreiben

Ist die ausgewählte Programmierumgebung ausgewählt, installiert und die Vorüberlegung gemacht, ist es endlich Zeit mit dem Programmieren zu beginnen. Im Folgenden wird mein Vorgehen bei der Programmierung beschrieben. Zu Beginn jedes Bausteins wird die Theorie zusammen mit der Implementierung in Python und am Ende die Verallgemeinerung für den Gebrauch in den nachfolgenden Bausteinen erläutert. So wurde in meinem Fall zuerst PyCharm geöffnet und ein neues Projekt angelegt. Diesem gab ich einen Namen, der sich ohne Probleme meiner Bachelorarbeit zuordnen ließ. Danach öffnete ich das Projekt und erstellte eine neue Python-Datei. Diese betitelte ich mit „Baustein1“. Eine weitere erstellte ich mit dem

Namen „Funktionen“ in der alle Bausteine in eine allgemein gültige Form gebracht werden, damit sie in den nachfolgenden Bausteinen wiederverwendet werden können.

Baustein 1

Die erste Aufgabe war, die Punkte zu erstellen und diese nach der x-Koordinate zu sortieren. Das hat den Vorteil, keinen komplexen und unübersichtlichen nicht x-monotonen Polygonzug zu erhalten, sondern einen x-monotonen, der sich besser darstellen und einsehen lässt. Aus diesem Grund wird zuerst die Anzahl der Punkte des Polygonzugs n festgelegt. Zum zufälligen Erzeugen von Punkten wird eine dementsprechende Funktion benötigt. In diesem Fall ist es die `random`-Funktion, die mit der gleichnamigen Bibliothek aufgerufen wird. Die `random`-Funktion benötigt zwei Eingaben, nachdem festgelegt wurde, welche Art von Wert verwendet werden soll. Um einen Definitionsbereich auszuwählen, musste ich zwischen ganzen Zahlen und Fließkommazahlen entscheiden. Aus Gründen der Übersichtlichkeit habe ich mich für Integer - also ganze Zahlen – entschieden. Somit war die erste Funktion `random.randint(0,100)`, wobei die 0 für den minimalen Betrag und die 100 für den maximalen Betrag des zufälligen Wertes steht. Die Funktion erzeugt bisher allerdings nur einen Wert, aber ein Punkt benötigt einen x- und einen y-Wert. Um die Anzahl an n Punkten zu erzeugen und zu speichern, erstellte ich eine Liste `P`, in der ich zwei Mal die Zufallsfunktion, das erste Mal für die x-Koordinate und das zweite Mal für die y-Koordinate, aufrufe. So entstand der erste Punkt. Dieser eine Punkt musste aber noch zu n Punkten werden, weshalb ich die Liste in eine `for`-Schleife mit `range(n)` eingeschlossen habe. Das einzige Problem war nun, dass bei der Ausgabe dieser Liste nur ein Punkt zu sehen war. Da dieses Programm immer einen Punkt in der Liste erzeugt und ihn im nächsten Durchlauf mit dem neu erzeugten überschreibt. Um das zu ändern braucht diese Zeile ein `append`. Dieses `append` hängt einen neuen Punkt an die Liste `P` an. Hierbei zeigte sich nun der Vorteil meiner Wahl von Integern. Die Liste von n Punkten besitzt 2n Werte. Tupel mit mehr als einem Eintrag werden mit einem Komma getrennt. Fließkommazahlen in Python werden in der Konsole mit einem Punkt dargestellt. Auch wenn der Unterschied zwischen Punkt und Komma besteht, dient eine Reduktion der Ziffern und Zeichen ähnlicher Art der Klarheit. Nun

war der erste Teil der Aufgabe gelöst, denn das Programm erstellte n Punkte mit x- und y-Koordinate. Der Quellcode sah damit so aus:

```
…
P = [ ]
for i in range[n]:
  P.append((random.randint(0,100),random.randint(0,100)))
```

Der zweite Teil des Programms, also die Sortierung nach x-Koordinate war einfacher als gedacht. Es existiert in Python eine Funktion namens `sort`. Diese erlaubt es dem Programmierer, entweder nach einem spezifischen Wert zu sortieren oder durch eine leere runde Klammer direkt nach dem ersten Wert zu sortieren. Der erste Wert der `append`-Funktion ist die x-Koordinate, weshalb in diesem Fall die leere Klammer genügte. Somit sind der zweite Teil sowie der erste Baustein mit folgender Zeile beendet:

```
P.sort()
```

Um Funktionen in Programmen oder Projekten wiederzuverwenden, sollten sie als sinnvoll benannte Funktionen definiert und mit ebenso sinnvoll benannten Argumenten gefüllt werden. Da sich dieser Baustein mit der Erstellung des Polygonzugs beschäftigt, war es sinnvoll, ihn `create_polyline` zu nennen. Nun war zu überlegen, welche Argumente diese Funktion benötigt. Variablen, die sich aus der zuvor aufgestellten Funktion als notwendig erweisen, müssen als ein Argument aufgeführt werden. Im Baustein 1 sind das die Anzahl Punkte n und der kleinste sowie der größte Wert in der `random`-Funktion. Die Definition der Funktion lautet somit wie folgt:

```
def create_polyline(n,xmin,xmax,ymin,ymax)
```

