FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

FH Aachen

Werkzeugmaschinen Labor der RWTH Aachen

Gruppe Digitale Prozesskette und Datenanalyse

Seminararbeit

Analyse und Optimierung von Fertigungsgenauigkeiten: Ein sensorischer Ansatz für die Fehlererkennung und Korrektur von CNC-Werkzeugbahnen

Altdorf, Florian

Aachen, 6. Januar 2025

1.Betreuer: Prof. Dr. Martin Reißel

2.Betreuer: M.Sc. Martin Krömer

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Analyse und Optimierung von Fertigungsgenauig-

keiten: Ein sensorischer Ansatz für die Fehlererkennung und Korrektur von CNC-Werkzeugbahnen

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Altdorf, Florian

Aachen, den 06.01.2025

Unterschrift der Studentin / des Studenten

Kurzfassung

Die vorliegende Arbeit untersucht die Analyse und Optimierung von Fertigungsgenauigkeiten im CNC-Fräsen mittels sensorischer Ansätze. Im Fokus steht die Entwicklung eines Tools zur Fehlererkennung und Korrektur von Werkzeugbahnen basierend auf dem Nearest-Neighbor-Algorithmus und der Berechnung von Korrekturwerten. Ausgangspunkt ist die Auswertung von Daten, die durch eine Abtragssimulation mit Dexel-Modellen und Koordinatenmessungen gewonnen wurden.

Zur effizienten Zuordnung der Messdaten zu Werkzeugbahnen wurde ein KD-Tree-basierter Ansatz implementiert. Zusätzlich ermöglicht die Integration mathematischer und informationstechnischer Konzepte wie der Vektorisierung die eine effiziente Verarbeitung großer Datenmengen darstellt. Die Laufzeitanalyse zeigt, dass die entwickelten Algorithmen auch bei großen Datensätzen stabil und praxistauglich sind.

Die Ergebnisse demonstrieren, dass durch die präzise Zuordnung von Messpunkten und die Berechnung von Fehlervektoren eine signifikante Verbesserung der Fertigungsgenauigkeit erreicht werden kann. Die Arbeit legt damit eine Grundlage für die Optimierung von Fertigungsprozessen und bildet die Basis für zukünftige Entwicklungen, wie die Integration in eine vollständige Werkzeugbahnplanung.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Erfassung und Auswertung von Daten aus Fertigungsprozessen ist ein zentraler Bestandteil der industriellen Forschung und ist entscheidend im Kontext des digitalen Wandels und der Industrie 4.0. Das Werkzeugmaschinenlabor WZL der RWTH Aachen zählt zu den führenden Forschungsinstituten Europas im Bereich Werkzeugmaschinen und der Vernetzung von Produktionsprozessen (Internet of Production).

Diese Seminararbeit fokussiert sich auf die Auswertung von Daten im Kontext des Fräsens. Aufgabe ist es, durch die systematische Auswertung der erfassten Fertigungsdaten den Bearbeitungsprozess zu optimieren. Insbesondere sollen Abweichungen der Werkzeugbahn identifiziert und entsprechende Maßnahmen zur Minimierung dieser Fehler abgeleitet werden. Daraus lässt sich eine Grundlage für die Planung von Werkzeugbahnen im Fräsen herleiten.

1.2 Problemstellung

Die zentrale Herausforderung dieser Arbeit besteht darin, die gemessenen lokalen Fehler mit dem Fertigungsprozess in Beziehung zu setzen. Hierfür ist eine präzise Zuordnung der Messdaten zu den Werkzeugbahnen erforderlich. Während der Fertigung werden die Werkzeugbahnen mit einer Abtastrate von 500 Hz aufgezeichnet. Diese Informationen, zusammen mit der Geometrie des Rohteils, dienen als Grundlage für eine am WZL entwickelte Abtragssimulation. Die Simulation transformiert die zeitabhängigen Informationen der Werkzeugbahnen in eine Oberflächengeometrie, die durch Dexel dargestellt wird.

Eine zusätzliche Herausforderung ergibt sich aus den unterschiedlichen Messstellen der Dexel-Daten und der Ergebnisse des Koordinatenmessgeräts. Um eine Verknüpfung dieser beiden Datensätze zu ermöglichen, werden die Dexelinformationen zunächst auf die Soll-Oberfläche des Werkstücks projiziert. Anschließend wird für jeden Messpunkt des Koordinatenmessgeräts der entsprechende Punkt aus den Dexel-Daten zugeordnet, um eine präzise Korrelation zwischen den beiden Datensätzen herzustellen.

Neben der Zuordnung der Messdaten ist die effiziente Verarbeitung großer Datenmengen eine weitere Anforderung. Die entwickelte Methode muss in der Lage sein, Größenordnungen von bis zu 10^6 Einträgen ressourcenschonend und zeitoptimiert zu verarbeiten.

Die Grundlage der Berechnungen bildet eine STL-Datei, die den Soll- bzw. Idealzustand des Werkstücks beschreibt. Die Korrekturwerte werden auf Basis der Vermessung eines bereits gefertigten Bauteils berechnet, wobei die lokalen Abweichungen und deren absolute Positionen durch ein Koordinatenmessgerät erfasst werden.

2 Theoretische Grundlagen

2.1 CNC-Fräsen

Nach der DIN-Norm 8580 zählt das Fräsen zu den Fertigungsverfahren mit geometrisch bestimmter Schneide. Allen spanenden Verfahren ist gemein, dass ein Schneidkeil mechanisch Material vom Werkstück abtrennt, wobei Späne entstehen.[DIN8580]

Das Fräsen ist ein subtraktives präzises Verfahren der spanenden Fertigung. Das Fräswerkzeug der Maschine wird mit einer relativen Geschwindigkeit zum Werkstück um die eigenen Achse rotiert und trägt mit einer Vorschubbewegung, meistens senkrecht zur Drehachse des Wehrstücks, spanend Schichten ab. [Brecher2018]

Eine entscheidende Weiterentwicklung des Fräsens ist die Einführung von Fräsmaschinen mit Computerized Numerical Control (CNC). Diese computergestützte Technologie hat die Bearbeitung revolutioniert, indem sie die automatisierte Fertigung von Werkstücken mit anspruchsvollen Geometrien ermöglicht.[Newmann2008]

Das in dieser Arbeit entwickelte Tool nutzt die Werkzeugbahndaten dieser CNC-Fräsmaschinen. Diese Daten stellen die Basis für die Analyse und Optimierung der Fertigungsprozesse dar und ermöglichen eine präzise Bewertung der Werkzeugbewegungen und ihrer Abweichungen vom Soll-Zustand. So wird eine Brücke zwischen der praktischen Fertigung und der datenbasierten Prozessoptimierung geschlagen.

2.2 Dexel-Modell

Das Dexel-Modell ist eine Methode zur Simulation subtraktiver Fertigungsprozesse, bei der Material abgetragen wird, um die gewünschte Geometrie eines Werkstücks zu erzeugen. Die Oberfläche des Werkstücks wird durch eine Aufreihung paralleler Linien, sogenannter Dexel, beschrieben. Jede Linie wird durch Start- und Endpunkte definiert und ist in einem regelmäßigen Feld angeordnet, das die Geometrie des Werkstücks diskret darstellt. Für die Darstellung im drei dimensionalen Raum werden die Dexel nicht nur in einer einzelnen Raumrichtung, sondern in drei orthogonalen Richtungen angeordnet. Diese Methode ist nach Boess unter der Tri-Dexel-Methode bekannt.

Im Kontext der subtraktiven Fertigung wird die drei dimensionale Dexel-Modelldarstellung verwendet, um den Materialabtrag durch die Bewegung der Frässchneide zu modellieren. Wie in Abbildung (2.1) dargestellt schneidet die Frässchneide die Dexel-Linien, wodurch deren Länge reduziert wird. An den Schnittstellen entstehen neue Dexel, die die verbleibende Geometrie nach dem Materialabtrag beschreiben. Dieser sich wiederholende Prozess ermöglicht es, die Geometrie des Werkstücks Schritt für Schritt zu aktualisieren, wobei jeder Bearbeitungsschritt direkt im Dexel-Modell abgebildet wird. Durch die kontinuierliche Anpassung der Dexel-Daten kann das Werkstück in Abhängigkeit vom Fertigungsprozess simuliert werden.

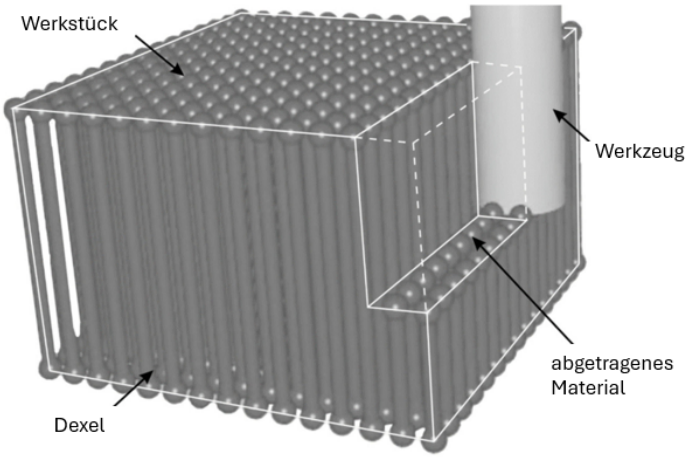

Abbildung 2.1: Dexel-Darstellung eines Fräsprozesses[Boess2021]

2.3 Mathematische Grundlagen

Das simulierte Dexel-Modell bildet in Verbindung mit den Daten der Werkzeugbahn und dem Soll-Werkstück, das in der STL-Datenstruktur, repräsentiert wird, die Grundlage für die Berechnungen zur Auswertung der Messdaten. Die STL-Datenstruktur beschreibt die Soll-Geometrie des Werkstücks durch eine diskrete Approximation, bei der die Oberfläche in eine Vielzahl von Dreiecken unterteilt wird. Diese Geometriebeschreibung erlaubt es, genaue Vergleiche zwischen der Soll- und Ist-Geometrie durchzuführen.

Die für die Auswertung der Abweichungen maßgebenden Berechnungen basieren auf Konzepten der analytischen Geometrie aus dem Bereich der linearen Algebra. Insbesondere wird die Berechnung der Korrekturwerte durch die Lagebeziehung zwischen Geraden und Ebenen ermöglicht. Diese Beziehung wird herangezogen, um den Schnittpunkt einer der Geraden, die eine Linie im Dexel-Model repräsentiert, mit einer Ebene oder Dreiecksfläche der STL-Datenstruktur zu berechnen.

Für die Berechnung der Lagebeziehung Gerade-Ebene wird eine Gerade wie in Gleichung (2.1 - 2.3) in Parameterform aufgestellt,

$$\vec{r}(t) = \vec{r}_0 + t\vec{v}, \tag{2.1}$$

$$\vec{r}_0 = (x_0, y_0, z_0), \tag{2.2}$$

$$\vec{v} = (v_x, v_y, v_z). \tag{2.3}$$

wobei $\vec{r}_0$ den Stützvektor, $\vec{v}$ den Richtungsvektor und t den Parameter darstellen.

Die Ebene kann durch die Koordinatengleichung dargestellt werden,

$$ax + by + cz + u = 0, \tag{2.4}$$

wobei (a, b, c) den Normalenvektor der Ebene und u eine Konstante repräsentieren.