Obwohl es nicht unbedingt notwendig ist, die Argumente für die Zufallsfunktion in x- und y-Koordinate aufzuteilen, ist es dennoch besser, dem potenziellen Nutzer die Möglichkeit zu bieten, unterschiedliche Größen anzugeben. Der Rest der Funktion bleibt bestehen. Der einzige Unterschied ist, dass sich die Werte in der `random`-

Funktion zu Variablen ändern. Als `return` wird `p` an das ausführende Programm zurückgegeben.

```
return p
```

Baustein 2

Der zweite Baustein beschäftigte sich mit der Aufgabe, die Liste der Punkte aus Baustein 1 mit Strecken von einem Punkt bis zu seinem Nachfolger auszuplotten - also graphisch darzustellen. Dafür bietet sich die `matplotlib`-Bibliothek mit der `pyplot` Funktion an. Um nicht bei jedem Aufruf den vollständigen Befehl `matplotlib.pyplot` ausschreiben zu müssen, wird diese Bibliothek beispielsweise als `plt` bezeichnet. Die Informationen im Pyplot Tutorial [11] zu der Funktion `plot` zeigen, dass diese bei zwei Koordinaten ein Tupel mit x-Koordinaten und eines mit y-Koordinaten benötigt. Unglücklicherweise sind in der Liste `P` n Tupel mit n Punkten, statt zwei Tupel mit n x- und y-Koordinaten versehen. Mit Hilfe der Seite „GeeksforGeeks“ [8] fand ich einen einfachen Weg für die Lösung dieses Problems.

```
Points = [item for t in p for item in t]
```

`p` ist in diesem Fall die erzeugte Liste aus `create_polyline` und `t` ist eine noch nicht verwendete Variable. `Points` besitzt somit 2n Werte ohne Tupel. Die x-Koordinaten befinden sich an jeder geraden Stelle, da eine Liste bei 0 beginnt und somit y-Koordinaten an jeder ungeraden Stelle dieser Liste speichert. Eine Berechnung, die sich dafür anbietet, diese Liste mit zwei Tupeln einzurichten, nennt sich Modulo-Rechnung. Sie berechnet den Rest in einer Division, das bedeutet, dass ein Wert i mod 2 (im Programm mit dem Operator `%2` dargestellt) gerade ist, wenn das Ergebnis der Rechnung 0 ist. Da sich jede x-Koordinate an einer geraden Stelle in der Liste befindet, lässt sich die Modulo-Rechnung wie folgt anwenden:

```
for i in range(2*n):
        if i%2==0:
             x.append(Points[i])
        else:
             y.append(Points[i])
```

Dieser Quellcode bedeutet, dass sich in der Liste `Points` jetzt 2n Koordinaten befinden und das Programm die Elemente dieser Liste einzeln verarbeitet und sollte die Stelle, an der sich das Element befindet, gerade sein, (in Python ist die Null eine gerade Zahl) wird der Inhalt von `Points` an der Stelle `i,` der Liste `x` hinzugefügt. Sollte das nicht der Fall sein und das Ergebnis der Modulo-Rechnung ungleich Null sein, wird der Inhalt von `Points` an der Stelle `i,` der Liste `y` hinzugefügt. Nun war es möglich, diese Punkte mit folgenden Zeilen auszuplotten.

```
plt.plot(x,y)
plt.show()
```

Die Ausgabe entsprach nun der Abbildung 1 und der zweite Baustein war fertig. Für die Verallgemeinerung mussten nur leichte Veränderungen vorgenommen werden. Die Funktion `plotting_lists` erhielt die Argumente `n` für die Anzahl der Punkte und `p` für die Liste an Punkten mit n Tupeln. Der Quellcode blieb gleich und die Listen x und y wurden mit der Funktion zurückgegeben. Da return nur einen Wert zurückgeben kann, mussten beide Listen als eins übertragen werden.

```
return [x, y]
```

Baustein 3

Die Aufgabe des Bausteins 3 war es, den ersten Suchbereich abzugrenzen und die erste Strecke durch den Anfangs- und Endpunkt zu ziehen. Dazu benötigte es eine festgelegte Anzahl `s` an Punkten für jeden Suchbereich. In der `range(s)` werden zwei Listen `sx` und `sy` mit den x- oder y-Koordinaten aus dem Ergebnis von `plotting_lists` gefüllt, die sich in dem Suchbereich von `s` Punkten befinden.

```
for i in range(s):
  sx.append(x[i])
  sy.append(y[i])
```

Um die Strecke zu bilden, werden die Grenzpunkte also der Anfangs- und Endpunkt benötigt. Erhalten werden diese durch die `append`-Funktion in den Listen

`gp_x` und `gp_y`. Dafür wird eine weitere Variable benötigt, da der Endpunkt bei `s-1` liegt, was als Rechnung an einigen Stellen im Programm problematisch sein kann. Diese hieß `e` und war das Ergebnis von `s-1`.

```
gp_x.append(sx[0])
gp_x.append(sx[e])
```

Im gleichen Format wurden `sy[0]` und `sy[e]` in `gp_y` eingefügt. Um sowohl die Strecke als auch den Polygonzug darzustellen, mussten nur beide vor dem `plt.show` stehen.

```
plt.plot(plist[0], plist[1])
plt.plot(gp_x, gp_y)
plt.show
```

In `plist[0]` befindet sich die Liste an x-Koordinaten aus der Funktion `plotting_lists`. `plist[1]` beinhaltet die Liste der y-Koordinaten. Die Ausgabe von `plt.show` liegt in Abbildung 10 vor.

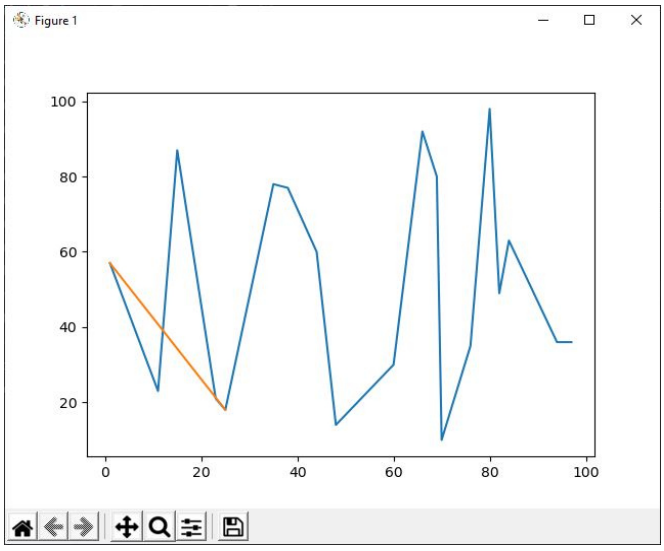

Abbildung 10: Polygonzug mit Strecke

Für diesen Baustein werden diesmal mehrere Funktionen benötigt. Zuerst wird die Funktion `search_region` erstellt, welche den Suchbereich abgrenzen soll. Diese erfordert das Argument `srstart`, da nicht jeder Suchbereich gleichmäßig nach n Punkten erneut beginnt. Dasselbe gilt für `srend`, da sich nach der Berechnung vom

Abstand zwischen mittleren Punkten und der Strecke die Anzahl der Punkte im Suchbereich ändern kann. Für die Funktion ist allerdings `plist`, die Liste der Punkte, am wichtigsten. Darauf folgt die Funktion `border_points` mit den Argumenten sr und `sb`, um die Grenzpunkte, also Anfangs- und Endpunkt, zu ermitteln. `sr` ist die Anzahl der Punkte in einem Suchbereich und `sb` das Ergebnis von `search_region`.

Baustein 4

Nachdem die Strecke erstellt wurde, ist es nun notwendig, den Abstand zwischen dieser und den einzelnen Punkten zu bestimmen. Da die Abstände zwischen dem Anfangspunkt `a`, dem Endpunkt `e` und der Strecke jeweils 0 sind, bedingt diese Berechnung mehr Rechenzeit bei der Ausführung des Programms. Aus diesem Grund werden nur die mittleren Punkte `mp` zur Berechnung verwendet. Diese müssen erst aus der Funktion `search_region` extrahiert werden. Um den Anfangs- und Endpunkt getrennt nutzen zu können, habe ich diese als Vektor mit Hilfe der `numpy`-Bibliothek in `a` und `e` gespeichert. Damit der Schreibaufwand gering blieb, habe ich diese als `np` eingefügt. Die Berechnung in einer Vektor-Form war dadurch möglich.

```
a = np.array([gp_x[0], gp_y[0], 0])
e = np.array([gp_x[1], gp_y[1], 0])
```

Für die Berechnung des Kreuzproduktes werden drei Koordinaten benötigt. Da dieser Polygonzug nur zweidimensional ist, wird die z-Koordinate als 0 deklariert. Um daraufhin die mittleren Punkte zu bestimmen, werden sie mit Hilfe der `append`-Funktion und einer `for`-Schleife in `range(sb-2)` an die Listen `mpx` und `mpy` angehängt.

```
for i in range(sb-2):
        mpx.append(sx[i+1])
        mpy.append(sy[i+1])
```