Für die Bestimmung des Schnittpunkts wird die Parametergleichung der Geraden in die Koordinatengleichung der Ebene substituiert.

$$a(x_0 + tv_x) + b(y_0 + tv_y) + c(z_0 + tv_z) + u = 0. \tag{2.5}$$

Durch Umstellen erhält man den Parameter t.

$$t = -\frac{ax_0 + by_0 + cz_0 + u}{av_x + bv_y + cv_z}. \tag{2.6}$$

Der Schnittpunkt $\vec{r}_{\text{Schnitt}}$ wird durch Substitution von t in die Parametergleichung der Geraden ermittelt.

$$\vec{r}_{\text{Schnitt}} = \vec{r}_0 + t \cdot \vec{v}. \tag{2.7}$$

Durch diese Methodik wird der Fehler zwischen dem simulierten Dexel-Modell und dem Soll-Werkstück systematisch ausgewertet. Dabei erfolgt eine Zuordnung der lokalen Abweichungen zu den entsprechenden Dreiecken der STL-Datenstruktur.

Die hier verwendeten mathematischen Ausdrücke basieren auf den Beschreibungen von Jung. [Jung2024]

2.4 Informationstechnische Grundlagen

Die effiziente Verarbeitung von großen Datenmengen ist ein Kernaspekt dieser Arbeit. Um das zu realisieren liegt es nahe sich an Informationstechnischen Strukturen und Algorithmen zu bedienen.

2.4.1 Nearest-Neighbor-Search

Die *Nearest Neighbor Search* ist ein Verfahren, das für einen gegebenen Abfragepunkt p den nächsten Nachbarn q_{nearest} aus einer Menge $M = \{q_1, q_2, ..., q_n\}$ zu finden. Dabei wird die geringste Distanz zwischen p und q mit der **euklidische Distanz** ermittelt. Diese wird durch die folgende Gleichung beschrieben:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}. \tag{2.8}$$

Für jeden Punkt q in der Menge M wird die Distanz $d(p,q)$ berechnet und der Punkt mit der geringsten Distanz wird als Lösung identifiziert. Die Lösung q_{nearest} wird iterativ in die Distanzberechnung $d(p, q_{\text{nearest}})$ substituiert und bei jedem Schritt neu berechnet. Eine Implementierung, bei der alle Punkte in der Menge M verglichen werden ist ineffizient, da der Berechnungsaufwand linear mit der Anzahl der Punkte wächst. Um dieses Problem zu lösen, werden optimierte Datenstrukturen wie Bäume eingesetzt. [Taunk2019]

Nach Ottman und Widmayer werden Bäume abstrahiert als Listenstrukturen mit einer endlichen Anzahl an Knoten und Kindern angesehen. Ein Knoten wird als Wurzel bezeichnet wenn er keinen Vorgänger und mindestens ein Kind besitzt. Kinder werden als Blätter bezeichnet wenn sie selbst keine Kinder haben. Die Ordnung eines Baumes gibt die maximale Anzahl an Kindern an die ein Knoten besitzen darf. Das gilt einheitlich für alle Knoten. Ein Baum der Ordnung zwei wird als Binärbaum bezeichnet und ist Grundlage für die in dieser Arbeit verwendete KD-Tree Baumstruktur. [Ottmann2017]

Eine effiziente Umsetzung der *Nearest Neighbor Search* basiert auf der Verwendung eines KD-Tree, der für die Verarbeitung von mehrdimensionalen Daten konzipiert ist. Ein KD-Tree strukturiert die Punkte so, dass der Suchraum durch Unterteilung der Daten in kleinere Bereiche reduziert wird. Jeder Knoten des Baums repräsentiert eine Dimension und teilt den Raum entlang einer Achse (x- oder y-Achse bei zweidimensionalen Daten) in zwei Unterräume auf.

Diese Aufteilung wird rekursiv durchgeführt, bis alle Punkte in den Blättern des Baumes untergebracht sind. Dadurch können Punkte, die nicht in der Nähe des Abfragepunkts p liegen, frühzeitig ausgeschlossen werden. Der Berechnungsaufwand wird infolgedessen reduziert.

Funktionsweise des KD-Trees

Die Funktionsweise eines KD-Trees kann anhand einer zweidimensionalen Darstellung leicht nachvollzogen werden (siehe Abbildung 2.2). In diesem Beispiel teilt der KD-Tree die Punkte in der Ebene durch Trennlinien entlang der Koordinatenachsen auf. Jede dieser Trennlinien entspricht einem Knoten im Baum, während die Blätter des Baums die eigentlichen Datenpunkte enthalten. Dadurch entsteht eine hierarchische Struktur, die es ermöglicht, den Raum effizient in Unterbereiche zu unterteilen.

Die Abbildung zeigt auf der linken Seite, wie die Punkte im 2D-Raum durch vertikale und horizontale Linien getrennt werden. Diese Linien resultieren aus der rekursiven Teilung entlang wechselnder Dimensionen. Der erste Knoten, die Wurzel des Baums, teilt den Raum entlang der x-Achse. In diesem Beispiel ist A der Wurzelknoten, dessen vertikale Linie die Punkte mit $x < A_x$ (links) von denen mit $x > A_x$ (rechts) trennt. Im nächsten Schritt wird der Raum entlang der y-Achse durch B geteilt, wodurch die Punkte oberhalb und unterhalb der Linie voneinander getrennt werden. Dieses Wechselspiel zwischen den Dimensionen setzt sich fort, bis jeder Bereich nur noch einen Punkt enthält.

Die rechte Seite der Abbildung zeigt die korrespondierende Baumstruktur. Jeder Knoten im Baum repräsentiert eine Trennlinie, die den Raum weiter unterteilt. Die Wurzelknoten und ihre Unterbäume repräsentieren größere Unterteilungen, während die Blätter die einzelnen Datenpunkte darstellen. Punkte wie C, D, E und F sind in den Blättern des Baums organisiert und markieren die Endpunkte der Teilungsstruktur.

Ein entscheidender Vorteil des KD-Trees zeigt sich in der Effizienz der Nearest Neighbor Search. Während einer Suchanfrage werden nur diejenigen Unterräume durchsucht, die potenziell den nächsten Nachbarn enthalten könnten. Dies führt dazu, dass viele unnötige Berechnungen vermieden werden, indem große Teile des Raums ausgeschlossen werden. Im Vergleich zu einer linearen Suche, bei der jeder Punkt überprüft werden müsste, reduziert der KD-Tree die Anzahl der notwendigen Berechnungen erheblich. Dies macht ihn besonders geeignet für Anwendungen, bei denen große Mengen an mehrdimensionalen Daten effizient verarbeitet werden müssen [Histrov2023].

Während der *Nearest Neighbor Search* werden nur die Unterräume durchsucht, die potenziell den nächsten Nachbarn enthalten könnten. Dadurch wird die Anzahl der notwendigen Berechnungen im Vergleich zu einer linearen Suche erheblich reduziert.[Histrov2023]

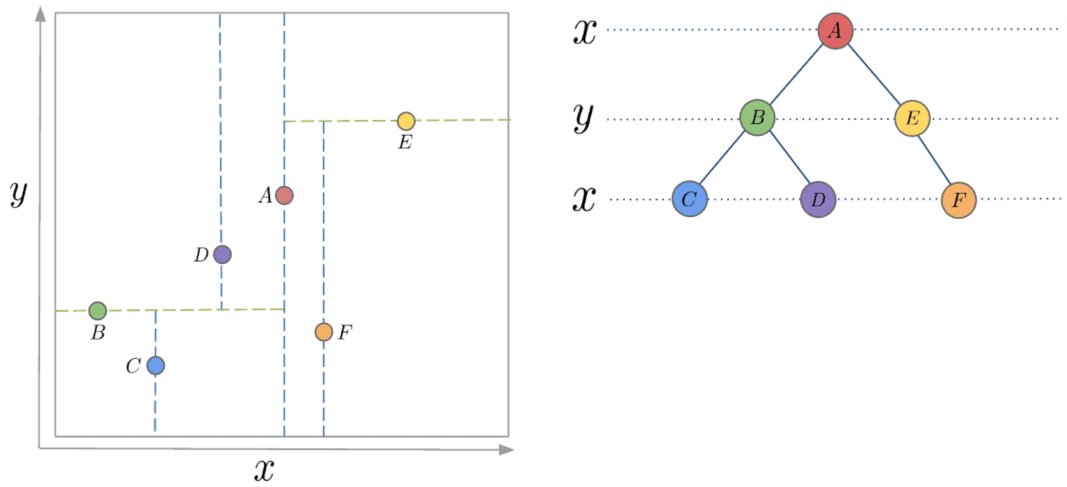

Abbildung 2.2: Zweidimensionale Darstellung eines KD-Trees mit Trennlinien entlang der x- und y-Achse.[Histrov2023]

Die Kombination aus *Nearest Neighbor Search* und KD-Tree ermöglicht es, Datensätze mit einer großen Anzahl von Punkten effizient zu durchsuchen. Die rekursive Unterteilung des Suchraums sorgt dafür, dass die Berechnung der Distanzen zielgerichtet und ressourcenschonend durchgeführt wird. Dies macht den KD-Tree zu einem Werkzeug für Anwendungen, bei denen große Datenmengen in mehrdimensionalen Räumen verarbeitet werden müssen.

2.4.2 Vektorisierung

Python, als interpretierte Sprache, weist bei numerischen Berechnungen Einschränkungen in der Laufzeiteffizienz auf. Das stellt eine Herausforderung dar, wenn Rechenoperationen in einem größerem Maßstab erforderlich sind. Um diese Einschränkung zu entschärfen, kommen Bibliotheken zum Einsatz, die speziell für effiziente numerische Operationen entwickelt wurden.[Sharp]

Eine der am weitesten verbreiteten und etablierten Bibliotheken ist NumPy [NumPy2020]. NumPy basiert auf einer Implementierung in der Programmiersprache C und ermöglicht eine signifikante Beschleunigung numerischer Berechnungen. Durch die Nutzung vor optimierter Routinen und kompilierten Codes bietet NumPy eine leistungsfähige Grundlage für skalierbare numerische Operationen.[NumInt2019]

Ein Schlüsselprinzip bei der effizienten Nutzung von NumPy ist die Vektorisierung, die auf der Verwendung von NumPy-Arrays basiert. NumPy-Arrays sind homogene Datenstrukturen mit fester Größe, die für effiziente numerische Berechnungen optimiert sind. Dabei werden die zu berechnenden Daten in NumPy-Arrays transformiert, sodass sie durch Vektor- und Matrixoperationen verarbeitet werden können. Dies ermöglicht es, skalierbare Datensätze ohne den Einsatz von iterativen Schleifen zu bearbeiten. Dieser Ansatz nutzt die Vorteile der linearen Algebra aus und stellt sicher, dass der Overhead der Python-Laufzeitumgebung reduziert wird. Die Vektorisierung führt nicht nur zu einer Reduktion der Berechnungszeiten, sondern auch zu einer Komplexitätsreduktion im Quellcode. Iterative Prozesse, werden durch skalierbare Matrixoperationen ersetzt. Dieser Ansatz ist insbesondere bei großen Datenmengen von Vorteil, um effiziente numerische Verfahren zu ermöglichen.[NumArr2019]

3 Zielstellung und Herangehensweise

Das Ziel dieser Arbeit ist die Entwicklung einer Softwareerweiterung zur Berechnung von Korrekturwerten für die Werkzeugbahn auf Basis eines Nearest-Neighbor-Ansatzes sowie der anschließenden Berechnung von Korrekturvektoren. Dabei basiert die Arbeit auf einer bestehenden Software, die bereits den Vergleich zwischen dem Soll-Werkstück (STL-Datei) und dem Ist-Werkstück (Dexel-Daten) ermöglicht. Diese bestehende Lösung berechnet die Schnittpunkte der Dexel-Daten mit der Soll-Oberfläche des Werkstücks und dient als Ausgangspunkt für die weiteren Arbeiten in dieser Arbeit.

Der Fokus dieser Arbeit liegt auf der effizienten Zuordnung der gemessenen Dexel-Schnittpunkte zu den Werkzeugbahnen mithilfe eines Nearest-Neighbor-Verfahrens sowie der anschließenden Berechnung der Abweichungsvektoren. Diese Abweichungen sollen zur gezielten Korrektur der Werkzeugbahn dienen, um die Fertigungsgenauigkeit zu optimieren.

3.1 Aufbau auf bestehender Software

In Vorarbeit zu dieser Arbeit wurde das Tool für die Berechnung der Fehler zwischen Ist- und Soll-Werkstück aus MATLAB[MATLAB1994] in Python übersetzt. Im Folgenden werden die Grundfunktionen der Bestandssoftware benannt.

3.1.1 Parameterdaten und Datenstrukturen

Die auszuwertenden Werkstückdaten bestehen aus einer dwp-Datei für die Dexel-Simulation als Ist-Werkstück und der STL-Datei als Soll-Werkstück. Die dwp-Datei ist ein betriebsinternes Dateiformat, welches für die Speicherung von Dexelinformationen benutzt wird. Für das Einlesen der Dateien werden die von Python gegebenen Bibliotheken NumPy[NumPy2020], Pandas[Pandas] und Trimesh[Trimesh] verwendet.

1. **Soll-Werkstück**: Das Soll-Werkstück in STL-Datenstruktur wird mithilfe von Trimesh als Polygonnetz eingelesen. Die verarbeiteten Daten der Datei resultieren in zwei Datenstrukturen von NumPy-Arrays die als Tupel ausgegeben werden.
   - **Vertices:** Die Ecken der Dreicke geschrieben
   - **Faces:** Die Dreiecksflächen
2. **Ist-Werkstück:** Das Ist-Werkstück liegt in dwp Format vor und wird mit einem Parser eingelesen. Die Funktion gibt Zellen in Form einer Liste und die Metadaten als Dictionary wieder.

3.1.2 Dexel-Punkte berechnen

Die durch den Parser analysierten Zellen mit Dexel-Informationen werden in der Funktion *calculate_dexel_point* verarbeitet, um die räumlichen Start- und Endpositionen der einzelnen Dexel zu berechnen.

def calculate_dexel_point(cells):

3.1.3 Schnittpunktberechnung

Die Funktion *calculate_dexel_errors* ermittelt aus den zuvor berechneten *dexel_points*, *faces* und *vertices* den Schnittpunkt bzw. Lotpunkte der Dexel mit den Dreiecksflächen, um so die Fertigungsabweichung auszuwerten.

def calculate_dexel_errors(cells, vertices, faces):

3.2 Aufbau der Softwareerweiterung

Nachfolgend werden die Hauptfunktionen der Erweiterung benannt und kurz beschrieben.

3.2.1 Nächster Nachbar Algorithmus

Die *find_nearest_neighbor* Funktion sucht für alle Lotpunkte nächste Messpunkte, sodass jedem Lotpunkt ein Zeitstempel zugeordnet werden kann.

find_nearest_neighbors(results, txt_allData)

3.2.2 Fehlervektor der Werkzeugbahn

Durch Zuordnung der Zeitstempel können die Abstände von den Lotpunkten zur zugeordneten Werkzeugposition ermittelt werden.

calculate_distances_for_toolpath(entries_with_timestamp, csv_table)

4 Nächster Nachbar Algorithmus

Die hier vorgestellte Funktion bildet den zentralen Teil der Berechnungen zur Zuordnung von Messergebnissen und geometrischen Simulationsdaten. Während die vorherigen Schritte auf bestehenden Softwarekomponenten zur Datenaufnahme und Datenverarbeitung aufbauten, stellt dieser Abschnitt eine Erweiterung dar, die die berechneten Schnittpunkte aus den Dexel-Daten mit den tatsächlichen Messdaten verknüpft. Dabei wird ein KD-Tree zur effizienten Suche mit dem Nearest-Neighbor-Algorithmus verwendet, um den nächsten Nachbarn zu ermitteln. Ziel ist es, den gemessenen Sensorpunkten zu jedem Zeitpunkt Schnittpunkte zuzuordnen, um die Abweichungen in der Werkzeugbahn besser berechnen zu können.

Die Funktion `find_nearest_neighbors` verknüpft berechnete Schnittpunkt mit den Sensordaten des Ist-Werkstücks aus einer txt-Datei, indem sie ein Nearest-Neighbor-Verfahren innerhalb eines definierten Radius verwendet. Als Eingangsparameter dient eine Liste der berechneten Schnittpunkte mit Zeitstempeln sowie ein `Pandas DataFrame`, das die XYZ-Koordinaten der Sensordaten enthält. Der Radius definiert den maximalen Abstand, innerhalb dessen ein Schnittpunkt einem Sensordatenpunkt zugeordnet werden kann. Die Funktion liefert eine Liste, die die XYZ-Koordinaten der Sensordaten mit den Zeitstempeln der nächstgelegenen Schnittpunkte kombiniert.

Zu Beginn werden die Schnittpunkte und ihre Zeitstempel aus den Eingabedaten extrahiert und die Sensordaten in Form eines NumPy-Arrays vorbereitet. Anschließend werden die Schnittpunkte in einem KD-Tree organisiert, um die Abfrage von Nachbarn effizient zu gestalten.

```
1 # KD-Tree fuer die schnellen Nachbarschaftsabfragen der Lotpunkte
2     tree = cKDTree(lot_points)
```

Für jeden Punkt der Sensordaten werden die Schnittpunkte im definierten Radius durchsucht. Liegen mehrere Schnittpunkte im Radius, wird derjenige mit der geringsten Distanz ausgewählt. Der Zeitstempel des nächstgelegenen Schnittpunkts wird dem Sensordatensatz zugeordnet und in die Ergebnisliste eingetragen.

```
# Durchlaufe die XYZ-Punkte aus der TXT-Datei und finde die naechstgelegenen Lotpunkte
    for idx, position in enumerate(cmm_positions):
        progress = (idx + 1) / len(cmm_positions) * 100
        print(f"\rBerechne nearest neighbor f r Lotpunkte... Fortschritt: {progress:.2f}%", end="")

        # Finde alle Lotpunkte innerhalb des Radius
        indices = tree.query_ball_point(position, r=radius)

        # Wenn mehrere Punkte innerhalb des Radius liegen, den mit dem kleinsten Abstand auswaehlen
        if indices:
            distances = [np.linalg.norm(lot_points[i] - position) for i in indices]
            nearest_index = indices[np.argmin(distances)]
            nearest_timestamp = timestamps[nearest_index]
            # Den zugehoerigen Timestamp uebernehmen
```

5 Fehlervektor der Werkzeugbahn

Die Funktion `calculate_distances_for_toolpath` stellt den finalen Schritt zur Bestimmung der Abweichungen zwischen den berechneten Lotpunkten und den tatsächlichen Werkzeugbahnpunkten dar. Mit den Ergebnissen der Funktion `find_nearest_neighbors`, berechnet diese Methode die Distanzen zwischen den Sensorkoordinaten mit den Zeitstempeln der nächstgelegenen Lotpunkten und den Punkten der Werkzeugbahn mit ähnlichen Zeitstempel. Gleichzeitig erfolgt eine Auswertung der Bewegungsrichtung des Werkzeugs, um Korrekturwerte für die Werkzeugbahn abzuleiten. Diese Funktion nutzt die Zeitstempel der Daten, um eine präzise Zuordnung zwischen den Lotpunkten und den Werkzeugbahnpunkten herzustellen.

Die Parameter der Funktion sind die Werkzeugbahnpunkte mit Zeitstempeln in einem `Pandas DataFrame` und die Ergebnisliste der `find_nearest_neighbors`-Funktion. Als Rückgabe liefert sie eine Liste von Fehlervektoren, die für jeden relevanten Zeitpunkt die Verschiebung der Werkzeugbahn beschreibt, sodass der gemessene Fehler minimal wird. Die Methode berechnet die Fehlervektoren durch folgende Schritte: Beginnend werden Bewegungsrichtungen entlang der Werkzeugbahn bestimmt, indem die Differenzen zwischen aufeinanderfolgenden Punkten ermittelt werden.

```
1 # Berechnet Bewegungskomponenten (Differenzen zwischen
    aufeinanderfolgenden Punkten in x, y und z)
2 csv_data['movement_direction_x'] = csv_data['x1'].diff()
3 csv_data['movement_direction_y'] = csv_data['y1'].diff()
4 csv_data['movement_direction_z'] = csv_data['z1'].diff()
```

Folgend wird für jeden Punkt ein Zeitfenster definiert, innerhalb dessen die Sensordaten aus den Eingabedaten berücksichtigt werden. Ist kein passender Zeitstempel vorhanden, wird der entsprechende Zeitpunkt ignoriert.

```
# Nearest Neighbor Resultate finden, die innerhalb eines
    Zeitfensters von +-2 ms liegen
    points = nearest_neighbor_results_df[
        (nearest_neighbor_results_df['Timestamp'] >= timestamp-
            2) &
        (nearest_neighbor_results_df['Timestamp'] <= timestamp +
            2)
    ]
```

Um die Abweichungen zwischen den Sensorpunkten und der Werkzeugbahn zu analysieren, wird im nächsten Schritt ein orthogonaler Vektor berechnet. Dieser ergibt sich aus dem Kreuzprodukt der Bewegungsrichtung des Werkzeugs mit der z-Achse. Der orthogonale Vektor dient als Messrichtung bzw. Referenz, um die Abweichung der Sensorpunkte zur Werkzeugbahn zu berechnen.

```
# Senkrechter Richtungsvektor (Kreuzprodukt von Bewegungsrichtung
    mit Z-Achse)
    corr_dir = np.cross(movement_direction, np.array([0, 0, 1]))
```

Die Abweichung zwischen den Sensorpunkten und der orthogonalen Richtung wird anschließend mithilfe des Skalarprodukts berechnet. Das Skalarprodukt quantifiziert, wie stark ein Sensorpunkt in die orthogonale Richtung „verschoben“ ist, indem es die Projektion des Sensorpunktes auf diese Richtung misst.

```
# Berechnung des Skalarprodukts zwischen korrigierter Richtung
    und Lotpunkten
    direction_errors = []
    for _, point in points.iterrows():
        point_position = np.array([point['X'], point['Y'], point[
            'Z']])
        direction_error = np.dot(point_position, corr_dir)
        direction_errors.append(direction_error)
```

Zur weiteren Verarbeitung wird der Mittelwert aller berechneten Abweichungen gebildet.

Dieser Mittelwert wird negiert, um eine Korrektur in die entgegengesetzte Richtung der Abweichung vorzunehmen. Der negierte Wert wird mit dem orthogonalen Vektor multipliziert, um einen dreidimensionalen Korrekturvektor zu erzeugen. Dieser Korrekturvektor beschreibt sowohl die Richtung als auch die Stärke der erforderlichen Anpassung, um die Abweichung zu minimieren.

```
# Mittelwert des Direction Errors berechnen und negieren
    mean_direction_error = -np.mean(direction_errors)
```

Abschließend wird der resultierende Korrekturvektor zusammen mit dem zugehörigen Zeitstempel in einer Ergebnisliste gespeichert. Diese Liste bietet eine strukturierte Übersicht der berechneten Korrekturwerte, die für die zukünftige Optimierung der Werkzeugbahn herangezogen werden können. Die Methode stellt somit sicher, dass sowohl zeitliche als auch geometrische Abhängigkeiten berücksichtigt werden, um die Korrekturvektoren exakt und konsistent zu berechnen.