Diese beiden Funktionen werden `sb-2` Mal durchgeführt. Da der Anfangs- und der Endpunkt nicht zu den mittleren Punkten gehören, werden diese ausgelassen. `i+1` zeigt, dass der Durchlauf ab der Stelle 1 in `sx` und `sy` beginnt und nicht ab der Stelle

0. Die Gleichung, die danach für die Berechnung des Abstandes verwendet wird, ist dieselbe, die im Kapitel 2.4 erklärt wurde. Sie sah im Quellcode wie folgt aus:

```
for o in range(len(mp)):
        mpa = mp[o]-a
        ea = e - a
        x = mpa[1]*ea[2]-mpa[2]*ea[1]
        y = mpa[2]*ea[0]-mpa[0]*ea[2]
        z = mpa[0]*ea[1]-mpa[1]*ea[0]
dist = (np.sqrt(np.square(x)+np.square(y)+np.square(z))) /
np.sqrt(np.square(ea[0])+np.square(ea[1])+np.square(ea[2])))
```

Die Gleichungen `x`, `y` und `z` entsprechen dem Kreuzprodukt aus Kapitel 2.4 in Matrix-Schreibweise. Das Ergebnis aus diesen Gleichungen ergibt einen Vektor (`x, y, z`), der das Lot zwischen Punkt und Strecke darstellt, aus dem daraufhin der Abstand `dist` bestimmt werden kann. Die Funktion `np.sqrt` ist die Berechnung der zweiten Wurzel und `np.square` die Berechnung des Quadrats. Somit wird in der Gleichung der letzten Zeile der Abstand wie folgt berechnet:

$$dist = \frac{\sqrt{x^2+y^2+z^2}}{\sqrt{ea_1{}^2+ea_2{}^2+ea_3{}^2}}$$

Für die Verallgemeinerung in der Datei „Funktionen.py“ werden also der Anfangs- und Endpunkt `a` und `e` des Suchbereichs benötigt, sowie eine Liste der mittleren Punkte `mp` und der Abstand `dist`. Der Anfangspunkt wird mittels der Funktion `start_point` festgelegt sowie der Endpunkt mit der Funktion `end_point`. Beide Funktionen ähneln stark denen in Baustein 4. Der einzige Unterschied ist, dass in beiden Funktionen die Liste `gp` in `gp_x` und `gp_y` aufgeteilt ist. Dagegen ist die Funktion `middle_points`, in welcher die mittleren Punkte bestimmt werden, ein genaues Abbild der Funktionen aus dem Baustein 4. Die letzte Funktion, die in diesem Baustein noch bedeutsam ist, ist die Abstandsberechnung `distance`. Gegenüber dem Baustein 4 besitzt `distance` in Funktionen eine Variable `dist_max`. Diese steht zu Beginn des Durchlaufs auf 0. Sollte das Ergebnis der Abstandsberechnung `dist` größer sein als der aktuelle `dist_max` Wert, wird dieser Wert zu `dist_max`.

```
if dist > dist_max:
```

```
        dist_max = dist
```

Sind alle mittleren Punkte im Suchbereich durchlaufen, wird der größte Abstand, also `dist_max`, von der Funktion `distance` zurückgegeben.

Auffällig ist jedoch, dass an einigen Stellen bei der Berechnung des Abstandes bedeutend kleinere Werte berechnet werden als eine Betrachtung der graphischen Darstellung in Abbildung 10 vermuten lässt. Der Grund dafür liegt darin, dass es sich bei der Berechnung von `distance` um den Abstand zwischen einem Punkt und einer Geraden handelt. Wenn also eine Strecke zwischen Anfangs- und Endpunkt nicht ausreicht, um ein senkrechtes Lot zu bilden, wird die Strecke zu einer Geraden, die sich über die Grenzpunkte hinaus erstreckt. Das führt dazu, dass das Lot vor dem Anfangs- oder hinter dem Endpunkt gebildet wird und der Punkt in der Berechnung näher an der Strecke scheint, als er in der graphischen Darstellung ist.

Baustein 5

Dieser Baustein entfernt die letzte Stelle des Suchbereichs nach dem Vergleich mit der Toleranz `t`, sollte `dist_max` größer sein als `t`. In der `range(sb, 1, -1)` bearbeitet eine `for`-Schleife die Funktionen `sr`, `gp`, `a`, `e`, `mp` und dist.

```
for j in range(sb, 1, -1):
```

Diese Schleife beginnt an der Stelle `sb` , also am Ende des Suchbereichs und mit jedem Durchlauf setzt sie sich dekrementierend fort, bis zu der ersten Stelle. Nachdem der Polygonzug erstellt wurde, werden alle aufgerufenen Funktionen ausgeführt. Darauf wird `dist` berechnet und anschließend mit folgenden Zeilen überprüft, ob der größte Abstand im Suchbereich kleiner als `t` ist.

```
if dist <=t:
        break
```

Sollte sich diese Bedingung als wahr erweisen, bleibt dieser Durchlauf in der `for`-Schleife der Einzige und Letzte. Bei einem `dist`-Wert größer als `t`, werden die Funktionen in der `for`-Schleife so oft ausgeführt, bis ein Durchlauf einen `dist`-Wert besitzt, der kleiner als `t` ist.