6 Laufzeitanalyse

Die Laufzeit ist bei der Verarbeitung von Datenmengen bis zu einer Größe von 10^6 Dateneinträgen ist ein nicht zu unterschätzender Faktor. Es gibt mehrere Möglichkeiten, diese zu beeinflussen. Bessere Hardware oder die Wahl einer geeigneten Programmiersprache sind zwei triviale Möglichkeiten. Bei begrenzten Hardwareressourcen und einer festgelegten Programmiersprache durch den Stakeholder bedarf es anderer Lösungen. Mathematische oder algorithmische Ansätze sind für die Problemlösung maßgebend, um die Laufzeit zu verkürzen. Es liegt daher nahe, den Code einer Laufzeitanalyse zu unterziehen, um Laufzeitkomplexitäten zu erkennen und Algorithmen effizienter gestalten zu können. Die Laufzeitkompläxitäten werden im Folgenden in der Landau-Notation formal beschrieben und durch Versuche mit verschiedenen Eingabegrößen geprüft.

6.1 Laufzeitbewertung Nearest-Neighbor-Search

6.1.1 Formale Laufzeitbewertung

Der Code durchläuft den gesamten Sensordatensatz. Das entspricht einer Laufzeit von $O(n)$. Für jeden dieser Punkte n wird eine Abfrage mittels KD-Tree durchgeführt. Eine Suche im KD-Tree entspricht im *worst- case* einer Laufzeit von $O(log(m))$. Im Fall des gefundenen Nachbarn k von Punkt n werden die Abstände zu jedem k-ten Nachbarn mit einer linearen Schleife über die k Nachbarn berechnet. Für die Laufzeit bedeutet das Addieren von k auf den m-ten Punkt im Baum.

Durch Zusammenführen der einzelnen formalen Laufzeitbestandteile können wir auf eine Laufzeit von $O(n((log(m)) + k))$ schließen.

6.1.2 Versuch mit verschiedenen Eingabegrößen

Der Versuch wurde mit drei verschiedenen Größen von jeweils zwei Eingabeparametern durchgeführt. Der Eingabeparameter *Schnittpunkte* wurde mit den Größen 10^3, 10^4 und 10^5 getestet. Abbildung (6.1) zeigt, dass die Laufzeit signifikant von der Größe des Schnittpunkt-Datensatzes (Schnittpunktanzahl) abhängt. Während kleinere Datensätze (Schnittpunktanzahl=1000 und Schnittpunktanzahl=10000) bei zunehmender Eingabedatengröße (TXT-Datei) eine sehr geringe CPU-Rechenzeit aufweisen, steigt die Laufzeit bei einem größeren Datensatz (Schnittpunktanzahl=100000) deutlich an. Dies zeigt sich besonders in der stärkeren Steigung der Laufzeitkurve für (Schnittpunktanzahl=100000).

Darüber hinaus wird ersichtlich, dass das Verfahren eine exponentielle Skalierung zeigt, wenn sowohl die Eingabedatengröße der txt-Datei als auch die Größe des Nearest-Neighbor-Datensatzes steigen. Dies weist darauf hin, dass die Laufzeit beim Nearest-Neighbor-Verfahren durch die kombinierte Größe beider Datensätze bestimmt wird, wobei größere Datensätze einen überproportionalen Einfluss auf die Rechenzeit haben.

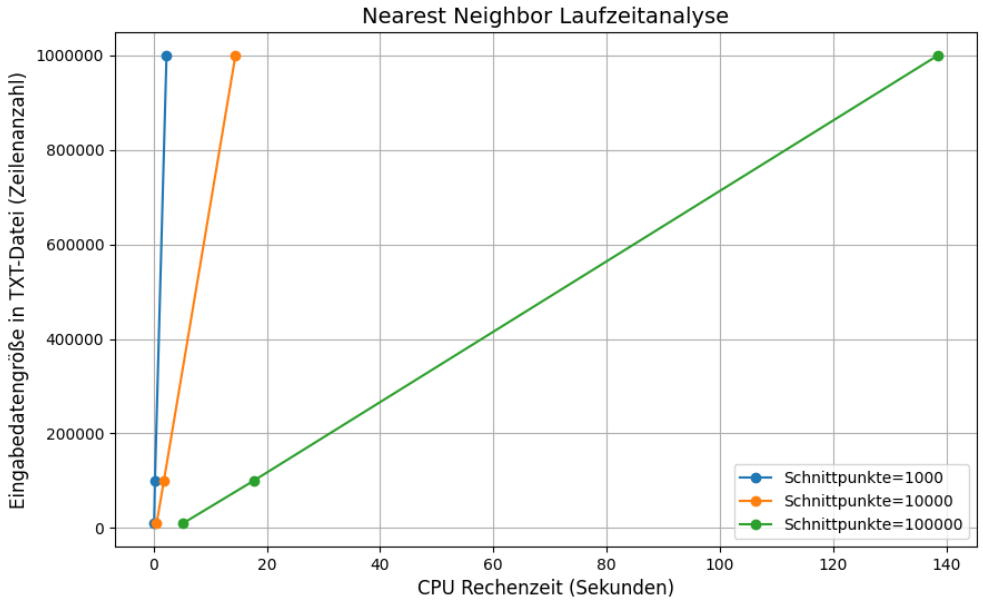

Abbildung 6.1: Nearest Neighbor Algorithmus Laufzeit (CPU Rechenzeit)

6.2 Laufzeitbewertung Toolpath-Distance

6.2.1 Formale Laufzeitanalyse

Der Trace-Datensatz in csv-Format berechnet für jede Zeile mit der Funktion .diff() aus dem pandas-framework die Differenzen zur vorhergehenden Zeile. Die Funktion aus dem pandas-framework durchläuft die Daten intern linear. Das entspricht $O(n)$ für n-Zeilen in der Laufzeitbewertung.

Der Trace-Datensatz wird linear mit einer *for-Schleife* durchlaufen. Das entspricht demnach einer Laufzeit von $O(n)$. Im nächsten Prozessschritt werden für jeden Sensorpunkt passende Zeitstempel in einem Zeitfenster gesucht. Das Durchlaufen der Sensorpunkte entspricht $O(m)$. Der Gesamtdurchlauf hat also für jeden Punkt n, für den m gefunden werden muss, eine Laufzeit von $O(m \cdot n)$. Die Berechnung des Skalarprodukts für die gefilterten Punkte wird für k Punkte durchgeführt. Das entspricht einer linearen Iteration über k Punkte, die n mal ausgeführt wird. Dieser Vorgang wird formal durch $O(n \cdot k)$ beschrieben. Gesamt betrachtet beträgt die Laufzeit dieses Moduls $O(n \cdot (m + k)$.

6.2.2 Versuch mit verschiedenen Eingabegrößen

Die Laufzeitanalyse zeigt in Abbildung (6.2) eine lineare Beziehung zwischen der Eingabegröße der csv-Datei (Zeilenanzahl) und der CPU-Rechenzeit (in Sekunden). Unabhängig von der Größe des Toolpath-Datasets (Lotpunktanzahl=1000, Lotpunktanzahl=10000, Lotpunktanzahl=100000) bleibt die Zunahme der Rechenzeit fast proportional zur Eingabegröße. Das deutet darauf hin, dass der Algorithmus für die Berechnung der Toolpath-Distance eine Zeitkomplexität besitzt, die direkt mit der Eingabegröße skaliert.

Besonders auffällig ist, dass die Steigung der Laufzeitkurven für die unterschiedlichen Toolpath-Datensätze nahezu identisch bleibt, was darauf hinweist, dass die Größe des Toolpath-Datensatzes nur einen minimalen Einfluss auf die Rechenzeit hat. Dies bestätigt, dass die Hauptlaufzeit durch die Anzahl der csv-Eingabezeilen bestimmt wird, während die Größe des Toolpath-Datensatzes eine untergeordnete Rolle spielt.

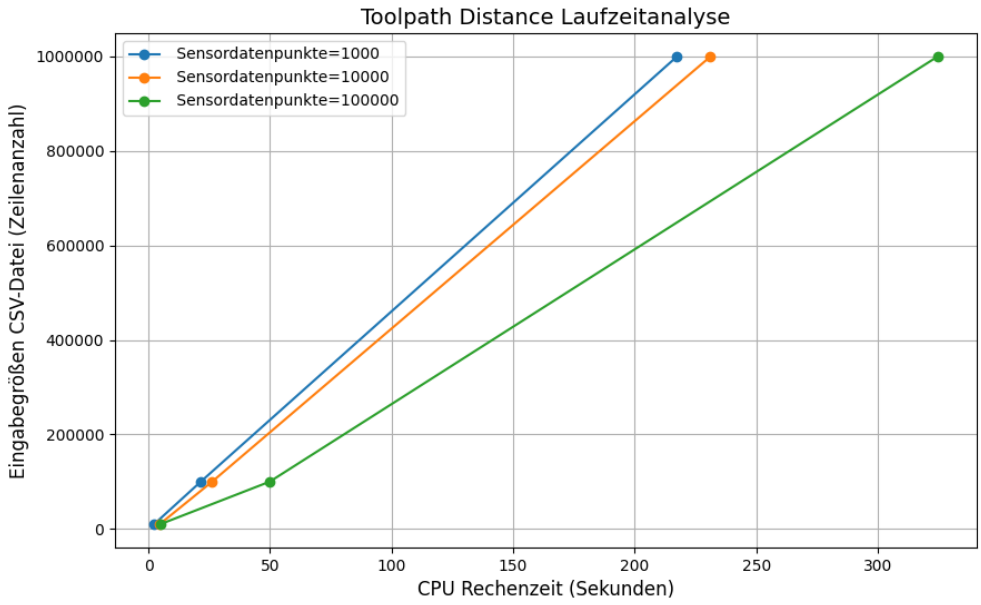

Abbildung 6.2: Toolpath-Distance Berechnungsverfahren Laufzeit (CPU Rechenzeit)

7 Ergebnis

Ziel dieser Seminararbeit war es ein Tool zur Analyse und Optimierung von Werkzeugbahnen zu entwickeln. In der Arbeit wurde gezeigt, dass durch bekannte in der Mathematik und Informatik etablierte Methoden ein komplexes Problem durch Abstrahierung auf fundamentale Konzepte zurückgeführt und gelöst werden kann. Herausfordernd war der Prozess des Verstehens der einzelnen Auswertungsschritte von Dexel bis zur Werkzeugposition in Abhängigkeit von der Zeit. Durch aufteilen des Problems in seine Kernbestandteile konnte die Herangehensweise ausgearbeitet werden.

7.1 Nächster Nachtbar

Der in Kapitel (4) vorgestellte Nächster Nachbar Algorithmus welcher in der Funktion *find_nearest_neighbors(results, txt_allData)* realisiert wurde verknüpft auf effiziente Weise Simulations- und Sensordaten. Dies ermöglicht eine zeitliche und räumliche Zuordnung zwischen den beiden Datenquellen, was für die spätere Auswertung der Werkzeugbahn von entscheidender Bedeutung ist.

Zusammenfassend mit Kapitel (6.1) lässt sich so feststellen, dass das Nearest-Neighbor-Verfahren für kleinere Datensätze effizient arbeitet, jedoch bei größeren Eingabegrößen, insbesondere der Schnittpunkte, eine optimierte Implementierung erforderlich ist, um die Laufzeit zu minimieren. Der in dieser Arbeit entwickelte und implementierte Algorithmus stellt mit einer Laufzeit von etwa 140 Sekunden bei einer Eingabegröße von 10^5 Schnittpunkten, welche als realistischer Wert angenommen werden kann, eine praxistaugliche Lösung vor.