In der Funktion `shorten_sr` werden die Argumente `srstart`, `sb`, `plist` und `t` übergeben. Darin iteriert eine `for`-Schleife die Funktionen `sr`, `gp`, `a`, `e`, `mp` und `dist` und der Suchbereich wird immer um einen Punkt verringert bis `dist_max` nicht mehr größer als `t` ist. Der `return`-Wert ergibt sich aus dem Startpunkt des Suchbereiches addiert mit der Variablen `j` aus der `for`-Schleife und subtrahiert mit 1. Dieser Wert gibt damit die letzte Stelle des Suchbereichs zurück. Da eine Liste mit 0 beginnt, erfolgt im Rückgabewert noch eine Subtraktion mit 1.

Baustein 6

Die Aufgabe dieses Bausteins lautet: Wird die Toleranz mit dem größten Abstand zwischen Punkt und Strecke im Suchbereich nicht mehr überschritten, werden die mittleren Punkte entfernt und ein neuer Suchbereich gebildet. Zu Beginn dieses Bausteins werden also `n`, `sb` und `t` festgelegt. Darauf werden die Funktionen `create_polyline` und `plotting_lists` aufgerufen. `plende`, also der letzte Punkt, wird vor dem ersten Durchlauf auf 0 gesetzt. Statt diese Funktionen in einer Schleife zu iterieren, werden sie nacheinander ausgeführt, um die nötigen Schritte in der Verallgemeinerung zu veranschaulichen. Im ersten Durchlauf werden zwei neue Listen `new_line_x` und `new_line_y` erstellt. Der erste Punkt wird mit `append(plist[0][plende])` für die x-Koordinate in der Liste `new_line_x` gespeichert. Das Gleiche passiert mit `plist[1]` für die y-Koordinate.

```
new_line_x.append(plist[0][plende])
new_line_y.append(plist[1][plende])
```

Im zweiten Durchlauf wird, vor der Speicherung des Startpunktes, `plende` mit der Funktion von Baustein 5 `shorten_sr` erneuert, indem für `srstart plende` verwendet und somit `plende` überschrieben wird.

```
plende = Funktionen.shorten_sr(plende, sb, plist, t)
```

Dieser Vorgang muss so lange wiederholt werden, bis `plende len(plist)-1` entspricht.

Eine der vereinfachten Funktionen heißt `get_new_line`. Sie entspricht dem ersten Durchlauf, in dem die neue Liste gespeichert wird, mit `plende` als Verweis auf

die letzte Stelle im Suchbereich. In der Funktion `eliminate` wird die Funktion `shorten_sr` aufgerufen, die daraufhin mit `plende` als Argument arbeitet sowie mit `sb`, `plist` und `t`. Der Rückgabewert ist `el`, in welchem die Anzahl der Punkte im finalen Suchbereich übergeben werden.

Baustein 7

Der Baustein 7 ermöglicht, alle Funktionen, die bisher erstellt worden sind, iterationsfähig zu machen. Das bedeutet, dass am Ende dieses Bausteins der Polygonzug vollständig vereinfacht sein soll. Bevor eine Iteration stattfindet, muss allerdings der Polygonzug vorhanden sein, weshalb `create_polyline` und `plotting_lists` zu Beginn des Bausteins aufgerufen werden müssen. Mit `plende = 0` kann bereits der Startpunkt im vereinfachten Polygonzug festgehalten werden, weshalb zu Beginn `get_new_line` aufgerufen wird. Darauf folgt eine `for`-Schleife in `range(n)`, die mit `if` die Bedingung aufstellt, dass, sollte `sb` kleiner oder gleich der noch fehlenden Punkte des Polygonzugs sein, `plende` durch die Funktion `eliminate` neu festgelegt wird und dieser in die Liste der neuen Punkte angefügt.

```
for y in range(n):
   if n-1-plende >= sb:
       plende = Funktionen.eliminate(plende, sb,   plist, t)
```

Sollte dies nicht der Fall sein und 3 oder 4 Punkte noch ausstehen, werden in diesem Suchbereich erneut die mittleren Punkte bestimmt, der Abstand berechnet und bei Toleranzüberschreitung erneut eine Reduktion der Punkte vorgenommen.

```
else:
    if n - plende > 2:
       for v in range(n - 1 - plende):
          plende = Funktionen.eliminate(plende, n-plende,
          plist, t)
          new_line    =    Funktionen.get_new_line(plist,
          plende)
```

Um auszuschließen, dass der Lang Algorithmus nicht vorsieht, eine Vereinfachung bei einer geringeren Anzahl an festgelegten Punkten im Suchbereich vorzunehmen, suchte ich nach einer zweiten Quelle, die diese Ungewissheit ausräumte. Das tat die Seite „psimpl“ [7]. Sind diese Voraussetzungen erfüllt, wird `plende` erneut der Liste hinzugefügt. Ist die Anzahl der übrigen Punkte kleiner oder gleich 2 sein, wird nur noch der letzte Punkt in die Liste `new_line` eingefügt. Sollte es keine weiteren Punkte geben, wird die Iteration mit `break` beendet und der vollständige vereinfachte Polygonzug kann ausgegeben werden.