7.2 Werkzeugbahn Distanz

Die in Kapitel (5) beschriebene Methode zur Berechnung der Korrekturvektoren der Werkzeugbahn ermöglicht eine präzise Analyse der Abweichungen zwischen Sensordaten und Werkzeugbahnpunkten. Sie berechnet Bewegungsrichtungen entlang der Werkzeugbahn durch Differenzen aufeinanderfolgender Punkte und berücksichtigt zeitliche Zusammenhänge mittels definierter Zeitfenster. Basierend auf diesen Daten erzeugt die Methode dreidimensionale Korrekturvektoren, die sowohl die Richtung als auch die Stärke der notwendigen Anpassungen beschreiben.

Die Analyse aus Kapitel (6.2) zeigt, dass der Algorithmus zur Berechnung der Toolpath-Distance robust und effizient auf große Eingabemengen reagiert. Die Laufzeit steigt linear mit der Anzahl der Werkzeugmesspunktdaten (csv-Zeilen), was auf eine solide algorithmische Umsetzung hinweist. Dies bestätigt, dass der Algorithmus bei großen Datensätzen stabil bleibt und keine exponentiellen Laufzeitsteigerungen verursacht.

Gleichzeitig wird deutlich, dass die Rechenzeit stark von der Anzahl der Werkzeugmesspunkte abhängt, da jede Zeile der csv-Datei iterativ verarbeitet wird. Eine Erhöhung der Werkzeugbahnmessdaten wirkt sich daher stärker auf die Laufzeit aus als eine vergleichbare Vergrößerung der Sensordaten aus der Nearest-Neighbor-Search. Für Anwendungen mit sehr großen Datensätzen wäre eine zusätzliche Optimierung sinnvoll, um die Effizienz weiter zu verbessern.

Trotz dieser Einschränkung überzeugt der Algorithmus durch eine hohe Leistungsfähigkeit und bietet eine verlässliche Grundlage für die Berechnung der Werkzeugbahndistanz.

8 Zusammenfassung und Ausblick

Diese Arbeit bildet eine Grundlage für die Auswertung der Werkzeugbahnen von CNC-Fräsmaschinen und bietet damit eine Basis für weiterführende Entwicklungen in diesem Bereich. Der Fokus lag auf der Implementierung funktionaler Anforderungen zur Analyse und Korrektur von Werkzeugbahnen, wobei eine präzise und effiziente Zuordnung von Messpunkten und die Berechnung des Korrekturvektors im Mittelpunkt stand. Der Korrekturvektor bildet das Fundament, um aufbauend auf dieser Arbeit eine gesamte Bahnplanung einer Werkzeugmaschine zu realisieren.

Durch Weiterentwicklung und Optimierung des hier vorgestellten Suchalgorithmus und die weiterführende gezielte Vorsortierung der Daten, beispielsweise durch die Einführung zusätzlicher Offsets, kann die Rechenzeit weiter reduziert werden. Solche Verbesserungen würden es ermöglichen, das Tool effizienter zu machen und längere Wartezeiten zu vermeiden.

Ein weiterer wichtiger Aspekt ist die Verbesserung der Nutzerfreundlichkeit durch die Entwicklung einer geeigneten Benutzeroberfläche. Diese würde die Anwendung des Tools auch für nicht-technische Anwender erleichtern und die Praxistauglichkeit des Tools steigern.

9 Literaturverzeichnis

[NumInt2019] URL: https://www.python4data.science/de/latest/workspace/numpy/intro.html (besucht am 12. 12. 2024).

[NumArr2019] URL: https://www.python4data.science/de/latest/workspace/numpy/vectorisation.html (besucht am 12. 12. 2024).

[Histrov2023] Hristo Baeldung. *Introduction to K-D Trees*. Last updated: 2023-12-05. n.d. URL: https://www.baeldung.com/cs/k-d-trees (besucht am 12. 12. 2024).

[Boess2021] Volker Böß und Berend Denkena und Marc-André Dittrich und Talash Malek und Sven Friebe. "Dexel-Based Simulation of Directed Energy Deposition Additive Manufacturing". In: *Journal of Manufacturing and Materials Processing* 5.1 (Jan. 2021). Open-Access-Artikel, der unter den Bedingungen der Creative-Commons-Attribution (CC BY) Lizenz veröffentlicht wurde, S. 9. DOI: 10.3390/jmmp5010009. URL: https://www.mdpi.com/2504-4494/5/1/9.

[Sharp] Sharp Coder Blog. *How to Optimize Python Code for Performance*. Zugriff am 6. Januar 2025. URL: https://de.sharpcoderblog.com/blog/how-to-optimize-python-code-for-performance#google_vignette.

[Brecher2018] Christian Brecher und Manfred Weck. *Werkzeugmaschinen und Fertigungssysteme 1*. 3. Aufl. NRW, Aachen: Springer, Aug. 2018. ISBN: 978-3-662-46565-3.

[Trimesh] Trimesh Developers. *Trimesh: A Python library for loading and using triangular meshes*. Zugriff am 6. Januar 2025. URL: https://trimsh.org/.

[DIN8580] *DIN 8580: Fertigungsverfahren — Einteilung, Begriffe*. Aktuelle Ausgabe zum Zeitpunkt der Erstellung. Berlin, 2003.

[NumPy2020] Charles R. Harris u. a. "Array programming with NumPy". In: *Nature* 585.7825 (Sep. 2020), S. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[Jung2024] Michael Jung. *Analytische Geometrie in der Ebene*. Berlin, Heidelberg: Springer, Nov. 2024. ISBN: 978-3-658-03262-3.

[Newmann2008] S.T. Newman u. a. "Strategic advantages of interoperability for global manufacturing using CNC technology". In: *Robotics and Computer-Integrated Manufacturing* 24.6 (2008), S. 699–708. DOI: 10.1016/j.rcim.2008.03.002. URL: https://www.sciencedirect.com/science/article/pii/S0736584508000459.

[Ottmann2017] Thomas Ottmann und Peter Wildmayer. *Algorithmen und Datenstrukturen*. 6. Aufl. Freiburg, Zürich: Springer, Apr. 2017. ISBN: 978-3-662-55650-4.

[Taunk2019] Kashvi Taunk und Sanjukta De und Srishti Verma und Aleena Swetapadma. "A Brief Review of Nearest Neighbor Algorithm for Learning and Classification". In: *Proceedings of the International Conference on Intelligent Computing and Control Systems (ICICCS 2019)*. Open-Access-Artikel, veröffentlicht unter den Bedingungen der IEEE-Lizenz. Bhubaneswar, India: IEEE, Juli 2019, S. 1255–1260. ISBN: 978-1-5386-8113-8. DOI: 10.1109/ICICCS.2019.9065330.

[Pandas] The pandas development team. *pandas: Python Data Analysis Library*. Zugriff am 6. Januar 2025. 2025. URL: https://pandas.pydata.org/.

[MATLAB1994] Inc. The MathWorks. *MATLAB Documentation*. Zugriff am 6. Januar 2025. 1994. URL: https://www.mathworks.com.

Abkürzungsverzeichnis

CNC : Computerized Numerical Control

WZL : Werkzeugmaschinen Labor

RWTH: Rheinisch-Westfälische Technische Hochschule

STL: Standard-Triangle-Language

Abbildungsverzeichnis

Anhang

```python
from data_loader import load_stl, load_csv_data, load_txt_data, load_all_txt_files
from dexel_analysis import calculate_dexel_errors, calculate_dexel_point
from cmm_analysis import find_nearest_neighbors
from toolpath_distance import calculate_distances_for_toolpath
from parser import dwp_parser
from plot_errors_on_stl import plot_errors_on_stl
from plot_dwp import plot_dwp
from testing import test_dexel, test_data
import numpy as np
import time

def main():
    #Laufzeitanalyse

    print("Starting CNC Defect Analysis...")

    # Datei-Paths
    dwp_file = "C:/Users/Florian Altdorf/Desktop/FH Aachen/Semester 5/Seminararbeit/CNC_DefectAnalysis/data/Bauteil.dwp"
    stl_file = "C:/Users/Florian Altdorf/Desktop/FH Aachen/Semester 5/Seminararbeit/CNC_DefectAnalysis/data/MusterLehmo_DALL.STL"
    csv_file = "C:/Users/Florian Altdorf/Desktop/FH Aachen/Semester 5/Seminararbeit/CNC_DefectAnalysis/data/trace_29965569_Measurements.csv"
    txt_file = "C:/Users/Florian Altdorf/Desktop/FH Aachen/Semester 5/Seminararbeit/CNC_DefectAnalysis/data/D16_1.1.txt"
    directory_path_txt = "C:/Users/Florian Altdorf/Desktop/FH Aachen/Semester 5/Seminararbeit/CNC_DefectAnalysis/data/Swimmingpool_1"

    # 1. Dexel-Daten aus der DWP-Datei laden
    cells, meta = dwp_parser(dwp_file)
    print("DWP-Daten gela den.")
    np.save('cells', cells)

    # 2. STL-Daten laden
    vertices, faces = load_stl(stl_file)
    print("STL-Daten geladen.")
    np.save('vertices', vertices)
    np.save('faces', faces)

    # 3. CSV und TXT laden
    csv_table = load_csv_data(csv_file)

    #lädt alle txt files
    txt_allData = load_all_txt_files(directory_path_txt)
    print(txt_allData)
    txt_table = load_txt_data(txt_file)

    # 4. Dexel-Punkte berechnen
    dexel_points = calculate_dexel_point(cells)

    # 5. Dexel-Fehleranalyse durchführen
    results = calculate_dexel_errors(dexel_points, vertices, faces)
    print("Dexel-Fehleranalyse abgeschlossen.")
    print("Erstes Ergebnis:", results[0])

    np.save('Schnittpunkte', results)


    # 6. Nearest Neighbor für Lotpunkte bestimmen
    nearest_neighbor_result = find_nearest_neighbors(results, txt_allData, 1)
    print("Timestamps zu Lotpunkten zugeordnet")
    print("Erste Struktur-Ausgabe:", nearest_neighbor_result[0])

    np.save('Lotpunkte', nearest_neighbor_result)

    # Filtere nur Einträge mit gültigen Zeitstempeln. Ungültige werden zu testzwecken ignoriert
    entries_with_timestamp = [
        entry for entry in nearest_neighbor_result
        if entry['Timestamp'] is not None and entry['X'] is not None
        and entry['Y'] is not None and entry['Z'] is not None
]
```

```
67      # 7. Berechne Distanz zum Werkzeug
68
69      if entries_with_timestamp:
70          distance_results = calculate_distances_for_toolpath(entries_with_timestamp,
            csv_table)
71          print("Abstände zur Werkzeugbahn berechnet.")
72      else:
73          print("Keine Einträge mit gültigen Zeitstempeln gefunden.")
74          distance_results = []
75
76      np.save('Korrekturvektoren', distance_results)
77      #Zwischenspeicher
78      #np.savetxt('..\data\datenspeicher', delimiter=';')
79
80      # Ausgabe zur Überprüfung1  ""
81      print("\nErgebnisse der Abstandsmessungen:")
82      for i, result in enumerate(distance_results[:20]):
83          print(f"\nErgebnis {i+1}: {result}")
84      else:
85          print("Keine gültigen Zeitstempel für Distanzberechnung gefunden.")
86
87      # Ausgabe der ersten 20 Einträge mit Zeitstempeln
88      print("\nEinträge mit Zeitstempeln:")
89      for i, entry in enumerate(entries_with_timestamp[:20]):
90          print(f"\nEintrag {i+1}: {entry}")
91
92      if not entries_with_timestamp:
93          print("Keine Einträge mit Zeitstempeln gefunden.")
94      else:
95          print(f"Insgesamt {len(entries_with_timestamp)} Einträge mit Zeitstempeln
            gefunden.")
96
97      # 8. Plot
98
99
100     #plot_errors_on_stl(stl_file, dexel_points, results, distance_results) #plot
        dexel_points
101
102 if __name__ == "__main__":
103     main()
104
105
```

```python
import numpy as np
import trimesh
import pandas as pd
import os


def load_stl(filename):
    """
    Lädt die STL-Datei und gibt die Vertices und Faces zurück.
    """
    print(f"Lade STL-Datei: {filename}")
    mesh = trimesh.load(filename)
    vertices = np.array(mesh.vertices)
    faces = np.array(mesh.faces)
    return vertices, faces

# CSV-Datei laden
def load_csv_data(csv_file_path):
    """
    Lädt Sensordaten aus einer CSV-Datei.

    Parameter:
    - csv_file_path: Der Pfad zur CSV-Datei

    Rückgabe:
    - DataFrame mit den Sensordaten
    """
    # liest zu testzwecken nur 1000 zeilen ein
    csv_data = pd.read_csv(csv_file_path, delimiter = ';')
    return csv_data

def load_txt_data(txt_file):
    """
    Lädt Sensordaten aus einer CSV-Datei.

    Parameter:
    - csv_file_path: Der Pfad zur CSV-Datei

    Rückgabe:
    - DataFrame mit den Sensordaten
    """
     # liest zu testzwecken nur 1000 zeilen ein
    columns = ['x','y','z','nx','ny','nz','ex','ey','ez']
    txt_data= pd.read_csv(txt_file, delimiter = ' ', header=None, names=columns)
    return txt_data


def load_all_txt_files(directory_path):
    """
    Lädt alle `.txt`-Dateien in einem Verzeichnis und speichert die Daten in einer Liste von DataFrames.

    Parameter:
    - directory_path: Der Pfad zum Verzeichnis mit `.txt`-Dateien.

    Rückgabe:
    - Eine Liste von DataFrames mit den Daten aus den `.txt`-Dateien.
    """
    combined_data = pd.DataFrame()  # Liste, um alle geladenen Daten zu speichern

    # Durchlaufe alle Dateien im Verzeichnis
    for file_name in os.listdir(directory_path):
        if file_name.endswith('.txt'):  # Nur `.txt`-Dateien berücksichtigen
            file_path = os.path.join(directory_path, file_name)
            try:
                # Lese die `.txt`-Datei ein
                columns = ['x','y','z','nx','ny','nz','ex','ey','ez']
                txt_data= pd.read_csv(file_path, delimiter = ' ', header=None, names=columns)
                combined_data = pd.concat([combined_data, txt_data], ignore_index=True)
                print(f"{file_name} erfolgreich geladen.")
```

```
70                 except Exception as e:
71                     print(f"Fehler beim Laden von {file_name}: {e}")
72 
73         return combined_data
```

```python
import sys
import numpy as np
import pandas as pd

def dwp_parser(filename):
    """
    Lädt die DWP-Datei und gibt die Zellen (cells) und Metadaten (meta) zurück.
    """
    with open(filename, 'r') as fh:
        # Überprüfe, ob die erste Zeile korrekt ist
        first_line = fh.readline().strip()
        if first_line != 'CHUNK_DEXEL_MODEL':
            raise ValueError("expected CHUNK_DEXEL_MODEL")

        # Meta-Daten einlesen
        meta = {
            'StepSize': float(fh.readline().strip()),
            'Resolution': float(fh.readline().strip())
        }

        cells = []
        num_cells = 0
        total_cellnum = 0
        # Einlesen der Zellen für jede Richtung (X, Y, Z)
        for i in range(3):
            current_direction = chr(ord('X') + i)
            num_cells_in_direction = int(fh.readline().strip())

            for _ in range(num_cells_in_direction):

                ####### max_cells um nur einen bruchteil einzulesen zum testen
                # if max_cells is not None and num_cells >= max_cells:
                #     break

                num_cells += 1
                cell = parse_cell(fh, total_cellnum)
                total_cellnum += 1
                cells.append(cell)

    print("Raw Array Shape:", cell['Raw'].shape)
    print("Values DataFrame:")
    print(cell['Values'].head())  # Zeigt die ersten paar Zeilen des DataFrames an
    print(cell['Values'].columns)  # Zeigt die Spaltennamen des DataFrames an

    # max = sys.float_info.min

    # for i in cells:

    #     point = i['values']['']

    #     if i > max:
    #         max = i

    return cells, meta


def parse_cell(fh, cell_id):
    """
    Parst eine einzelne Zelle aus der Datei und gibt ein Dictionary zurück.
    """
    cell = {
        'Direction': float(fh.readline().strip()),
        'NumberOfDexelValues': int(fh.readline().strip()),
        'MinVectorOfDexelCell': np.fromstring(fh.readline().strip(), sep=' '),
        'MaxVectorOfDexelCell': np.fromstring(fh.readline().strip(), sep=' '),
        'CenterVectorOfDexelCell': np.fromstring(fh.readline().strip(), sep=' '),
        'TextureOffset': np.fromstring(fh.readline().strip(), sep=' '),
        'TextureScale': np.fromstring(fh.readline().strip(), sep=' '),
        'DirectionMagnitudeU': np.fromstring(fh.readline().strip(), sep=' '),
        'DirectionMagnitudeV': np.fromstring(fh.readline().strip(), sep=' '),
        'DirectionMagnitudeW': np.fromstring(fh.readline().strip(), sep=' '),
        'NumberOfDexelsInDexelCell': int(fh.readline().strip()),
```

```
            'cell_id': cell_id
        }

        # Anzahl der Spalten in der Tabelle berechnen
        num_columns = 7 + cell['NumberOfDexelValues'] + 5 + cell['NumberOfDexelValues']

        # Werte für die Dexels einlesen
        if cell['NumberOfDexelsInDexelCell'] > 0:
            values = np.loadtxt([fh.readline().strip() for _ in range(cell[
            'NumberOfDexelsInDexelCell'])])
            if values.ndim == 1:
                values = values[np.newaxis, :]  # Falls nur eine Zeile vorhanden ist

            # Spalten der Werte zuweisen
            cell['Raw'] = values
            cell['Values'] = pd.DataFrame({
                'TextureU': values[:, 0],
                'TextureV': values[:, 1],
                'StartDepth': values[:, 2],
                'StartTime': values[:, 3],
                'StartNormalX': values[:, 4],
                'StartNormalY': values[:, 5],
                'StartNormalZ': values[:, 6],
                'EndDepth': values[:, 7 + cell['NumberOfDexelValues']],
                'EndTime': values[:, 7 + cell['NumberOfDexelValues'] + 1],
                'EndNormalX': values[:, 7 + cell['NumberOfDexelValues'] + 2],
                'EndNormalY': values[:, 7 + cell['NumberOfDexelValues'] + 3],
                'EndNormalZ': values[:, 7 + cell['NumberOfDexelValues'] + 4]
            })

            if any(cell['Values']['EndDepth'] > 10):
                pass

        else:
              cell['Values'] = pd.DataFrame(columns=[
                'TextureU', 'TextureV', 'StartDepth', 'StartTime',
                'StartNormalX', 'StartNormalY', 'StartNormalZ',
                'EndDepth', 'EndTime', 'EndNormalX', 'EndNormalY', 'EndNormalZ'
            ])


        return cell

```

```python
import numpy as np
import pyvista as pv

def calculate_dexel_errors(cells, vertices, faces):
    """
        Berechnet die Schnittpunkte zwischen Dexel-Linien und Dreiecksflächen.
        Parameter:
        ----------
        cells : numpy.ndarray
            Dexel-Daten mit Start-/Endpunkten und Zeitstempeln.
        vertices : numpy.ndarray
            Eckpunkte des 3D-Modells.
        faces : numpy.ndarray
            Dreiecksflächen des Modells.

        Rückgabewert:
        -------------
        results : list of dict
            Schnittpunkte mit Koordinaten und Zeitstempeln.

        Beschreibung:
        -------------
        Filtert relevante Dreiecke anhand räumlicher Ausdehnung und berechnet
        die nächsten Schnittpunkte der Dexel-Linien mit den Dreiecken.

    """
    results = []
    cell_ids = cells[:, 8]
    cell_IDs = np.unique(cell_ids)
    # Bereite Dreiecksflächen nur einmal vor
    v0 = vertices[faces[:, 0], :]
    v1 = vertices[faces[:, 1], :]
    v2 = vertices[faces[:, 2], :]

    triMinXs = np.min(np.vstack([v0[:, 0], v1[:, 0], v2[:, 0]]).T, axis=1)
    triMaxXs = np.max(np.vstack([v0[:, 0], v1[:, 0], v2[:, 0]]).T, axis=1)
    triMinYs = np.min(np.vstack([v0[:, 1], v1[:, 1], v2[:, 1]]).T, axis=1)
    triMaxYs = np.max(np.vstack([v0[:, 1], v1[:, 1], v2[:, 1]]).T, axis=1)
    triMinZs = np.min(np.vstack([v0[:, 2], v1[:, 2], v2[:, 2]]).T, axis=1)
    triMaxZs = np.max(np.vstack([v0[:, 2], v1[:, 2], v2[:, 2]]).T, axis=1)

    start_points = cells[:, 0:3]
    end_points = cells[:, 3:6]
    timestamps_start = cells[:, 6]
    timestamps_end = cells[:, 7]

    # Berechne die erweiterten Start- und Endpunkte der Dexel nur einmal
    modified_dexel_start, modified_dexel_end = extend_dexel(start_points, end_points)

    for idx, cell_ID in enumerate(cell_IDs):
        progress = (idx + 1) / len(cell_IDs) * 100
        print(f"\rBerechne Dexel-Fehler und Lotpunkte... Fortschritt: {progress:.2f}%"
        , end="")

        selector = cell_ID == cell_ids

        dexel_min_xs = np.min([modified_dexel_start[selector, 0], modified_dexel_end[
        selector, 0]])
        dexel_max_xs = np.max([modified_dexel_start[selector, 0], modified_dexel_end[
        selector, 0]])
        dexel_min_ys = np.min([modified_dexel_start[selector, 1], modified_dexel_end[
        selector, 1]])
        dexel_max_ys = np.max([modified_dexel_start[selector, 1], modified_dexel_end[
        selector, 1]])
        dexel_min_zs = np.min([modified_dexel_start[selector, 2], modified_dexel_end[
        selector, 2]])
        dexel_max_zs = np.max([modified_dexel_start[selector, 2], modified_dexel_end[
        selector, 2]])

        # Filtere mögliche Dreiecke für diesen Dexel
        possible_triangles = (
            (dexel_max_xs >= triMinXs) & (dexel_min_xs <= triMaxXs) &
```

```
            (dexel_max_ys >= triMinYs) & (dexel_min_ys <= triMaxYs) &
            (dexel_max_zs >= triMinZs) & (dexel_min_zs <= triMaxZs)
        )

        possible_V0 = v0[possible_triangles]
        possible_V1 = v1[possible_triangles]
        possible_V2 = v2[possible_triangles]

        nearest_start = None
        nearest_end = None

        for idx in np.where(selector)[0]:

    # Vektorisierte Berechnung aller möglichen Schnittpunkte
            if len(possible_V0) > 0:
                intersects, valid_intersects = line_plane_intersection_batch(
                possible_V0, possible_V1, possible_V2,
                modified_dexel_start[idx], modified_dexel_end[idx]
            )

            inside_mask = inside_triangle_batch(possible_V0, possible_V1, possible_V2,
             intersects)

            # überspringt die leeren Masken
            if inside_mask.shape[0]== 0:
                break

            # Filtert nur die Punkte innerhalb des Dreiecks
            valid_intersections = intersects[inside_mask & valid_intersects]

            if len(valid_intersections) > 0:
                # Berechne alle Abstände und wähle den nächsten Punkt
                distances_start = np.linalg.norm(valid_intersections -
                modified_dexel_start[idx], axis=1)
                distances_end = np.linalg.norm(valid_intersections -
                modified_dexel_end[idx], axis=1)

                nearest_start = valid_intersections[np.argmin(distances_start)]
                nearest_end = valid_intersections[np.argmin(distances_end)]

            # Speichere die Ergebnisse
            if nearest_start is not None and nearest_end is not None:
                results.append({
                    "X": nearest_start[0], "Y": nearest_start[1], "Z": nearest_start[2
                    ], "Timestamp": timestamps_start[idx]
                })
                results.append({
                    "X": nearest_end[0], "Y": nearest_end[1], "Z": nearest_end[2],
                    "Timestamp": timestamps_end[idx]
                })

    print("\nBerechnung abgeschlossen.")

    return results

def calculate_dexel_point(cells):
    """
    Berechnet die Start- und Endpunkte für jeden Dexel basierend auf der
    `cells`-Struktur und
    ordnet die zugehörigen Zeitstempel zu.

    Parameter:
    - cells: Liste von Dictionaries mit Dexel-Informationen.

    Rückgabe:
    - Xi: Liste mit den Start- und Endpunkten sowie den Zeitstempeln für jeden Dexel.
    """
    Xi = []
    total_cells = len(cells)

    for idx, cell in enumerate(cells):
        progress = (idx + 1) / total_cells * 100
```

```
132             print(f"\rBerechne Dexel-Punkte... Fortschritt: {progress:.2f}%", end="")
133 
134             # Überprüfen, ob die Zelle Werte enthält
135             if "Values" not in cell or cell["Values"].empty:
136                 continue
137 
138             # Extrahiere die notwendigen Werte aus der Zellenstruktur
139             min_vector = cell["MinVectorOfDexelCell"]
140             texture_u = cell["Values"]["TextureU"]
141             texture_v = cell["Values"]["TextureV"]
142             start_depth = cell["Values"]["StartDepth"]
143             end_depth = cell["Values"]["EndDepth"]
144             direction_u = cell["DirectionMagnitudeU"]
145             direction_v = cell["DirectionMagnitudeV"]
146             direction_w = cell["DirectionMagnitudeW"]
147             cell_id = np.repeat(cell["cell_id"], len(texture_u), axis = 0)
148             cell_id.shape = (len(texture_u), 1)
149             # Zeitstempel für Start und Ende
150             start_time = cell["Values"]["StartTime"]
151             end_time = cell["Values"]["EndTime"]
152 
153             if len(end_time) ==  0:
154                 print(f'Cell {idx} is empty')
155 
156             # Berechne die Startposition
157             start_pos = (
158                 min_vector
159                 + np.outer(texture_u, direction_u)
160                 + np.outer(texture_v, direction_v)
161                 + np.outer(start_depth, direction_w)
162             )
163 
164             # Berechne die Endposition
165             end_pos = (
166                 min_vector
167                 + np.outer(texture_u, direction_u)
168                 + np.outer(texture_v, direction_v)
169                 + np.outer(end_depth, direction_w)
170             )
171 
172             # plotter = pv.Plotter()
173 
174             # plotter.add_points(
175             #     np.array(start_pos), render_points_as_spheres=True, point_size=5
176             # )
177             # plotter.add_points(
178             #     np.array(end_pos), render_points_as_spheres=True, point_size=5
179             # )
180             # plotter.show_axes()
181             # plotter.show_grid()
182             # plotter.show()
183 
184 
185             # Kombiniere Start- und Endpositionen mit Zeitstempeln
186             combined_points = np.hstack([start_pos, end_pos])
187 
188             # Füge Zeitstempel als separate Spalten hinzu
189             start_time_column = np.array(start_time).reshape(-1, 1)
190             end_time_column = np.array(end_time).reshape(-1, 1)
191 
192             # Erstelle die vollständige Struktur mit Start- und Endpositionen und
                Zeitstempeln
193             full_data = np.hstack([combined_points, start_time_column, end_time_column,
                cell_id])
194 
195             # Speichere die Daten in Xi
196             Xi.append(full_data)
197 
198         # Stapel alle Ergebnisse zusammen, um eine Gesamtstruktur zu erhalten
199         Xi = np.vstack(Xi)
200 
201 
```

```
202         return Xi
203
204     ########todo milimeter vergrößern
205     def extend_dexel(start_dexel, end_dexel, millimeter=10):
206         """
207         Erweitert einen Dexel um eine vorgegebende Länge in Millimetern.
208
209         Parameter:
210         - start_dexel: Startpunkt des Dexels (np.array).
211         - end_dexel: Endpunkt des Dexels (np.array).
212         - millimeter: Erweiterungslänge (Standardwert: 0.5 mm).
213
214         Rückgabe:
215         - extended_start_dexel: Erweiterter Startpunkt.
216         - extended_end_dexel: Erweiterter Endpunkt.
217         """
218         # Richtung des Dexels berechnen
219         direction = end_dexel - start_dexel
220         norm_direction = direction / np.linalg.norm(
221             direction, axis = 1  ###############Performance alle vektot Operationen auf
                einmal machen
222         )[:, np.newaxis]  # Normalisieren des Richtungsvektors
223
224         # Start- und Endpunkte erweitern
225         extended_start_dexel = start_dexel - norm_direction * millimeter
226         extended_end_dexel = end_dexel + norm_direction * millimeter
227
228         return extended_start_dexel, extended_end_dexel
229
230
231     def line_plane_intersection_batch(A, B, C, P1, P2):
232         """
233         Berechnet den Schnittpunkt einer Linie (definiert durch P1 und P2)
234         mit einer Ebene (definiert durch die Punkte A, B und C) für mehrere Dreiecke
            gleichzeitig.
235
236         Parameter:
237         - A, B, C: Arrays von 3D-Punkten, die die Ecken der Dreiecke definieren (Shape:
            (n, 3))
238         - P1, P2: 3D-Start- und Endpunkte der Linie (jeweils Shape: (1, 3) oder (3,))
239
240         Rückgabe:
241         - intersection_points: Der Schnittpunkt der Linie und der Ebene für jedes Dreieck
            (Shape: (n, 3))
242         - valid_mask: Boolean Array, das anzeigt, ob der Schnittpunkt gültig ist
243         """
244         # Berechne Vektoren der Dreiecke
245         vector1 = B - A  # Shape: (n, 3)
246         vector2 = C - A  # Shape: (n, 3)
247         N = np.cross(vector1, vector2)  # Normalenvektor der Ebene, Shape: (n, 3)
248         norm_N = np.linalg.norm(N, axis=1, keepdims=True)  # Norm des Normalenvektors,
            Shape: (n, 1)
249         n = N / norm_N  # Normierter Normalenvektor, Shape: (n, 3)
250
251         # Richtung der Linie berechnen
252         direction = (P2 - P1).reshape(1, 3)  # Shape: (1, 3) für einfaches Broadcasting
253
254
255         # Berechnung Parameter k für den Schnittpunkt
256         denominator = np.einsum('ij,ij->i', n, direction)  # Skalarprodukt, Shape: (n,)
257         numerator = np.einsum('ij,ij->i', n, (A - P1))  # Skalarprodukt, Shape: (n,)
258
259         # Vermeidung Division durch Null
260         valid_mask = np.abs(denominator) > 1e-6
261         k = np.zeros_like(numerator)
262         k[valid_mask] = numerator[valid_mask] / denominator[valid_mask]
263
264         # Berechnung Schnittpunkte nur für gültige Dreiecke
265         intersection_points = P1 + k[:, np.newaxis] * direction  # Shape: (n, 3)
266
267         return intersection_points, valid_mask
268
```

```python


def inside_triangle_batch(A, B, C, S):

    if A.shape[0] == 0:
    # Keine gültigen Schnittpunkte vorhanden, überspringe die Berechnung
        return np.zeros(A.shape[0], dtype=bool)  # Gibt eine Maske mit False zurück

    v0 = C - A
    v1 = B - A
    v2 = S - A

    dot00 = np.einsum("ij,ij->i", v0, v0)
    dot01 = np.einsum("ij,ij->i", v0, v1)
    dot02 = np.einsum("ij,ij->i", v0, v2)
    dot11 = np.einsum("ij,ij->i", v1, v1)
    dot12 = np.einsum("ij,ij->i", v1, v2)

    inv_denom = 1.0 / (dot00 * dot11 - dot01 * dot01)
    u = (dot11 * dot02 - dot01 * dot12) * inv_denom
    v = (dot00 * dot12 - dot01 * dot02) * inv_denom

    return (u >= 0) & (v >= 0) & (u + v <= 1)

```

```python

import numpy as np
from scipy.spatial import cKDTree
import pandas as pd
import time

def find_nearest_neighbors(perpendicular_points, txt_table, radius):
    """
    Verwendet das Nearest-Neighbor-Verfahren mit Radiusbeschränkung, um die nächstgelegenen
    XYZ-Punkte aus der TXT-Datei den berechneten Lotpunkten zuzuordnen und die vorhandenen Timestamps
    aus den Lotpunkten weiterzugeben.

    Parameter:
    - perpendicular_points: Liste der Ergebnisse von `calculate_dexel_errors`, enthält Lotpunkte und Timestamps.
    - txt_table: Pandas DataFrame mit den XYZ-Koordinaten.
    - radius: Der maximale Abstand in mm, in dem ein Sensorpunkt zu einem Lotpunkt gefunden werden soll.

    Rückgabe:
    - Eine Liste mit Strukturen, die Lotpunkte, zugeordnete XYZ-Koordinaten und die ursprünglichen Timestamps enthalten.
    """
    #Laufzeitanalyse
    #start_time_NN = time.time()
    #cpu_start_NN = time.process_time()
    counter = 1
    # Extrahiere die Sensorpositionen aus der TXT-Datei (XYZ-Koordinaten)
    cmm_positions = txt_table[['x', 'y', 'z']].values

    # Extrahiere die Lotpunkte-Koordinaten und Timestamps aus perpendicular_points
    lot_points = np.array([(p['X'], p['Y'], p['Z']) for p in perpendicular_points])
    timestamps = [p['Timestamp'] for p in perpendicular_points]

    # KD-Tree für die schnellen Nachbarschaftsabfragen der Lotpunkte
    tree = cKDTree(lot_points)

    results = []

    # Durchlaufe die XYZ-Punkte aus der TXT-Datei und finde die nächstgelegenen Lotpunkte
    for idx, position in enumerate(cmm_positions):
        progress = (idx + 1) / len(cmm_positions) * 100
        print(f"\rBerechne nearest neighbor für Lotpunkte... Fortschritt: {progress:.2f}%", end="")

        # Findet alle indices der Koordinatenmesspunkte innerhalb des Radius
        indices = tree.query_ball_point(position, r=radius)

        # 2. Wenn mehrere Punkte innerhalb des Radius liegen, den mit dem kleinsten Abstand auswählen
        if indices:
            counter = counter + 1
            distances = [np.linalg.norm(lot_points[i] - position) for i in indices]
            nearest_index = indices[np.argmin(distances)]
            nearest_timestamp = timestamps[nearest_index]  # Den zugehörigen Timestamp übernehmen

            # Ergebnis speichern
            results.append({
                'X': position[0],
                'Y': position[1],
                'Z': position[2],
                'Timestamp': nearest_timestamp  # Original-Timestamps weitergegeben
            })

        else:
            pass

    #Laufzeitanalyse
```

```
64        #end_time_NN = time.time()
65        #cpu_end_NN = time.process_time()
66
67        #print(end_time_NN-start_time_NN)
68        #print(cpu_end_NN - cpu_start_NN)
69        print(counter)
70        print("\nNearest Neighbor Suche abgeschlossen.")
71        return results
72
```

```python
from scipy.spatial import cKDTree
import numpy as np
import pandas as pd
import time

def calculate_distances_for_toolpath(nearest_neighbor_results, csv_data):
    """
    Berechnet den Abstand zwischen den Lotpunkten und den nächstgelegenen XYZ-Koordinaten basierend
    auf den Zeitstempeln der CSV-Daten.

    Parameter:
    - nearest_neighbor_results: Liste der Ergebnisse von `find_nearest_neighbors`, enthält
      Lotpunkte, zugeordnete XYZ-Koordinaten und Timestamps.
    - csv_data: Pandas DataFrame mit Zeitstempeln und den Werkzeugbahnpunkten.

    Rückgabe:
    - Eine Liste mit Abständen und zugehörigen Zeitstempeln.
    """
    distance_results = []
    counter = 0
    #transform into pandas dataFrame
    nearest_neighbor_results_df = pd.DataFrame(nearest_neighbor_results)

    # 1. Bewegungskomponenten berechnen (Differenzen zwischen aufeinanderfolgenden Punkten in x, y und z)
    csv_data['movement_direction_x'] = csv_data['x1'].diff()
    csv_data['movement_direction_y'] = csv_data['y1'].diff()
    csv_data['movement_direction_z'] = csv_data['z1'].diff()

    # Schleife durch jeden Punkt in csv_data für die Berechnung
    for idx, (index, row) in enumerate(csv_data.iterrows()):

        progress = (idx + 1) / len(csv_data) * 100
        print(f"\rBerechne toolpath distance... Fortschritt: {progress:.2f}%", end="")


        timestamp = row['time_us']

        # 2. Nearest Neighbor Resultate finden, die innerhalb eines Zeitfensters von +-2 ms liegen
        points = nearest_neighbor_results_df[
            (nearest_neighbor_results_df['Timestamp'] >= timestamp- 2) &
            (nearest_neighbor_results_df['Timestamp'] <= timestamp + 2)
        ]

        # 3. Prüfen, ob passende Punkte gefunden wurden, ansonsten überspringen
        if points.empty:
            continue

        # 4. Richtung des Werkzeugpfads (Bewegungsrichtung) berechnen
        movement_direction = np.array([
            row['movement_direction_x'],
            row['movement_direction_y'],
            row['movement_direction_z']
        ])

        # Überspringe, wenn Bewegungsrichtung NaN ist (z.B. erster Punkt)
        if np.isnan(movement_direction).any():
            counter = counter +1
            continue

        # 5. Senkrechter Richtungsvektor (Kreuzprodukt von Bewegungsrichtung mit Z-Achse)
        corr_dir = np.cross(movement_direction, np.array([0, 0, 1]))

        # 6. Berechnung des Skalarprodukts zwischen korrigierter Richtung und Lotpunkten
        direction_errors = []
        for _, point in points.iterrows():
            point_position = np.array([point['X'], point['Y'], point['Z']])
```

```
67              direction_error = np.dot(point_position, corr_dir)
68              direction_errors.append(direction_error)
69
70          # 7. Mittelwert des Direction Errors berechnen und negieren
71          mean_direction_error = -np.mean(direction_errors)
72
73          # 8. Fehlerwert mit korrigierter Richtung multiplizieren, um finalen
            Fehlervektor zu erhalten
74          final_error_vector = mean_direction_error * corr_dir
75
76          # 9. Ergebnis mit Zeitstempel speichern
77          result = {
78              'timestamp': row['time_us'],
79              'error_x': final_error_vector[0],
80              'error_y': final_error_vector[1],
81              'error_z': final_error_vector[2]
82          }
83          distance_results.append(result)
84
85      print(counter)
86
87      return distance_results
88
```

```python
import numpy as np
import pandas as pd
import time
from toolpath_distance import calculate_distances_for_toolpath
from cmm_analysis import find_nearest_neighbors

def run_runtime_tests():
    # Datengrößen
    input_sizes_csv = [10000, 100000, 1000000]  # Für csv_data
    input_sizes_txt = [1000, 10000, 100000]        # Für txt_data
    inputdata_size_nn = [10000, 100000, 1000000]  # Für nearest_neighbor_results
    inputdata_size_td = [1000, 10000, 100000]        # Für Toolpath-Distance

    results_nn = []
    results_td = []

    path_result_nn_file = "C:\\Users\\Florian Altdorf\\Desktop\\FH Aachen\\Semester 5\\Seminararbeit\\CNC_DefectAnalysis\\src\\Laufzeittests\\laufzeitanalyse_results_nn.txt"
    path_result_td_file = "C:\\Users\\Florian Altdorf\\Desktop\\FH Aachen\\Semester 5\\Seminararbeit\\CNC_DefectAnalysis\\src\\Laufzeittests\\lauzeitanalyse_result_td.txt"
    # Nearest Neighbor Tests
    with open(path_result_nn_file, "w") as nn_file:
        nn_file.write("Nearest Neighbor Test Results:\n")
        nn_file.write("nn_size\ttxt_size\truntime_sec\n")

        for nn_size in inputdata_size_nn:
            for txt_size in input_sizes_txt:
                # Testdaten generieren
                nearest_neighbor_results = [
                    {'X': np.random.uniform(-50, 50),
                     'Y': np.random.uniform(-50, 50),
                     'Z': np.random.uniform(-50, 50),
                     'Timestamp': np.random.uniform(0, nn_size)}
                    for _ in range(nn_size)
                ]

                txt_data = pd.DataFrame({
                    'x': np.random.uniform(-50, 50, txt_size),
                    'y': np.random.uniform(-50, 50, txt_size),
                    'z': np.random.uniform(-50, 50, txt_size)
                })

                # Laufzeitmessung
                start_time = time.process_time()
                find_nearest_neighbors(nearest_neighbor_results, txt_data, radius=5)
                end_time = time.process_time()

                elapsed_time = end_time - start_time
                results_nn.append({'nn_size': nn_size, 'txt_size': txt_size, 'runtime_sec': elapsed_time})
                nn_file.write(f"{nn_size}\t{txt_size}\t{elapsed_time:.2f}\n")

    # Toolpath Distance Tests
    with open(path_result_td_file, "w") as td_file:
        td_file.write("Toolpath Distance Test Results:\n")
        td_file.write("td_size\tcsv_size\truntime_sec\n")

        for td_size in inputdata_size_td:
            for csv_size in input_sizes_csv:
                # Testdaten generieren
                nn_results = [
                    {'X': np.random.uniform(-50, 50),
                     'Y': np.random.uniform(-50, 50),
                     'Z': np.random.uniform(-50, 50),
                     'Timestamp': np.random.uniform(0, td_size)}
                    for _ in range(td_size)
                ]

                csv_data = pd.DataFrame({
                    'time_us': np.linspace(0, csv_size, csv_size),
```

```
                    'x1': np.random.uniform(-50, 50, csv_size),
                    'y1': np.random.uniform(-50, 50, csv_size),
                    'z1': np.random.uniform(-50, 50, csv_size)
                })

                # Laufzeitmessung
                start_time = time.process_time()
                calculate_distances_for_toolpath(nn_results, csv_data)
                end_time = time.process_time()

                elapsed_time = end_time - start_time
                results_td.append({'td_size': td_size, 'csv_size': csv_size,
                'runtime_sec': elapsed_time})
                td_file.write(f"{td_size}\t{csv_size}\t{elapsed_time:.2f}\n")

    print("Tests abgeschlossen. Ergebnisse wurden in nearest_neighbor_results.txt und
    toolpath_distance_results.txt gespeichert.")

if __name__ == "__main__":
    run_runtime_tests()

# # Testcode zur Laufzeitanalyse
# def test_toolpath_runtime():
#     input_sizes_csv = [10000, 100000, 1000000] # 10^4 - 10^6 für csv
#     input_sizes_txt = [100, 1000, 10000]  # 10^2 - 10^4 für txt
#     inputdata_size_nn = [10000, 100000, 1000000] # 10^4 - 10^6
#     inputdata_size_td = [100, 1000, 10000] # 10^2 - 10^4
#     results = []

#     # 1. Vorabgenerierung aller Testdaten
#     test_data_tpd = {}
#     test_data_nn = {}

#     # csv
#     for size in input_sizes_csv:
#         csv_data = pd.DataFrame({
#             'time_us': np.linspace(0, size, size),
#             'x1': np.random.uniform(-50, 50, size),
#             'y1': np.random.uniform(-50, 50, size),
#             'z1': np.random.uniform(-50, 50, size)
#         })


#     # nn
#     for size in inputdata_size_td:
#         nearest_neighbor_results = [
#             {'X': np.random.uniform(-50, 50),
#              'Y': np.random.uniform(-50, 50),
#              'Z': np.random.uniform(-50, 50),
#              'Timestamp': np.random.uniform(0, size)}
#             for _ in range(size)
#         ]


#     for size in inputdata_size_nn:
#         # Generiere Lotpunkte mit Zeitstempeln
#         perpendicular_points = [
#             {'X': np.random.uniform(-50, 50),
#              'Y': np.random.uniform(-50, 50),
#              'Z': np.random.uniform(-50, 50),
#              'Timestamp': np.random.uniform(0, size)}
#             for _ in range(size)
#         ]

#     for size in input_sizes_txt:
#         # Generiere TXT-Daten mit XYZ-Koordinaten
#         txt_table = pd.DataFrame({
#             'x': np.random.uniform(-50, 50, size),
#             'y': np.random.uniform(-50, 50, size),
#             'z': np.random.uniform(-50, 50, size)
#         })

```

```
#         # Speichere die vorbereiteten Daten
#         test_data_tpd[size] = (nearest_neighbor_results, csv_data)
#         test_data_nn[size]  = (perpendicular_points, txt_table)

#     # 2. Laufzeitanalyse
#     for size in input_sizes:
#         print(f"\nStarte Test mit Eingabegröße: {size}")

#         # Lade vorbereitete Testdaten
#         nearest_neighbor_results, csv_data = test_data_tpd[size]

#         # Messung der Laufzeit für die Funktion
#         start_time_nn = time.process_time()
#         find_nearest_neighbors(perpendicular_points, txt_table, 1)
#         end_time_nn = time.process_time()

#         start_time = time.process_time()
#         calculate_distances_for_toolpath(nearest_neighbor_results, csv_data)
#         end_time = time.process_time()

#         elapsed_time_nn = end_time_nn - start_time_nn
#         elapsed_time = end_time - start_time

#         print(f"Laufzeit für Eingabegröße {size}: {elapsed_time_nn:.2f} Sekunden")
#         print(f"Laufzeit für Eingabegröße {size}: {elapsed_time:.2f} Sekunden")

#         # Ergebnisse speichern
#         results.append({'input_size': size, 'runtime_sec_nn': elapsed_time_nn,
'runtime_sec': elapsed_time})

#     # Ergebnisse als Tabelle anzeigen
#     print("\nLaufzeitanalyse:")
#     print(pd.DataFrame(results))

# if __name__ == "__main__":
#     test_toolpath_runtime()
```