Da dieser Baustein dazu gebraucht wird, die wesentlichsten Bausteine bis zum Ende des Polygonzugs zu wiederholen, um im letzten Schritt den vereinfachten Polygonzug ausgeben zu können, erhielt die Funktion den Namen `redo_to_simplify`. Damit diese auch bei einem bereits vorhandenen Polygonzug verwendet werden kann, wird `plist` in den Argumenten verlangt, sowie die Vorgaben zur Größe des Suchbereichs, der Toleranz und der Anzahl der Punkte im Polygonzug. Vor der for-Schleife wird `plende` auf 0 gesetzt und der Anfangspunkt direkt in die Funktion `new_line` eingefügt. Danach folgt der exakte Aufbau von Baustein 7 und daraufhin wird `new_line` als Liste zurückgegeben.

Baustein 8

Baustein 8 ist der letzte Baustein und beendet die Aufgabe mit einer Anwendungsmöglichkeit. Er befasst sich damit, eine gpx-Datei, also einen GPS-Track, zu vereinfachen. Mit Hilfe von „deparkes“ [6] einem Blog von Duncan Parkes, sowie der Antwort von Charles P. auf „stack overflow“ [4] war es mir möglich einen einfachen Weg zu finden, GPS-Tracks zu nutzen. Dafür war es notwendig die Bibliothek `gpxpy.gpx` zu importieren. Außerdem war es wichtig eine gpx-Datei zu finden. Die Internetseite fernwege.de [9] war dafür die geeignete Quelle. Jeder der eigene GPS-Tracks zur Verfügung stehen hat, darf diese auch verwenden. Ich habe mir den Malerweg ausgesucht und heruntergeladen. Ist die Datei auf dem Rechner vorhanden, muss diese in das Python-Projekt kopiert werden. Mit `open('d_malerweg_001.gpx')` kann nun auf die Datei zugegriffen werden. `gpx.parse` bringt die Datei auf eine Form, die das Programm lesen kann.

```
gpx_file = open('d_malerweg_001.gpx')
gpx = gpxpy.parse(gpx_file)
```

Darauf werden Breiten- und Längengrade aus der Datei extrahiert und können in Punkte umgewandelt werden.

```
for track in gpx.tracks:
    for segment in track.segments:
        for point in segment.points:
            x.append(point.latitude)
            y.append(point.longitude)
```

Besitzt man diese Punkte muss `n` durch `len(x)` ermittelt werden. Dadurch fand ich heraus, dass die gpx-Datei des Malerwegs 1084 Punkte gespeichert hatte. Aus diesem Grund wählte ich eine Anzahl von 10 Punkten für den Suchbereich aus. Als Toleranz nutzte ich den Wert 0.05. Nun war es nur noch nötig, die Funktion `redo_to_simplify` aufzurufen. Bereits im Anschluss war es möglich, den vereinfachten Polygonzug auszugeben.

In der Verallgemeinerung wird in der Funktion `get_gpx_data gpx_file_name` als Argument benötigt. Dazu ist noch einmal zu betonen, dass diese Datei in dem Python-Projekt stehen muss. Die Extraktion der Breiten- und Längengrade blieb gleich. Am Ende gibt diese Funktion `plist` zurück. Die Ausführung der Funktionen aus Baustein 7 und 8 zeigt, dass, wie in Abbildung 11 zu sehen, die Vereinfachung des Malerweges erfolgt ist.

So habe ich den Malerweg mit 1084 Punkten auf einen Polygonzug mit 122 Punkten vereinfacht. Wie der vereinfachte Polygonzug in vielen Stellen mit dem Original übereinstimmt, weicht er an einigen weit davon ab. Um weniger Details des originalen Polygonzugs zu verlieren, müssten die Punkte im Suchbereich und die Toleranz verringert werden. So erhält man einen Polygonzug mit mehr Punkten in der Vereinfachung und damit mehr Einzelheiten des Originals beizubehalten.

Mit entsprechenden gpx-Dateien lassen sich viele Arten von zweidimensionalen Polygonzügen bearbeiten. Unter anderem können so Ländergrenzen kartographisch vereinfacht werden. Ist nicht das gewünschte Level der Vereinfachung erreicht, ist es

möglich in diesem Programm die Anzahl der Punkte im Suchbereich sowie die Toleranz zu bearbeiten und nach eigenen Wünschen anzupassen.

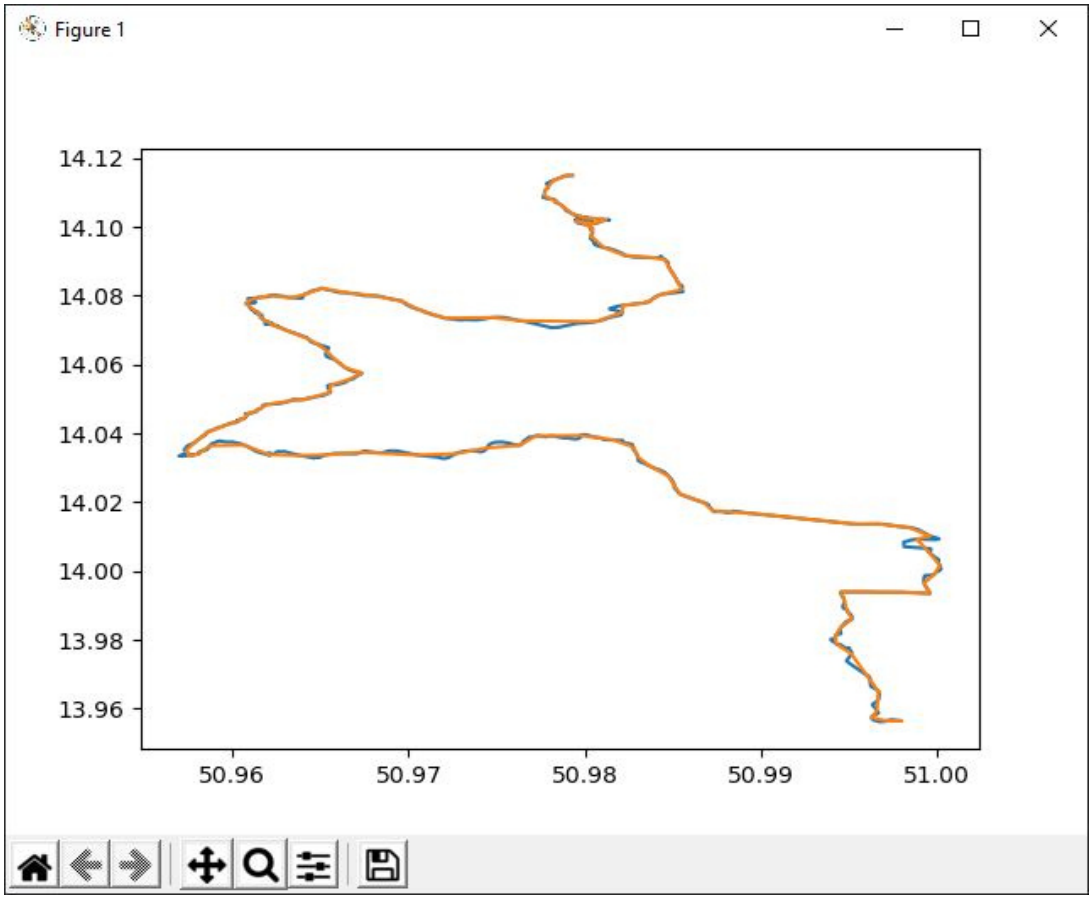

Abbildung 11: Malerweg (blau: Original, orange: vereinfacht)

4 Fazit

Die Aufgabe bestand darin, ein Programm zu entwickeln, das in dem Praktikum von „Algorithmische Geometrie“ Anwendung finden kann. Dazu sollte es so aufgeteilt werden, dass innerhalb von etwa 10 Wochen je eine Teilaufgabe bearbeitet werden kann. Diese sollten so gestellt sein, dass auch Studierende ohne Python-Kenntnisse diese leicht bearbeiten können. Es sollte eine Vereinfachung von Polygonzügen betrachtet werden und letztendlich in einem praktischen Beispiel angewandt werden.

In der Literaturrecherche habe ich mich auf die Suche nach passenden Algorithmen für die Vereinfachung begeben und informierte mich über Hilfen hinsichtlich der Programmierung und nützlichen GPS-Tracks. Ich habe ein Programm entwickelt, das aus 8 Bausteinen mit ansteigendem Schwierigkeitsgrad besteht, die modular aufeinander aufbauen. Sie vereinfachen einen Polygonzug mit Hilfe des Lang Algorithmus und finden in Baustein 8 Anwendung in Form eines GPS-Tracks. Die Aufgaben für die Bausteine sind so gewählt, dass auch ein unerfahrener Python-Nutzer die Möglichkeit besitzt, im Praktikum in „Algorithmischer Geometrie“ jede Woche einen Baustein abzugeben. Des Weiteren ist es den Studierenden möglich, durch dieses Programm einige Funktionen von Python kennenzulernen und sich mit diesen auseinanderzusetzen. Der modulare Aufbau lässt diesbezüglich Veränderungen zu, sollte das Programm am Ende nicht den gewünschten Anforderungen entsprechen.

Eine Vereinfachung mit Hilfe dieses Programms, lässt sich mit keinem anderen Dateitypen als gpx-Dateien ohne weiteren Programmieraufwand erreichen, wobei es nach der Implementierung eines Polygonzugs keine Relevanz hat, ob dieser x-monoton oder nicht x-monoton ist.

Dieses Programm kann in der Kartographie Einsatz finden. Da Karten in einem verkleinerten Maßstab dargestellt werden, ist eine Reduktion der Details sinnvoll. In Einträgen im Grundbuch bezüglich der Grundstücksgrenzen sind Vereinfachungen weniger sinnvoll, da es in solchen Fällen zum Rechtsstreit führen kann.

Um den Grad der Vereinfachung darzustellen, wäre es möglich, eine Funktion zu implementieren, welche die Länge des originalen Polygonzugs und der

Vereinfachung ermittelt, um beide miteinander vergleichen zu können. Diese Berechnung erfordert eine weitere Vektorrechnung. Ein weiterer Schritt wäre die Berechnung der Shape Dissimilarity, um die Abweichungen zwischen Original und Vereinfachung darzustellen. Eine Erweiterung, die eine größer Komplexität aufweisen könnte, wäre die Umsetzung dieses Programms im dreidimensionalen Raum. Dafür müssten weitere Überlegungen angestellt werden, wie sich die dritte Dimension auf die Berechnung des Abstandes zwischen mittleren Punkten und der Strecke auswirkt.

Abschließend ist zu sagen, dass einige Erweiterungsmöglichkeiten existieren. Diese Bachelorarbeit und das Programm bieten hierfür eine solide Grundlage sowie für die Einführung in ein Praktikum des Moduls „Algorithmischer Geometrie“ mit Hilfe der Programmiersprache Python zum Thema Vereinfachung von Polygonzügen.

5 Abbildungsverzeichnis

6 Quellenverzeichnis

Literatur:

(1) Gerhard Merziger, Günter Mühlbach, Detlef Wille, Thomas Wirth, Binomi Verlag, „Formeln + Hilfen Höhere Mathematik“, ISBN: 978-3-923923-36-6, 7. Auflage

(2) Wenzhong Shi und ChuiKwan Cheung, The Cartographic Journal, „Performance Evaluation of Line Simplification Algortihms for Vector Generalization“, Ausgabe: Vol. 43 No.1. März 2006

Internet:

(3) Bernd Klein, Python Kurs, „NumPy Tutorial“, unter: https://www.python-kurs.eu/numpy.php (abgerufen am: 22.11.2020)

(4) Charles P., stack Overflow, Antwort zur Frage: „How to extract .gpx data with python“, unter: https://stackoverflow.com/questions/11105663/how-to-extract-gpx-data-with-python (abgerufen am:21.11.2020)

(5) Don Ho, Notepad++, „what is Notepad++“, unter: https://notepad-plus-plus.org/ (abgerufen am: 11.11.2020)

(6) Duncan Parkes, deparkes, „Python GPX Import Coordinates“, unter: https://deparkes.co.uk/2016/05/20/python-gpx-import/ (abgerufen am: 21.11.2020)

(7) Elmar de Koning, psimpl, „Lang simplification“, unter: http://psimpl.sourceforge.net/lang.html (abgerufen am: 22.11.2020)

(8) everythingispossible, GeeksforGeeks, „Python| Convert list of tuples into list“, unter: https://www.geeksforgeeks.org/python-convert-list-of-tuples-into-list/ (abgerufen am: 18.11.2020)

(9) fernwege.de, fernwege.de „GPS Tracks zum Download für Wanderwege“, unter: https://www.fernwege.de/gps/tracks/index.html (abgerufen am: 21.11.2020

(10) JetBrains, JetBrains, „PyCharm“, unter: https://www.jetbrains.com/de-de/pycharm/ (abgerufen am: 11.11.2020)

(11) matplotlib, Matplotlib, „Pyplot tutorial“, unter: https://matplotlib.org/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py (abgerufen am: 17.11.2020)

(12) Marta Szyndlar, STXNEXT, „The Best Python IDEs and Code Editors (According to Our Developers and the Python Community)“, unter: https://www.stxnext.com/blog/best-python-ides-code-editors/ (abgerufen am: 10.11.2020)

(13) Serdar Yegulalp, Computerwoche, „Die richtige Python IDE finden“, unter: https://www.computerwoche.de/a/python-lernen-leicht-gemacht,3331244,2 (abgerufen am: 11.11.2020)

7 Eigenständigkeitserklärung

Name: Behr

Vorname: Mandy Katharina

geboren am: 16.06.1998

Matr. Nr.: 23214

Hiermit erkläre ich gegenüber der Hochschule Merseburg, dass meine Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften übernommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht und auch noch nicht veröffentlicht.

Datum: Unterschrift: