

Dokumentation der Abschlussprüfung im Ausbildungsberuf
„Mathematisch-Technische*r Softwareentwickler*in“,
Prüfungsbereich: „Entwicklung eines Softwaresystems“

Entwicklung einer Software zur Simulation einer „Spidercam“

Patrick Gustav Blaneck (20000)

IT Center RWTH Aachen University

Programmiersprache:  Python



2. Dezember 2022

Eidesstattliche Erklärung

Ich erkläre verbindlich, dass das vorliegende Prüfprodukt von mir selbstständig erstellt wurde. Die als Arbeitshilfe genutzten Unterlagen sind in der Arbeit vollständig aufgeführt. Ich versichere, dass der vorgelegte Ausdruck mit dem Inhalt der von mir erstellten digitalen Version identisch ist. Weder ganz noch in Teilen wurde die Arbeit bereits als Prüfungsleitung vorgelegt. Mir ist bewusst, dass jedes Zuwiederhandeln als Täuschungsversuch zu gelten hat, der die Anerkennung des Prüfprodukts als Prüfungsleistung ausschließt.

Name: Patrick Gustav Blaneck

Aachen, den 2. Dezember 2022



Unterschrift der/des Auszubildenden

Inhaltsverzeichnis

1	Aufgabenanalyse	5
1.1	Eingabe	6
1.1.1	Format	6
1.1.2	Fehlerquellen und -behebung	6
1.2	Ausgabe	7
1.2.1	Format	7
1.3	Mathematische Modellierung	7
1.3.1	Berechnung der Seillängen	7
1.3.2	Bewegung der Spidercam	8
1.3.3	Abbrechen der Bewegung	10
2	Verfahrensbeschreibung	12
2.1	Datenstrukturen	12
2.1.1	Phase	12
2.1.2	Movement	13
2.1.3	Spidercam	14
2.1.4	Controller	15
2.1.5	Input und Output	15
2.1.6	Plotter	16
2.2	Algorithmen	17
2.2.1	Verarbeitung der Eingabedatei	17
2.2.2	Bewegung der Spidercam	18
2.2.3	Erstellung der Ausgabedatei	19
3	Programmbeschreibung	21
3.1	Allgemeiner Programmablauf	21
3.2	Beispielablauf	23
4	Testing	27
4.1	Testfälle	27
4.2	Beobachtungen	27
5	Abweichungen vom ursprünglichen Konzept	32
6	Zusammenfassung und Ausblick	33
6.1	Zusammenfassung	33
6.2	Ausblick	33
	Anhang	34

A	Benutzeranleitung	34
A.1	Installation	34
A.1.1	Installation mit conda	34
A.1.2	Installation mit pip	34
A.2	Benutzung	35
A.2.1	Beispiele	35
A.2.2	Fehlermeldungen	36
A.2.3	Skripte	36
B	Entwicklerdokumentation	37
C	Hilfsmittel	38
D	Testdateien	39
E	Quellcode	44

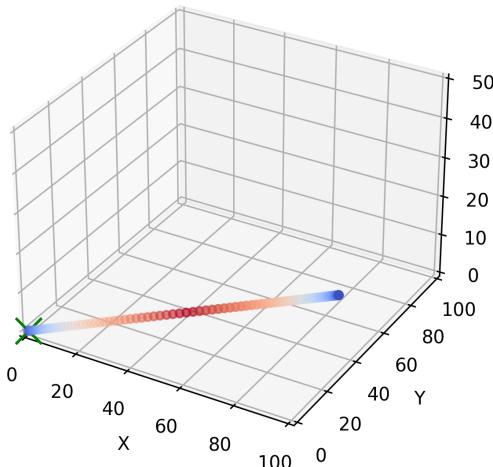
1 Aufgabenanalyse

Im Rahmen dieses Softwareprojekts soll ein Programm erstellt werden, welches aus gegebenen Eckdaten und einer Liste an Instruktionen die Bewegung einer sogenannten *Spidercam* simuliert. Definition aus der Aufgabenstellung:

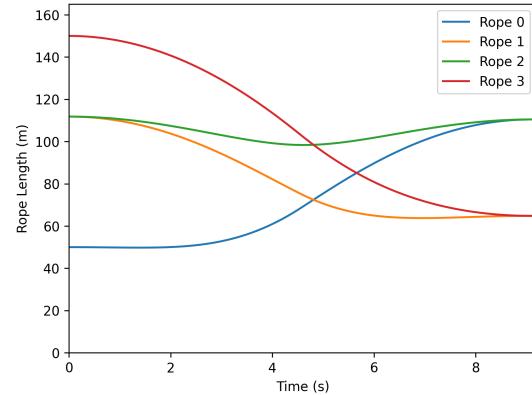
Spidercams sind Kameras, die von vier Drahtseilen in der Luft gehalten werden. Die Drahtseile sind an vier festen Positionen verankert, können aber über dort angebrachte Seilwinden in der Länge variiert werden.

Die besondere Schwierigkeit besteht unter anderem daraus, dass:

- eine lineare Bewegung der Spidercam nicht unbedingt in einer linearen Veränderung der Seillängen resultiert (siehe Abbildung 1),
- Instruktionen „live“ verarbeitet werden - also nicht nur die Bewegung der Spidercam, sondern auch die Veränderung der Seillängen in „Echtzeit“ simuliert werden muss¹,
- die Spidercam sowohl beschleunigen als auch abbremsen muss und
- Instruktionen abgebrochen und in eine Warteschlange gestellt werden können.



(a) Lineare Bewegung der Spidercam



(b) Veränderung der Seillängen

Abbildung 1: Beispiel für die Veränderung der Seillängen bei einer einfachen linearen Bewegung der Spidercam. Es ist zu erkennen, dass sich die Seillängen nicht linear verändern. Die Farbe der Bewegungspunkte gibt dabei an, wie schnell sich die Spidercam in diesem Moment bewegt. Das grüne X markiert die Startposition der Spidercam.

Die einzelnen Punkte werden in den folgenden Abschnitten genauer erläutert.

¹Die Aufgabenstellung spricht konkret davon, dass die Instruktionen nicht vorausschauend verarbeitet werden dürfen.

1.1 Eingabe

1.1.1 Format

Die Parameter für die Simulation einer Spidercam werden in Form einer Textdatei eingelesen. Ein Beispiel für eine Eingabedatei ist in Abbildung 2 zu sehen.

```
1 # Beispiel
2 dim 70 100 30
3 start 10 80 10
4 vmax 6
5 amax 2
6 freq 2
7 0 50 40 30 # Instruktion 1
8 20 10 80 10 # Instruktion 2
9 22 50 40 30      # Instruktion 3
10 23 35 50 30      # Instruktion 4
11 27.5 10 80 20 # Instruktion 5
```

Abbildung 2: Beispiel für eine Eingabedatei

Kommentare werden mit einem # eingeleitet und dauern bis zum Ende der Zeile an. Die Parameter werden in der Reihenfolge der folgenden Liste eingelesen²:

- Dimension des Spielfelds (`dim x y z`),
- Startkoordinaten der Spidercam (`start x y z`),
- Maximalgeschwindigkeit der Spidercam (`vmax v`),
- (Maximal-)Beschleunigung der Spidercam (`amax a`),
- Diskretisierung der Bewegung (`freq f`),
- Beliebige Anzahl an Instruktionen mit Zeitpunkt und Zielkoodinaten (`t x y z`).

1.1.2 Fehlerquellen und -behebung

Die offensichtlichste Fehlerquellen sind:

- Dimension besitzt nicht positive Werte
- Startposition außerhalb des Spielfelds
- Maximalgeschwindigkeit oder -beschleunigung sind nicht positiv
- Diskretisierung ist nicht positiv
- Instruktionen außerhalb des Spielfelds³ oder nicht vorhanden

Fehlerhafte Eingaben werden vor Initialisierung der Simulation abgefangen und mit einer Fehlermeldung beendet.

²Von einer *syntaktisch* korrekten Eingabedatei kann ausgegangen werden.

³Von chronologisch korrekt sortierten Instruktionen kann ausgegangen werden.

1.2 Ausgabe

1.2.1 Format

Für die Simulation sollen zwei CSV-Ausgabedateien erzeugt werden, die zu diskreten Zeitpunkten von $t = 0$ bis zum Zeitpunkt der Beendigung der letzten Instruktion t_c die Längen der Drahtseile und die Positionen der Spidercam in der in der Eingabedatei angegebenen Frequenz f [$\text{Hz} = \text{s}^{-1}$] speichern.

Ausgabedatei 1 (siehe Abbildung 3) enthält pro Zeile die Längen eines Seils zu jedem diskreten Zeitpunkt $t_i = i \cdot f^{-1}$, wobei $i \in [0, t_c \cdot f]$. Für jeden Verankerungspunkt wird also eine Zeile angelegt.

Ausgabedatei 2 (siehe Abbildung 4) enthält zeilenweise:

- Dimension des Spielfelds,
- diskrete Zeitpunkte,
- x -Koordinate der Spidercam zu jedem Zeitpunkt t_i ,
- y -Koordinate der Spidercam zu jedem Zeitpunkt t_i ,
- z -Koordinate der Spidercam zu jedem Zeitpunkt t_i .

```
1 83.0662,82.90594,82.427746,[...],64.0056,64.03124
2 101.9803,101.7352,101.0,[...],44.94510,44.7213
3 30.0,30.111,30.4576,[...],77.86780,78.1024
4 66.332,66.20721,65.8356,[...],63.14055,63.2455
```

Abbildung 3: Beispiel für eine Ausgabedatei 1

```
1 70,100,30
2 0.0,0.5,1.0,[...],12.5,13.0,13.5
3 10.0,10.16666666666666,10.66666666666666,[...],49.83333333333336,50.0
4 80.0,79.833333333333,79.333333333333,[...],40.1666666666664,40.0
5 10.0,10.083333333334,10.3333333333334,[...],29.9166666666668,30.0
```

Abbildung 4: Beispiel für eine Ausgabedatei 2

1.3 Mathematische Modellierung

1.3.1 Berechnung der Seillängen

Die Verankerungspunkte der Drahtseile sind eindeutig durch die Dimension des Spielfeldes gegeben. Es gilt:

$$R_1 = \begin{pmatrix} 0 \\ 0 \\ d_z \end{pmatrix}, \quad R_2 = \begin{pmatrix} d_x \\ 0 \\ d_z \end{pmatrix}, \quad R_3 = \begin{pmatrix} 0 \\ d_y \\ d_z \end{pmatrix}, \quad R_4 = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$$

Die Längen der Drahtseile l_i zu einem beliebigen Punkt im Raum $A = (x, y, z)$ sind dann trivialerweise gegeben durch:

$$l_i = d(R_i, A) = \|(R_i - A)\|_2 = \sqrt{(x - R_{i,x})^2 + (y - R_{i,y})^2 + (z - R_{i,z})^2}$$

1.3.2 Bewegung der Spidercam

Die Konstanten der Eingabedatei a_{\max} und v_{\max} werden hier als gegeben betrachtet. Mithilfe dieser Konstanten können dann der Zeitraum t_{\max} und die Distanz d_{\max} berechnet werden, die die Spidercam in der Simulation benötigt, um die maximale Geschwindigkeit v_{\max} zu erreichen. Es gilt:

$$t_{\max} = \frac{v_{\max}}{a_{\max}} \quad \text{und} \quad d_{\max} = \frac{v_{\max}^2}{2 \cdot a_{\max}}$$

Die maximale Geschwindigkeit v_{\max} wird also nach t_{\max} bzw. d_{\max} erreicht.

Eine Bewegung der Spidercam ist dann eindeutig durch den Startpunkt A und den Zielort B gegeben. Es können zwei Fälle unterschieden werden.

Eine *dreiphasige Bewegung* tritt auf, wenn die Spidercam die maximale Geschwindigkeit v_{\max} auf dem Weg von A nach B erreichen kann.

Es gilt ($d(A, B) > 2 \cdot d_{\max}$)

$$\underbrace{(A)}_{\text{Start}} \rightarrow \underbrace{\left(A + \frac{d_{\max}}{d(A, B)} \cdot (B - A) \right)}_{\text{Zwischenpunkt}} \rightarrow \underbrace{\left(B - \frac{d_{\max}}{d(A, B)} \cdot (B - A) \right)}_{\text{Zwischenpunkt}} \rightarrow \underbrace{(B)}_{\text{Ziel}}$$

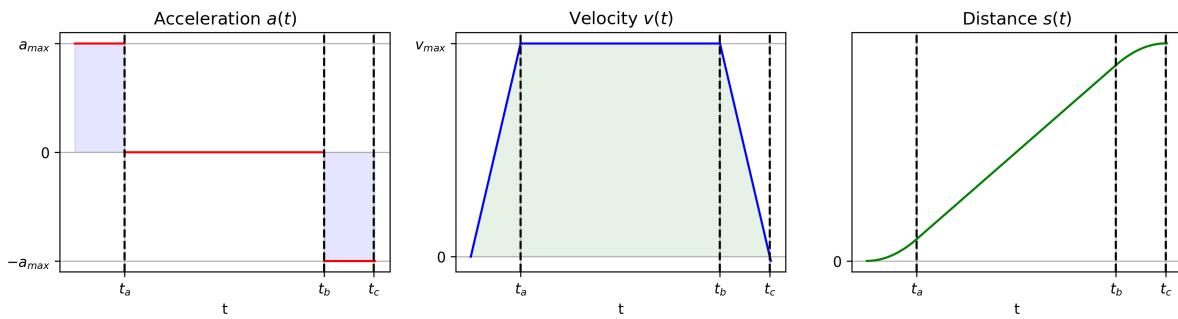


Abbildung 5: Bewegung der Spidercam in drei Phasen. t_a entspricht dem Zeitraum, in dem die Spidercam maximal beschleunigt wird. t_b ist der Zeitpunkt, an dem abgebremst wird und t_c ist der Zeitpunkt, an dem die Spidercam ihren Zielort erreicht hat.

Es gilt also, dass nach t_{\max} bzw. d_{\max} der Zwischenpunkt t_a erreicht wird, an dem die maximale Geschwindigkeit v_{\max} erreicht ist. Danach bewegt sich die Spidercam mit

konstanter Geschwindigkeit v_{\max} bis zum Zeitpunkt t_b , ab dem abgebremst werden muss, um am Zielort die Geschwindigkeit 0 zu erreichen. Analog zum Beschleunigen benötigt die Spidercam zum Abbremsen auf 0 die Zeit t_{\max} bzw. die Distanz d_{\max} . Für die Phase der konstanten Geschwindigkeit gilt also:

$$d_{\text{CON}} = d(A, B) - 2 \cdot d_{\max}$$

Da die Geschwindigkeit konstant ist, gilt für die Zeit t_{CON} und damit t_b :

$$t_{\text{CON}} = \frac{d_{\text{CON}}}{v_{\max}} = \frac{d(A, B) - 2 \cdot d_{\max}}{v_{\max}} \implies t_b = t_{\max} + t_{\text{CON}}$$

Für diese Phase gilt damit (siehe auch Abbildung 5):

$$a(t) = \begin{cases} a_{\max} & \text{für } 0 \leq t \leq t_a \\ 0 & \text{für } t_a < t \leq t_b \\ -a_{\max} & \text{für } t_b < t \leq t_c \end{cases}$$

$$v(t) = \int_0^t a(u) \, du = \begin{cases} a_{\max} \cdot t & \text{für } 0 \leq t \leq t_a \\ v_{\max} & \text{für } t_a < t \leq t_b \\ v_{\max} - a_{\max} \cdot (t - t_b) & \text{für } t_b < t \leq t_c \end{cases}$$

$$s(t) = \int_0^t v(u) \, du = \begin{cases} \frac{a_{\max}}{2} \cdot t^2 & \text{für } 0 \leq t \leq t_a \\ s(t_a) + v_{\max} \cdot (t - t_a) & \text{für } t_a < t \leq t_b \\ s(t_a) + s(t_b) + v_{\max} \cdot (t - t_b) - \frac{a_{\max}}{2} \cdot (t - t_b)^2 & \text{für } t_b < t \leq t_c \end{cases}$$

Eine *zweiphasige Bewegung* (siehe Abbildung 6) tritt auf, wenn die Spidercam die maximale Geschwindigkeit v_{\max} *nicht* auf dem Weg von A nach B erreichen kann.

Es gilt ($d(A, B) \leq 2 \cdot d_{\max}$):

$$\underbrace{(A)}_{\text{Start}} \rightarrow \underbrace{\left(A + \frac{d(A, B)}{2} \cdot (B - A) \right)}_{\text{Zwischenpunkt}} \rightarrow \underbrace{(B)}_{\text{Ziel}}$$

Es ist also direkt ersichtlich, dass eine Phase konstanter Geschwindigkeit entfällt. Die höchste Geschwindigkeit in dieser Bewegung v_x wird also aufgrund der gleich langen Beschleunigungs- und Abbremsphasen genau zwischen A und B erreicht. Die Zeit t_x ist der Zeitpunkt, an dem die maximale Geschwindigkeit v_x erreicht wird.

Es gilt also:

$$d_x = \frac{d(A, B)}{2} \implies t_x = \frac{d(A, B)}{2 \cdot v_{\max}} \implies v_x = a_{\max} \cdot t_x \left(= \frac{d_x}{t_x} \right)$$

Die Berechnungen in dieser Form werden später auch in der Implementierung verwendet.

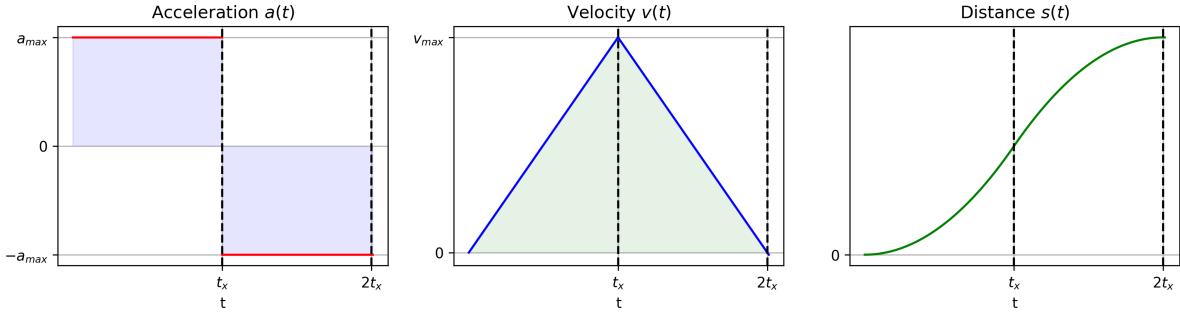


Abbildung 6: Bewegung der Spidercam in zwei Phasen. t_x ist der Zeitpunkt, an dem abgebremst wird und $2t_x$ ist der Zeitpunkt, an dem die Spidercam ihren Zielort erreicht hat.

1.3.3 Abbrechen der Bewegung

Instruktionen zur Bewegung der Spidercam können zu jedem Zeitpunkt t stattfinden. Befindet sich die Spidercam jedoch bereits in Bewegung während eine neue Instruktion ankommt, so wird die aktuelle Bewegung abgebrochen, indem die Spidercam sofort anfängt zu bremsen⁴. Die neue Bewegung beginnt dann erst, wenn die Spidercam ihre Geschwindigkeit auf 0 gebracht hat. Kommt eine neue Instruktion an, obwohl bereits eine Instruktion auf Ausführung wartet, so wird die alte Instruktion überschrieben, da diese als veraltet angenommen wird.

Für die weitere Implementierung ist es hier wichtig zu klären, wie sich die Bremsdistanz d_{DEC} , die Bremszeit t_{DEC} und das neue Ziel B' bestimmen lassen.

Die Bremsdistanz d_{DEC} ist die Distanz, die die Spidercam zurücklegt, wenn sie von ihrer aktuellen Geschwindigkeit v auf 0 abgebremst wird. Die Bremsdistanz ist also gleich der Fläche unter der Geschwindigkeitskurve $v(t)$ der aktuellen Bremsphase (siehe Abbildung 7)

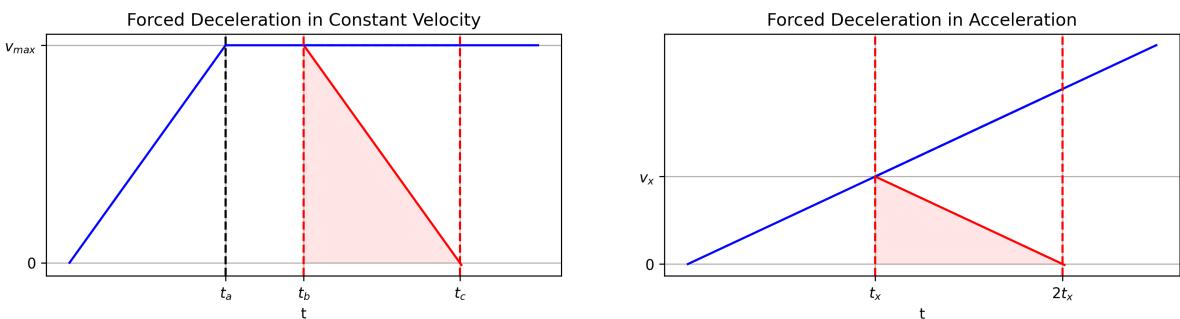


Abbildung 7: Berechnung der Bremsdistanz d_{DEC} . Die rote Fläche unter der Geschwindigkeitskurve ist die Bremsdistanz.

⁴Wenn bereits gebremst wird, muss nicht neu abgebremst werden.

Da die Beschleunigung konstant ist, gilt:

$$v(t) = \begin{cases} v_{\max} & \text{für Abbruch in Phase konstanter Geschwindigkeit} \\ a_{\max} \cdot t & \text{für Abbruch in Beschleunigungsphase} \end{cases}$$

Damit lassen sich dann die Bremszeit t_{DEC} und die Bremsdistanz d_{DEC} berechnen:

$$t_{\text{DEC}} = \frac{v(t)}{a_{\max}} \implies d_{\text{DEC}} = \frac{v(t)^2}{2 \cdot a_{\max}}$$

Die neue Position B' ist der Punkt, an dem die Spidercam anhalten würde, wenn sie von ihrer aktuellen Position A für die Bremsdistanz d_{DEC} in Richtung B fahren würde. Dazu wird der Vektor $(B - A)$ normiert und mit der Bremsdistanz d_{DEC} multipliziert:

$$B' = A + \frac{d_{\text{DEC}}}{d(A, B)} \cdot (B - A)$$

2 Verfahrensbeschreibung

Für den zu simulierenden Sachverhalt lohnt es sich, jede Bewegung der Kamera in Phasen zu unterteilen, um unübersichtliche Strukturen zu vermeiden. So bietet es sich an, eine *Phase* (*Phase*) als entweder eine *Beschleunigungs-* (*ACCELERATION*), *Konstantgeschwindigkeits-* (*CONSTANT_VELOCITY*) oder eine *Bremsphase* (*DECELERATION*) zu definieren.

Eine *Bewegung* (*Movement*) besteht dann aus entweder einer Beschleunigungs-, einer Konstantgeschwindigkeits- und einer Bremsphase oder nur aus einer Beschleunigungs- und einer Bremsphase.

Eine Instanz der Spidercam (*Spidercam*) enthält dann zusätzlich zu bekannten Konstanten v_{\max} (*max_velocity*) und a_{\max} (*acceleration*) eine Liste von Bewegungen (*movements*) und eine einelementige Warteschlange (*queue*). Zusätzlich kennt die Spidercam ihre initiale Position (*start*).

2.1 Datenstrukturen

2.1.1 Phase

Eine Phase (*Phase*) wird als eine Klasse modelliert, die eindeutig durch folgende Attribute definiert ist:

- *movement*: Eine Referenz auf die Bewegung, zu der die Phase gehört.
- *mode*: Ein Wert aus dem Enum *Mode*, der angibt, ob es sich um eine Beschleunigungs- (*ACCELERATION*), Konstantgeschwindigkeits- (*CONSTANT_VELOCITY*) oder Bremsphase (*DECELERATION*) handelt.
- *start*: Startkoordinaten der Phase.
- *starting_velocity*: Startgeschwindigkeit der Phase.
- *dest*: Endkoordinaten der Phase.

Nachdem eine Phase mit den genannten Attributen initialisiert wurde, kann mithilfe der Funktion *update()* die Phase aktualisiert werden. Dabei wird abhängig vom Modus der Phase berechnet, welche Distanz (*distance*) am Ende der Phase zurückgelegt wurde und wie lange die Phase dauert (*duration*).

Phase
Mode : Mode destination : tuple distance : ndarray duration logger : NoneType, RootLogger, Logger mode : Mode movement : Movement start : tuple starting_velocity : float
calc_distance(): float calc_duration(): float get_position(offset: float): tuple update(): None

Abbildung 8: Klassendiagramm der Klasse Phase

2.1.2 Movement

Eine Bewegung (Movement) wird als eine Klasse modelliert, die eindeutig durch folgende Attribute definiert ist:

- `spidercam`: Eine Referenz auf die Spidercam, zu der die Bewegung gehört.
- `start_time`: Startzeitpunkt der Bewegung.
- `start`: Startkoordinaten der Bewegung.
- `destination`: Endkoordinaten der Bewegung.

Nachdem eine Bewegung mit den genannten Attributen initialisiert wurde, können mithilfe der Funktion `calculate_phases()` die Phasen der Bewegung berechnet werden. Diese werden dann in der Liste `phases` gespeichert.

Die Start- und Endkoordinaten der Bewegung können mithilfe der Funktionen `start()` und `dest()` abgerufen werden. Dabei werden die Startkoordinaten der ersten bzw. die Endkoordinaten der letzten Phase zurückgegeben.

Movement
logger : RootLogger, NoneType, Logger phases : list spidercam : Spidercam start_time : float calculate_phases(start: tuple, destination: tuple): None destination(): tuple distance(): float duration(): float end_time(): float get_phase(time: float): spidercam_simulator.Phase get_position(time: float): tuple start(): tuple start_deceleration(time: float): None

Abbildung 9: Klassendiagramm der Klasse Movement

2.1.3 Spidercam

Eine Spidercam (Spidercam) wird als eine Klasse modelliert, die eindeutig durch folgende Attribute definiert ist:

- **controller:** Eine Referenz auf den Controller, zu dem die Spidercam gehört.
- **max_velocity:** Maximale Geschwindigkeit der Spidercam.
- **acceleration:** Beschleunigung der Spidercam.
- **start:** Startkoordinaten der Spidercam.

Nachdem eine Spidercam mit den genannten Attributen initialisiert wurde, können mithilfe der Funktion `calc_constants()` weitere Konstanten der Bewegung berechnet werden. Diese sind konkret wie in 1.3 beschrieben t_{\max} (`time_vmax`) und d_{\max} (`distance_vmax`).

Spidercam
acceleration : float controller : Controller dist_vmax : float logger : NoneType, RootLogger, Logger max_velocity : float movements : list movements : list queue : NoneType queue : NoneType, Movement start : tuple time_vmax : float calc_constants(): None get_position(time: float): tuple move(instruction: spidercam_simulator.Instruction, time: float): None

Abbildung 10: Klassendiagramm der Klasse Spidercam

2.1.4 Controller

Um die Bewegungen der Spidercam zu steuern, wurde ein Controller (`Controller`) implementiert. Dieser wird als eine Klasse modelliert, die eindeutig durch folgende Attribute definiert ist:

- `dim`: Dimension des Raumes, in dem sich die Spidercam bewegt.
- `start`: Startkoordinaten der Spidercam.
- `max_velocity`: Maximale Geschwindigkeit der Spidercam.
- `acceleration`: Beschleunigung der Spidercam.
- `freq`: Diskretisierungsfrequenz der Bewegung.
- `instructions`: Liste der Bewegungsanweisungen.

Nachdem der Controller mit den genannten Attributen initialisiert wurde, berechnet dieser mithilfe der Funktion `calc_anchors()` die fixen Ankerpunkte der Drahtseile. Diese werden in der Liste `anchors` gespeichert.

Anschließend kann der Controller mit der Funktion `run()` gestartet werden. Dabei wird in diskreten Zeitintervallen der Frequenz `freq` die Spidercam entweder mit einer neuen Instruktion beauftragt, wenn eine noch nicht bearbeitete Instruktion zum aktuellen Zeitpunkt existiert, oder die Spidercam zum aktuellen Zeitpunkt aktualisiert. In jedem Zeitpunkt wird die aktuelle Position der Spidercam bestimmt und damit die aktuellen Längen der Drahtseile berechnet. Diese werden dann in den Listen `cam_positions` und `rope_lengths` gespeichert.

Controller
anchors : list cam_positions : list dim : tuple freq : float instructions : Optional[list[spidercam_simulator.Instruction]] logger : Logger, RootLogger, NoneType rope_lengths : list spidercam : Spidercam
from_dict(data: dict): 'Controller' get_rope_lengths(position: tuple): list[tuple] run(): tuple[list[tuple], list[tuple]] store_anchors(): None

Abbildung 11: Klassendiagramm der Klasse Controller

2.1.5 Input und Output

Um insbesondere den Import von Eingabedateien und den Export von Ausgabedateien zu vereinfachen, wurden einige Hilfsklassen und -funktionen implementiert.

FileHandler Die Klasse `FileHandler` ist eine Hilfsklasse, die zu einer Eingabedatei die entsprechenden Ausgabedateien erzeugt. Diese wird instanziert, indem ihr die Pfade zur *Eingabedatei* und zum *Ausgabeordner* übergeben werden. Basierend auf dem Namen der Eingabedatei werden dann die Pfade zu den Ausgabedateien bzw. die Ausgabedateien selbst definiert.

Instruction Die Klasse `Instruction` ist eine Hilfsklasse, die eine Bewegungsanweisung repräsentiert. Diese wird instanziert, indem ihr eine syntaktisch korrekte Zeile aus einer Eingabedatei übergeben wird, in der die Bewegungsanweisung definiert ist. Die Instanz enthält dann den definierten Startzeitpunkt der Bewegungsanweisung (`start_time`) und das Ziel der Bewegungsanweisung (`destination`).

Parser Die Klasse `Parser` ist eine Hilfsklasse, die eine Eingabedatei einliest und die darin definierten Parameter und Bewegungsanweisungen extrahiert. Die statische Methode `parse_input()` dieser Klasse erwartet als Parameter einen String, der den Inhalt einer Eingabedatei enthält. Die Methode erzeugt dann ein Dictionary mit den extrahierten Parametern und Bewegungsanweisungen und gibt dieses zurück.

Die ebenfalls statische Methode `parse_output()` dieser Klasse erwartet als Parameter die folgenden zu exportierenden Daten:

- Für Ausgabedatei 1:
 - `rope_lengths`: Liste der Längen der Drahtseile in jedem Zeitpunkt.
- Für Ausgabedatei 2:
 - `dim`: Dimension des Raumes, in dem sich die Spidercam bewegt.
 - `freq`: Diskretisierungsfrequenz der Bewegung.
 - `cam_positions`: Liste der Positionen der Spidercam in jedem Zeitpunkt.

Entsprechend des in 1.2 definierten Ausgabeformats werden die Daten in zwei Strings umgewandelt und zurückgegeben.

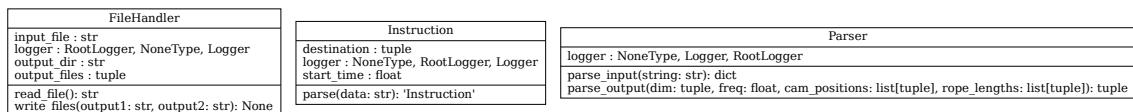


Abbildung 12: Klassendiagramme der Hilfsklassen `FileHandler`, `Instruction` und `Parser`

2.1.6 Plotter

Auf diese Klasse soll hier nur kurz eingegangen werden. Sie ist eine Hilfsklasse, die die Ausgabedateien 1 und 2 in Form von Graphen visualisiert. Diese wird instanziert, indem ihr analog zur Funktion `parse_output()` die entsprechenden Daten übergeben werden.

Plotter
cam_positions : ndarray dim : Optional[tuple] freq : float logger : Logger, RootLogger, NoneType name : Optional[str] output_dir : Optional[str] rope_lengths : ndarray
plot(): None plot_cam_positions(): None plot_rope_lengths(): None

Abbildung 13: Klassendiagramm der Klasse Plotter

2.2 Algorithmen

Die genutzten Algorithmen unterteilen sich in drei Kategorien:

- Verarbeitung der Eingabedatei
- Bewegung der Spidercam
- Erstellung der Ausgabedatei

2.2.1 Verarbeitung der Eingabedatei

Die Eingabe wird wie bereits definiert in der Funktion `parse_input()` der Klasse `Parser` verarbeitet.

Der Eingabestring wird Zeile für Zeile durchlaufen und die Zeilen werden anhand der ersten Zeichen der Zeilen der jeweiligen Kategorie (`dim`, `start`, ...) zugeordnet.

Anschließend werden die einzelnen Parameter auf ihre semantische Korrektheit überprüft (siehe dazu die definierten Fehlerquellen in [1.1.2](#)). Sind alle Parameter korrekt, wird ein Dictionary mit den extrahierten Parametern und Bewegungsanweisungen zurückgegeben.

Für eine Visualisierung siehe Abbildung [14](#).

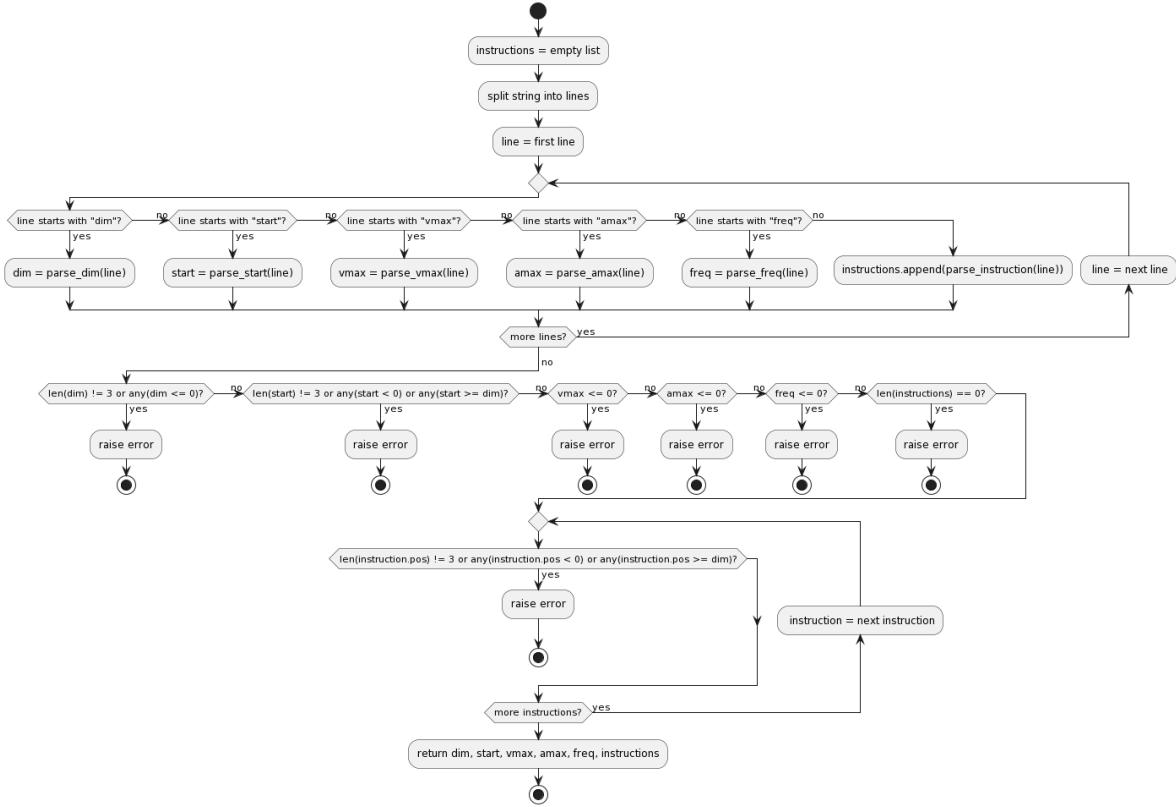


Abbildung 14: Aktivitätsdiagramm der Funktion `parse_input()` der Klasse `Parser`

2.2.2 Bewegung der Spidercam

Die Bewegung der Spidercam wird in der Funktion `move()` der Klasse `Spidercam` durchgeführt. Dabei kann zwischen zwei Fällen unterschieden werden:

1. Die Spidercam bekommt eine neue Bewegungsanweisung `instruction`, siehe [2.2.2](#).
2. Die Spidercam erhält einen Zeitpunkt `time` und soll für diesen aktualisiert werden, siehe [2.2.2](#).

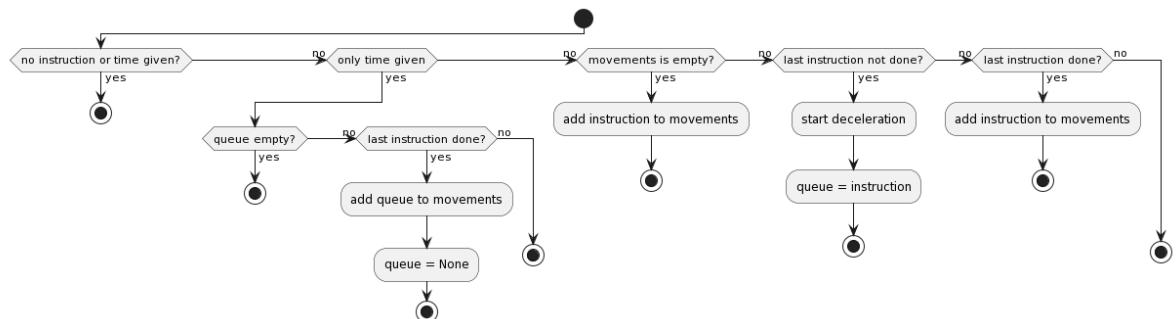


Abbildung 15: Aktivitätsdiagramm der Funktion `move()` der Klasse `Spidercam`

Neue Bewegungsanweisung Wird der Spidercam eine neue Bewegungsanweisung `instruction` übergeben, so wird die übergebene Instruktion als neues Movement in der Liste `movements` der Spidercam gespeichert, wenn entweder noch keine Bewegung stattgefunden hat (`movements` ist leer) oder die letzte Bewegung bereits abgeschlossen ist (`movements[-1].end_time()` ist kleiner als `time`). Der Startpunkt der Bewegung ist dabei also entweder das Ziel der letzten Bewegung oder die Startposition der Spidercam.

Andernfalls wird zum Zeitpunkt `time` der Bremsvorgang eingeleitet (siehe [2.2.2](#)) und die aktuelle Warteschlange `queue` überschrieben.

Aktualisierung für Zeitpunkt Wird der Spidercam ein Zeitpunkt `time` übergeben, so wird überprüft, ob die aktuelle Bewegung abgeschlossen ist. Ist dies der Fall, wird, sofern vorhanden, die nächste Bewegung aus der Warteschlange `queue` in die Liste `movements` übernommen und die Warteschlange wird geleert.

Ist die aktuelle Bewegung noch nicht abgeschlossen, so muss nichts weiter getan werden.

Bremsvorgang Der Bremsvorgang wird mittels der Funktion `start_deceleration()` der Klasse Movement eingeleitet, wenn die Spidercam eine neue Bewegungsanweisung erhält, die mit der aktuellen Bewegung kollidiert.

Dabei wird zuerst geprüft, in welche Bewegungsphase der aktuelle Zeitpunkt `time` fällt und wie viel Zeit in der Phase vergeht (`offset`). In einer Bremsphase muss nichts weiter getan werden, da die Spidercam bereits abgebremst wird.

Andernfalls wird zunächst bestimmt, welcher Weg zum Zeitpunkt seit Beginn der Phase zurückgelegt und welcher Punkt erreicht wurde. Dieser Punkt ist dann der Zielpunkt für die unterbrochene Phase und der Startpunkt der neuen Bremsphase.

Anschließend wird die neue Bremsphase so abgeändert, dass die Startgeschwindigkeit (`starting_velocity`) der Bremsphase gleich der aktuellen Geschwindigkeit der Spidercam ist (siehe [1.3](#)) und der Modus der Bremsphase auf `DECELERATION` gesetzt wird. Es ist trivial, dass nach $v(t) \cdot a_{\max}^{-1}$ Sekunden die Geschwindigkeit der Spidercam 0 ist. Der Zielpunkt der Bremsphase ist also eben dieser Punkt, der nach $v(t) \cdot a_{\max}^{-1}$ Sekunden während der Bremsphase erreicht wird.

2.2.3 Erstellung der Ausgabedatei

Nach [1.2](#) sollen pro Eingabe zwei Ausgabedateien erstellt werden. Diese Aufgabe wird in der Funktion `parse_output()` der Klasse Parser durchgeführt. Diese erhält als Parameter die Dimension des Raumes, die Diskretisierungsfrequenz und die Listen der Kamerapositionen und der Seillängen zu jedem Zeitpunkt.

Die erste Datei soll die Längen der Drahtseile zu jedem diskreten Zeitpunkt enthalten. Dazu wird zunächst die Liste `rope_lengths` transponiert, da pro Zeile die Längen eines einzelnen Seils auszugeben sind und die Liste `rope_lengths` pro Zeile die Längen aller Seile zu einem Zeitpunkt enthält. Transponieren erzeugt also aus einer $n \times 4$ -Matrix eine $4 \times n$ -Matrix, durch deren Zeilen dann iteriert werden kann. Dies erfolgt wie in Listing 1 beschrieben.

Die zweite Datei soll die Dimension, die diskreten Zeitpunkte und die Kamerapositionen zu jedem Zeitpunkt enthalten. Analog zu den Seillängen wird auch hier zunächst die Liste `camera_positions` transponiert, um pro Zeile eine Koordinate der Kameraposition zu erhalten. Der Rest der Ausgabe erfolgt analog zu Listing 1.

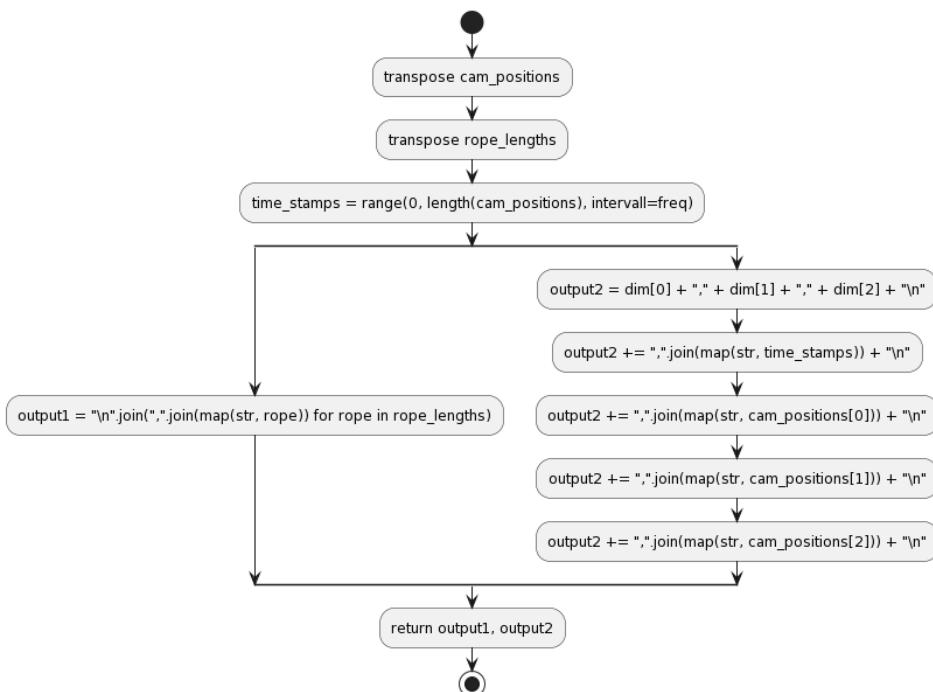


Abbildung 16: Aktivitätsdiagramm der Funktion `parse_output()`

```
1 output1 = "\n".join(", ".join(map(str, rope)) for rope in rope_lengths)
```

Listing 1: Erstellung der Ausgabedatei 1 (Längen der Drahtseile)

```
1 # adding the dimensions to the output
2 output2 = f'{dim[0]},{dim[1]},{dim[2]}\n'
3 # adding the time stamps to the output
4 output2 += ", ".join(map(str, time_stamps)) + "\n"
5 # adding the positions of the camera to the output
6 output2 += ", ".join(map(str, cam_positions[0])) + "\n"
7 output2 += ", ".join(map(str, cam_positions[1])) + "\n"
8 output2 += ", ".join(map(str, cam_positions[2])) + "\n"
```

Listing 2: Erstellung der Ausgabedatei 2 (Kamerapositionen)

3 Programmbeschreibung

3.1 Allgemeiner Programmablauf

Wie bereits in der Verfahrensbeschreibung beschrieben, werden folgende Datenstrukturen (insbesondere Klassen) benötigt:

- Eine Klasse `Controller` zum Steuern der Spidercam, siehe [2.1.4](#).
- Eine Klasse `Spidercam` zum Modellieren der Spidercam, siehe [2.1.3](#).
- Eine Klasse `Movement` zum Modellieren der Bewegungen der Spidercam, siehe [2.1.2](#).
- Eine Klasse `Phase` zum Modellieren der Phasen der Bewegungen der Spidercam, siehe [2.1.1](#).

Die Klassen stehen wie in Abbildung [17](#) dargestellt in Beziehung zueinander.

Zum Start des Programms wird die Datei `__main__.py` ausgeführt. Diese Datei enthält die Hauptfunktion des Programms. Sie liest eine Konfigurationsdatei `config.ini` und Argumente aus der Kommandozeile ein und überprüft diese auf Korrektheit.

Anschließend wird für alle definierten Eingabedateien ein `Controller`-Objekt instanziert. Dieses Objekt aktualisiert bzw. instruiert die Spidercam dann zu den entsprechenden diskreten Zeitpunkten. Zu jedem Zeitpunkt wird die Position der Spidercam bestimmt und die Längen zu den Verankerungspunkten der Drahtseile berechnet. Die Simulation stoppt dann, wenn der `Controller` keine weiteren Instruktionen mehr hat und die Spidercam alle Bewegungen ausgeführt hat.

Anschließend werden die Ergebnisse entsprechend in die Ausgabedateien geschrieben.

Die Aufrufhierarchie ist beispielhaft in Abbildung [18](#) zu sehen.

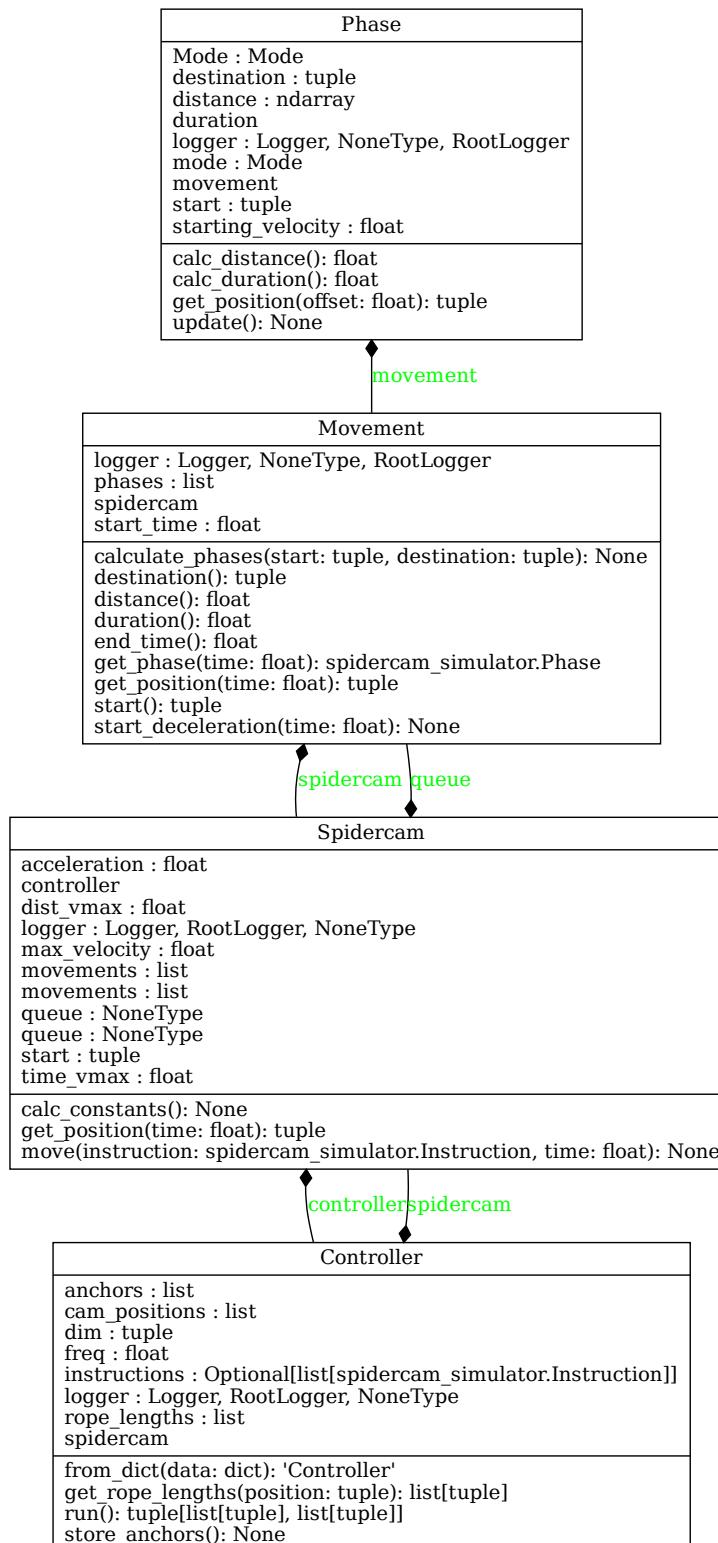


Abbildung 17: Klassendiagramm des Programms

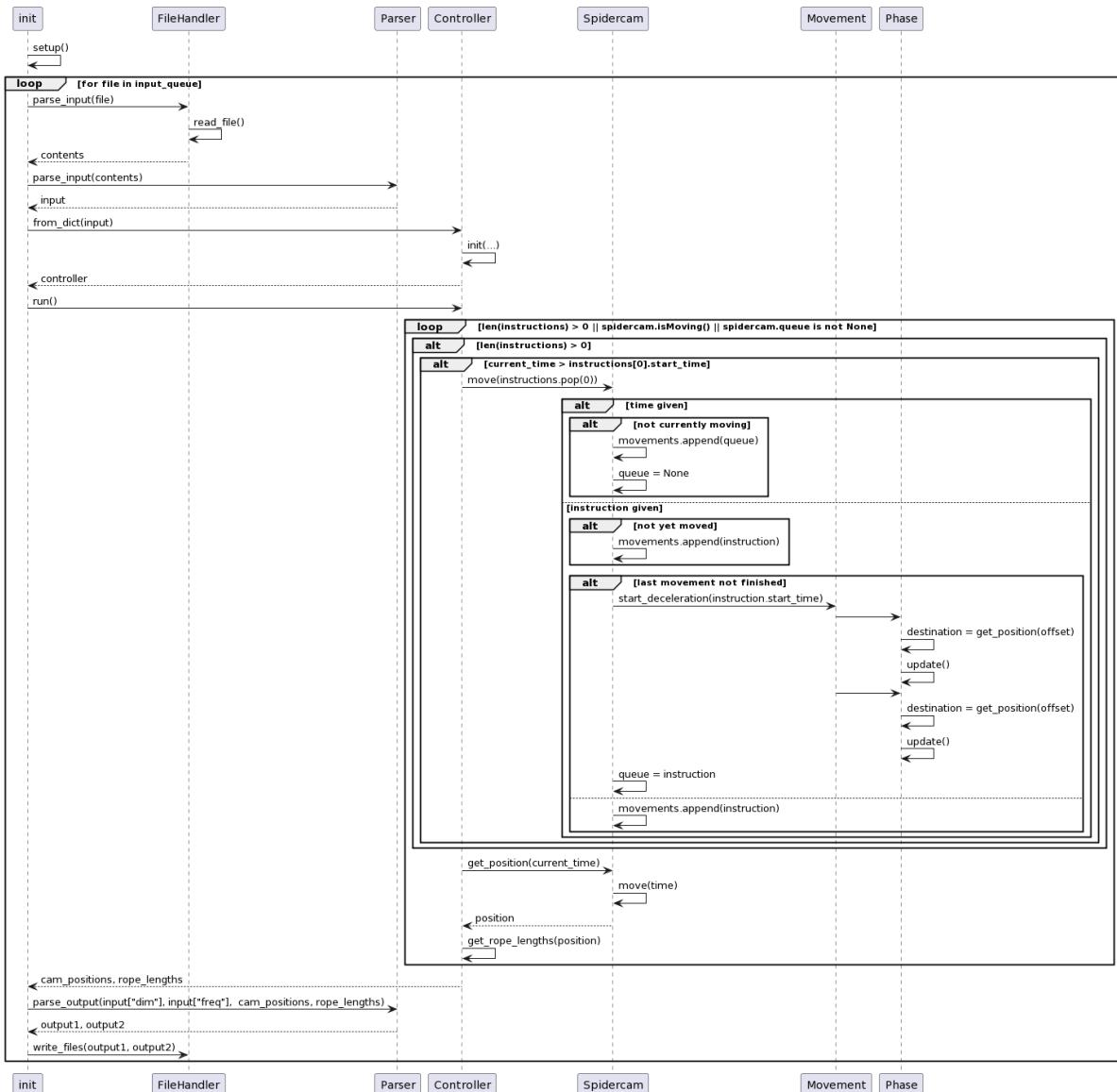


Abbildung 18: Aufrufhierarchie des Programms

3.2 Beispielablauf

Gegeben sei der Input der Datei `IHK_2.txt`, siehe D.

- In der `__main__.py` wird mithilfe der Klassen `FileHandler` und `Parser` die Datei eingelesen und damit eine `Controller`-Instanz erzeugt, die wiederum ebenfalls eine `Spidercam`-Instanz erzeugt.
- Mithilfe der Funktion `run()` wird nun der Kern des Programmablaufs gestartet.
- Die `Spidercam` befindet sich zum Start an den Koordinaten $s = (10, 80, 10)$.
- Die Maximalgeschwindigkeit der `Spidercam` ist $v_{\max} = 6$ mit einer Beschleunigung von $a = 2$.
- Die Zeit wird auf 0 gesetzt und von nun an in 0.5-Sekunden-Schritten erhöht.

- Für den Zeitpunkt 0 existiert eine Instruktion mit Zielkoordinaten $k_1 = (50, 40, 30)^T$.
- Der Controller weist mit `move(instruction)` die Spidercam an, sich zu diesen Koordinaten zu bewegen.
- Sie überprüft, ob die bisherige Bewegung abgeschlossen ist. Dies ist der Fall, da bisher noch keine Bewegung stattgefunden hat.
- Die leere Liste der Bewegungen `movements` wird nun mit einer neuen Instanz der Klasse `Movement` erweitert, initialisiert mit den Startkoordinaten der Spidercam und den Zielkoordinaten der Instruktion.
 - Anhand der Distanz $d(s, k_1)$ wird bestimmt, wie viele Phasen die Bewegung benötigt.
 - Es gilt $d(s, k_1) = \|(k_1 - s)\| = 60$.
 - Da $d(s, k_1) > 2 \cdot v_{\max} \cdot a^{-1}$, wird die Bewegung in drei Phasen unterteilt.
 - Die erste Phase ist eine Beschleunigungsphase mit Start s und Ziel

$$p_1 = s + \frac{v_{\max} \cdot a^{-1}}{d(s, k_1)} \cdot (k_1 - s) = (16, 74, 13)^T$$

- Die zweite Phase ist eine Konstantgeschwindigkeitsphase mit Start p_1 und Ziel
- $$p_2 = k_1 - \frac{v_{\max} \cdot a^{-1}}{d(s, k_1)} \cdot (k_1 - s) = (44, 46, 27)^T$$
- Die dritte Phase ist eine Bremsphase mit Start p_2 und Ziel k_1 .
 - Die Phasen werden nun in die Bewegung eingefügt.

- Für alle Zeitpunkte bis zur nächsten Instruktion beim Zeitpunkt 20 werden nun die Position der Spidercam berechnet.
 - Der Controller ruft mit die Funktion `get_position(time)` der Spidercam auf.
 - Die Spidercam bestimmt, ob der Zeitpunkt in, zwischen, vor oder nach einer Bewegung liegt.
 - Die Fälle zwischen, vor und nach sind trivial, da die Spidercam in diesen Fällen einfach die Start- bzw. Endposition der Bewegung zurückgibt.
 - Wird das Movement gefunden, in dem sich der Zeitpunkt befindet, bestimmt dieses die Position der Spidercam.
 - * Es wird bestimmt, in welcher Phase sich der Zeitpunkt befindet und welcher Offset t_o innerhalb der Phase erreicht wurde.
 - * Entsprechend der Phase, der Startgeschwindigkeit v_s und des Offsets kann die Position der Spidercam berechnet werden:

$$s(t_o) = \begin{cases} \frac{a \cdot t_o^2}{2} & \text{für die Beschleunigungsphase} \\ v_{\max} \cdot t & \text{für die Konstantgeschwindigkeitsphase} \\ v_s \cdot t_o - \frac{a \cdot t_o^2}{2} & \text{für die Bremsphase} \end{cases}$$

- * Die Position entspricht dann der Startposition der Bewegung plus der entsprechend skalierten Richtung.

- Mithilfe der Position der Spidercam werden nun auch die Längen der Drahtseile bestimmt und gespeichert.
- Für den Zeitpunkt 20 existiert eine Instruktion mit Zielkoordinaten $k_2 = (10, 80, 10)^T$.
- Die Spidercam hat ihre bisherige Bewegung abgeschlossen.
- movements wird nun analog zu vorher mit einer neuen Movement-Instanz erweitert.
- Für die Zeitpunkte bis zur nächsten Instruktion beim Zeitpunkt 22 werden nun die Positionen der Spidercam und Längen der Seile wie oben berechnet.
- Zum Zeitpunkt 22 existiert eine Instruktion mit Zielkoordinaten $k_3 = (50, 40, 30)^T$.
- Die Spidercam hat ihre bisherige Bewegung noch nicht abgeschlossen.
- Die Spidercam ruft die Funktion `start_deceleration(time)` auf dem letzten Movement auf.
 - Es wird die Phase der Bewegung bestimmt, in der sich der Zeitpunkt befindet.
 - Der Endpunkt dieser Phase wird auf den aktuellen Zeitpunkt gesetzt und die Phase geupdated.
 - Die folgende Phase wird zu einer Bremsphase abgeändert und das Ziel entsprechend [1.3](#) bestimmt.
- Mithilfe der aktualisierten Bewegung wird nun ein neues Movement erzeugt mit dem Ende der letzten Bewegung als Start ($s' = (44.66, 45.33, 27.33)^T$) und Ziel k_3 .
- Die Entfernung der beiden Koordinaten ist kleiner als $2 \cdot v_{\max} \cdot a^{-1}$, sodass die Bewegung zweiphasig ist.
 - Die erste Phase ist eine Beschleunigungsphase mit Start s und Ziel

$$p_1 = s + \frac{d(s', k_3)}{2} \cdot (k_3 - s') = (47.33, 45.33, 27.33)^T$$

- Die zweite Phase ist eine Konstantgeschwindigkeitsphase mit Start p_1 und Ziel k_3 .
- Dieses neue Objekt wird nun als queue der Spidercam gespeichert.
- Erneut erfolgt die Berechnung der Positionen der Spidercam und Längen der Seile für die Zeitpunkte bis zur nächsten Instruktion beim Zeitpunkt 23.
- Zum Zeitpunkt 23 existiert eine Instruktion mit Zielkoordinaten $k_4 = (30, 50, 20)^T$.
- Die Spidercam hat ihre bisherige Bewegung noch nicht abgeschlossen, bremst allerdings schon ab. Die letzte Bewegung muss also nicht aktualisiert werden.
- Analog wird nun wieder ein Movement erzeugt und die queue der Spidercam überschrieben.
- Die Bestimmung der Zeitpunkte erfolgt wieder bis zum Zeitpunkt 27.5.
- Zum Zeitpunkt 27.5 existiert eine Instruktion mit Zielkoordinaten $k_5 = (10, 80, 20)^T$.
- Die bisherige Bewegung ist abgeschlossen, die Berechnung funktioniert also analog zur zweiten Instruktion.
- Nach Abschluss dieser letzten Instruktion terminiert die `run()`-Funktion des

Controllers.

- Die Funktion gibt die Liste der berechneten Kamerapositionen und Längen der Drahtseile zurück, die dann entsprechend in Ausgabedateien geschrieben werden.

4 Testing

4.1 Testfälle

Die Testfälle sind in Abbildung 19 aufgelistet.

Die konkreten Testdateien sind in D aufgelistet.

Testfall	Beschreibung	Bemerkung
Normalfälle		
IHK_1	IHK-Beispiel 1	
IHK_2	IHK-Beispiel 2	
spiral	„Spiralförmige“ Bewegung	
single	Eine einzige Bewegung	
Grenzfälle		
diagonals	Bewegung zwischen Eckpunkten	
edges	Bewegung am Rand	
multiple_per_step	Mehrere Instruktionen in Zeitintervall	
multiple_simultaneously	Mehrere Instruktionen gleichzeitig	
same_dest	Instruktionen mit gleichem Ziel	
Fehlerfälle		
invalid_dim	dim ungültig	Koordinate kleiner 0
invalid_vmax	vmax ungültig	vmax kleiner 0
invalid_amax	amax ungültig	amax kleiner 0
invalid_freq	freq ungültig	freq kleiner 0
invalid_instruction	Ungültige Instruktion	Koordinate außerhalb
invalid_no_instruction	Keine Instruktion	

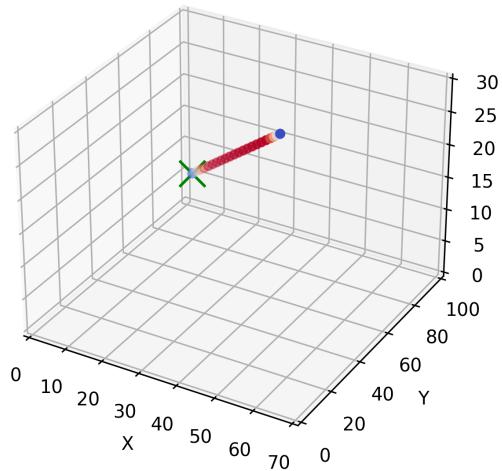
Abbildung 19: Testfälle

4.2 Beobachtungen

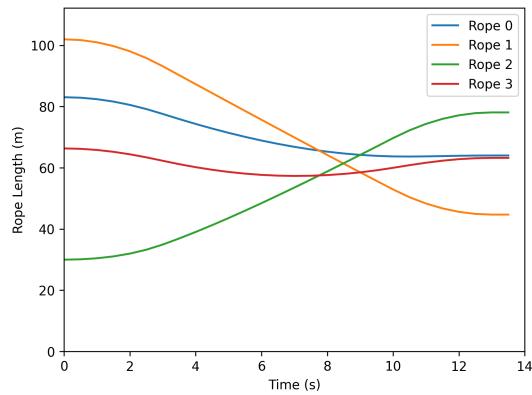
Für alle definierten Testfälle wurden die Ergebnisse mit den erwarteten Ergebnissen verglichen. Es konnte kein Fehlverhalten festgestellt werden. Das bedeutet:

- Fehlerfälle wurden korrekt erkannt und abgefangen.
- Grenz- und Normalfälle wurden korrekt berechnet.

Einige Ergebnisse sind in den folgenden Abbildungen dargestellt.

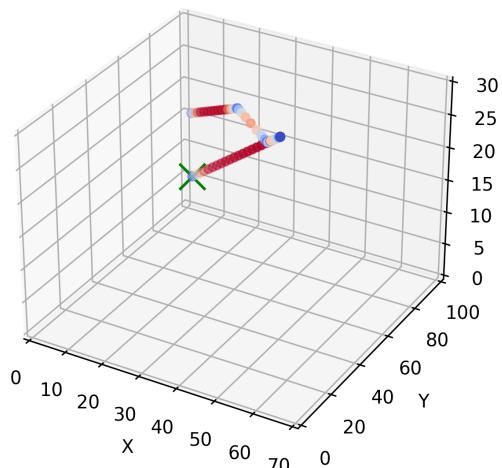


(a) Bewegung der Spidercam für Test IHK_1

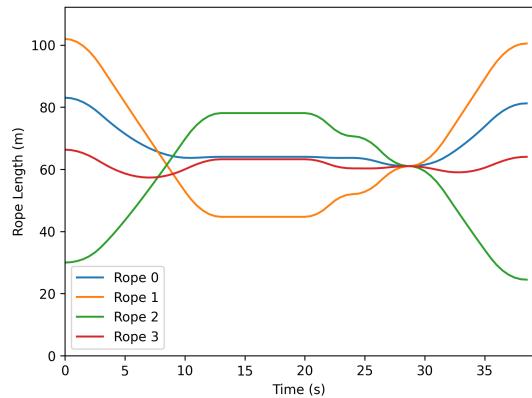


(b) Veränderung der Seillängen für Test IHK_1

Abbildung 20: Ergebnisse für Test IHK_1



(a) Bewegung der Spidercam für Test IHK_2



(b) Veränderung der Seillängen für Test IHK_2

Abbildung 21: Ergebnisse für Test IHK_2

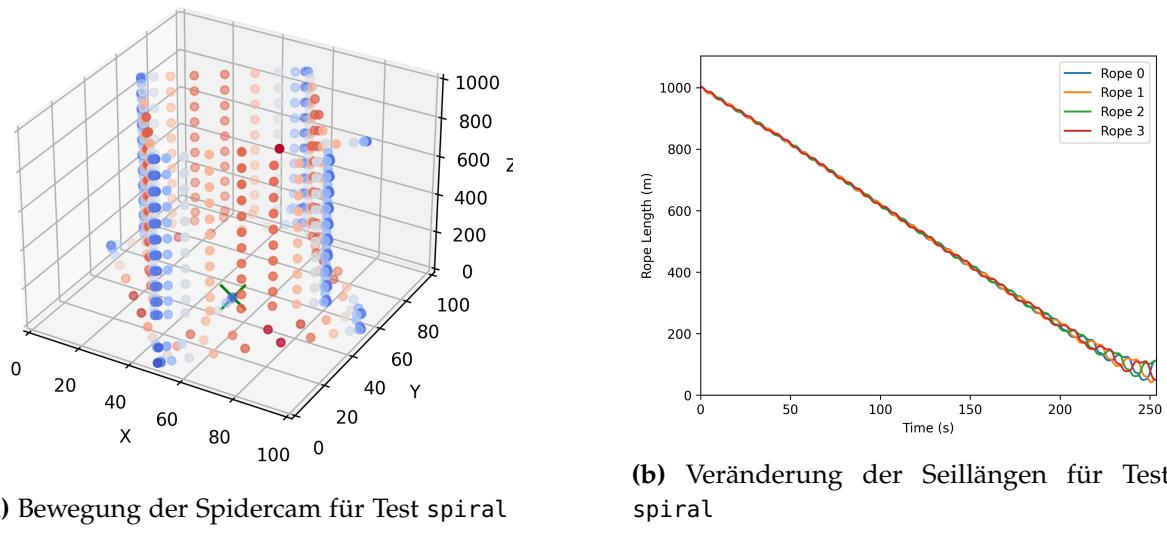


Abbildung 22: Ergebnisse für Test spiral

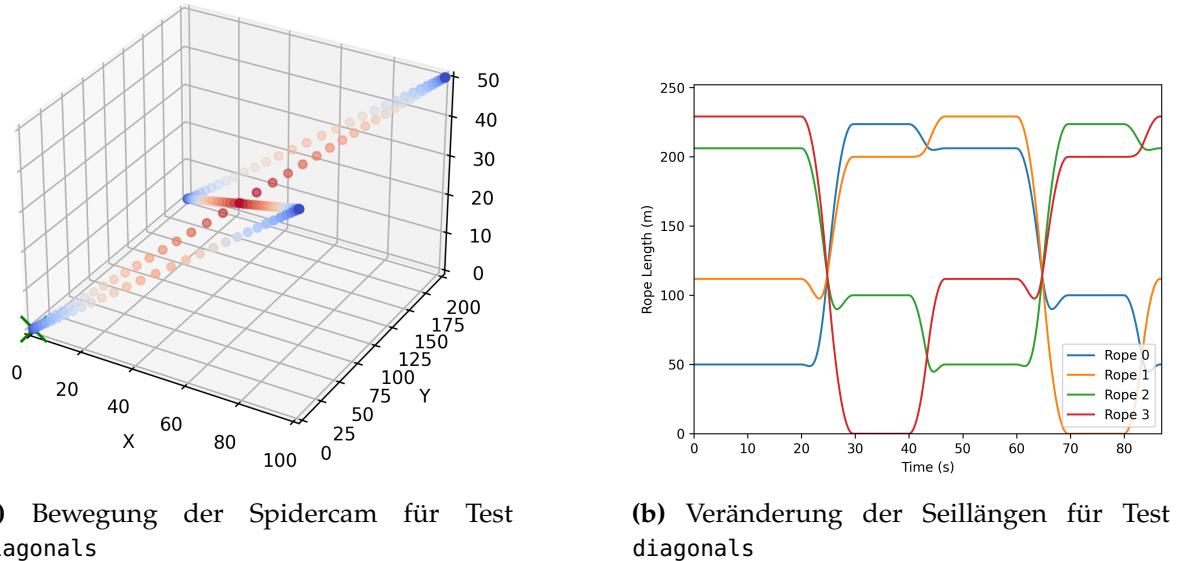
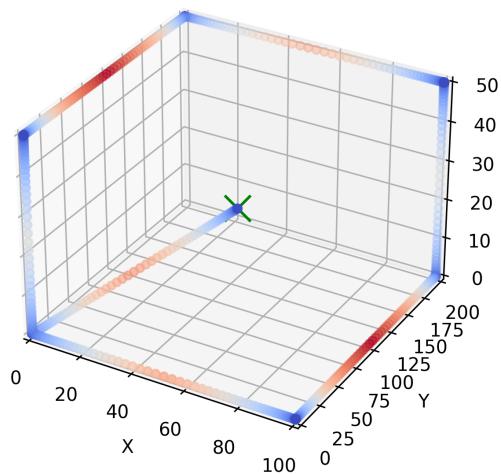
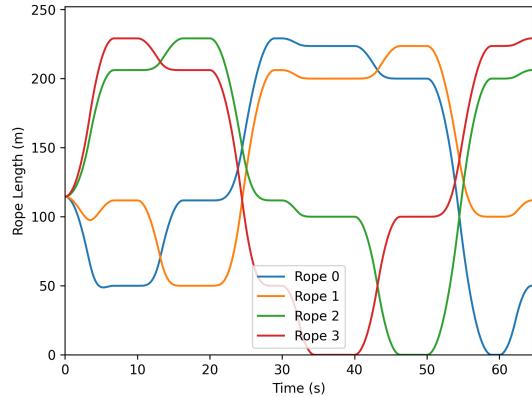


Abbildung 23: Ergebnisse für Test diagonals

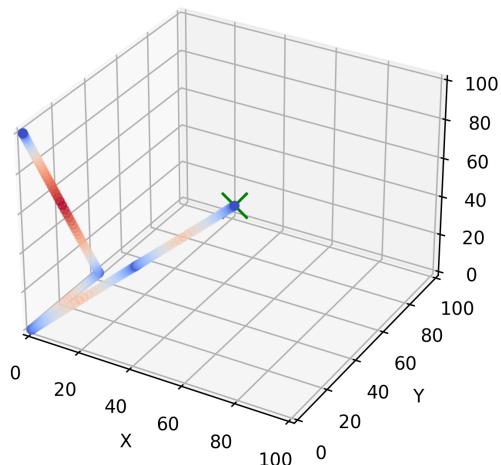


(a) Bewegung der Spidercam für Test edges

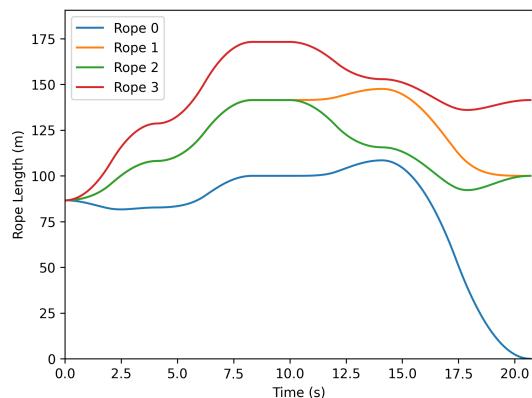


(b) Veränderung der Seillängen für Test edges

Abbildung 24: Ergebnisse für Test edges



(a) Bewegung der Spidercam für Test multiple_per_step



(b) Veränderung der Seillängen für Test multiple_per_step

Abbildung 25: Ergebnisse für Test multiple_per_step

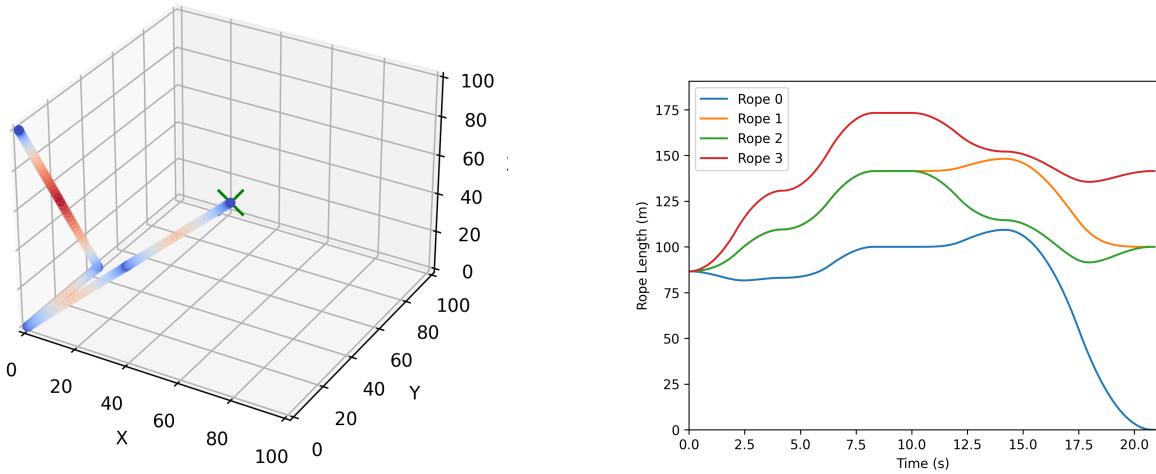


Abbildung 26: Ergebnisse für Test `multiple_simultaneously`

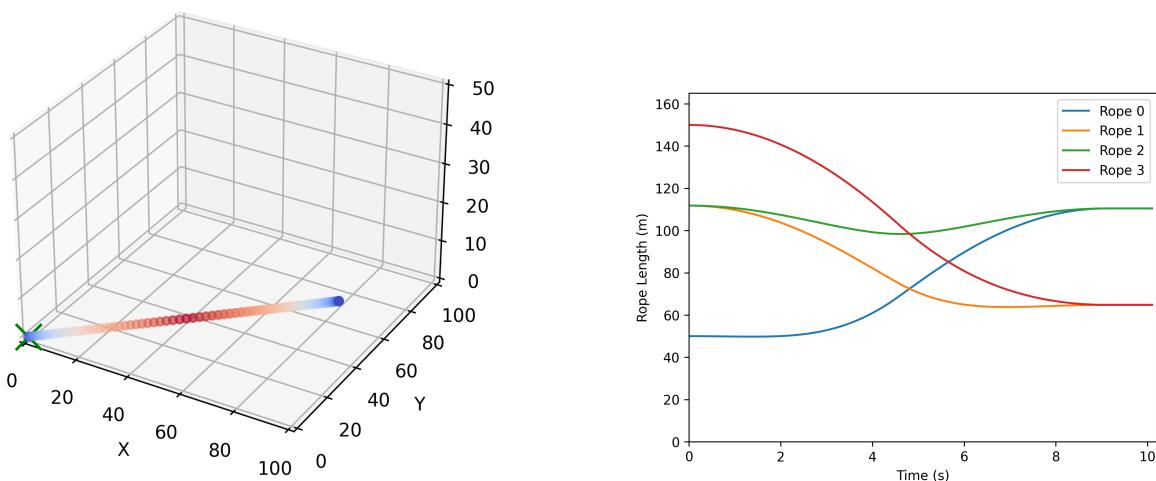


Abbildung 27: Ergebnisse für Test `test_same_dest`

Besonders interessant sind die letzten beiden Beispiele. Hier wird deutlich, dass die Spidercam auch bei mehreren gleichzeitig auftretenden Bewegungen korrekt funktioniert.

5 Abweichungen vom ursprünglichen Konzept

Im Laufe der Entwicklung des Programms sind einige Abweichungen vom ursprünglichen Konzept aufgetreten. Im Allgemeinen sind diese Abweichungen aufgrund von erkannter Redundanz oder aufgrund von technischen Limitierungen bzw. Verbesserungen entstanden.

Redundant waren beispielsweise:

- Abspeichern von Start- und Endkoordinaten in der Movement-Klasse, da diese Informationen bereits in den zugehörigen Phasen enthalten sind.
- Analog das Abspeichern von Distanz und Dauer in der Movement-Klasse.
- Dimension, Frequenz und Queue sind nun sinnvoller gespeichert als im ursprünglichen Entwurf.

Von den Algorithmen wurde nur minimal abgewichen. Teilweise wurden leicht andere Formeln zur Berechnung der Bewegungen verwendet um potentielle Rundungsfehler durch Operationen auf Gleitkommazahlen zu minimieren.

Insbesondere bei der Berechnung von Distanzen, Zeiten und Geschwindigkeiten wurde bevorzugt auf die Verwendung von numpy-Arrays und -funktionen zurückgegriffen. Diese sind deutlich schneller, einfacher und robuster als die selbstgeschriebenen Alternativen.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wurde ein Programm zur Simulation einer Spidercam konzipiert und implementiert. Die Simulation wurde in der Programmiersprache Python realisiert. Entstanden ist ein Python-Modul, welches die Simulation einer Spidercam in einem beliebigen 3D-Raum ermöglicht. Eingabedateien können sowohl einzeln, als auch automatisch nacheinander verarbeitet werden.

Alle relevanten Testfälle wurden erfolgreich durchlaufen. Insbesondere wurde dies erkenntlich durch das Überprüfen der erstellten Logs in `python/logs/` und der erstellten Visualisierungen.

6.2 Ausblick

In Zukunft könnte man zur Optimierung beispielweise untersuchen, ob folgende Punkte Vorteile bringen:

- Die Simulation in Echtzeit laufen lassen und Instruktionen live übertragen.
- Bei nötigen Bremsphasen die Zielkoordinaten anpassen, sodass kürzere Wege zurückgelegt werden müssen.
- Bei nötigen Brems- und Beschleunigungsphasen die Beschleunigung z. B. linear anpassen, damit die Bewegung nicht abrupt endet oder startet.
- Die Anwendung in eine GUI einbinden.
- Berechnungen parallelisieren.
- Eine noch performantere Programmiersprache verwenden.

Anhang

A Benutzeranleitung

A.1 Installation

A.1.1 Installation mit conda

Zur Installation mit conda wird eine verwendbare conda-Instanz benötigt.

Empfohlen wird Miniconda: <https://docs.conda.io/en/latest/miniconda.html>.

Die Installation funktioniert dann wie folgt:

1. Navigation in das Projektverzeichnis /python/.
2. Erstellen der conda-Umgebung mithilfe der dort hinterlegten environment.yml:
`conda env create -f environment.yml`
3. Aktivieren der conda-Umgebung:
`conda activate spidercam_simulator`
4. Installieren der spidercam_simulator-Bibliothek:
`pip install -e .`

A.1.2 Installation mit pip

Zur Installation mit pip wird eine verwendbare python-Instanz benötigt.

Empfohlen wird Python 3.10: <https://www.python.org/downloads/>.

Die Installation funktioniert dann wie folgt:

1. Navigation in das Projektverzeichnis /python/.
2. Installieren der benötigten Bibliotheken:
`pip install -r requirements.txt`
3. Installieren der spidercam_simulator-Bibliothek:
`pip install -e .`

A.2 Benutzung

Das Modul kann nach der Installation wie folgt ausgeführt werden:

```
python -m spidercam_simulator
```

Dabei wird das Modul mit den Standardwerten ausgeführt, die in der Datei config.ini hinterlegt sind. Es wird empfohlen, diese Datei nicht abzuändern.

Es werden standardmäßig alle Dateien aus dem Verzeichnis python/input/ verarbeitet. Die Ausgabe erfolgt in das Verzeichnis python/output/. Erstellt werden:

- Eine .csv-Datei entsprechend der Ausgabedatei 1, siehe 1.2. Sie endet auf _1.csv.
- Eine .csv-Datei entsprechend der Ausgabedatei 2, siehe 1.2. Sie endet auf _2.csv.
- Ein 3D-Plot der Bewegung der Spidercam. Die Datei endet auf _cam_pos.png.
- Ein 2D-Plot der Längen der Drahtseile. Die Datei endet auf _rope_lengths.png.

Folgende Argumente können an das Modul übergeben werden:

- -input, -i: Pfad zu einer Eingabedatei oder einem Verzeichnis.
- -output, -o: Pfad zu einem Ausgabeverzeichnis.
- -debug, -d: Aktiviert den Debug-Modus.⁵
- -no-plot, -np: Deaktiviert die Erstellung der Plots.

Im Ordner python/logs/ befindet sich eine Log-Datei, die Informationen über die Ausführung des Moduls enthält. Sie wird bei jedem Programmstart überschrieben.

A.2.1 Beispiele

- Ausführen des Moduls mit den Standardwerten:

```
python -m spidercam_simulator
```

- Ausführen des Moduls mit Debug-Modus:

```
python -m spidercam_simulator -d
```

- Ausführen des Moduls mit Debug-Modus und Deaktivierung der Plots:

```
python -m spidercam_simulator -d -np
```

- Ausführen des Moduls mit einer Eingabedatei:

```
python -m spidercam_simulator -i input_file.csv
```

- Ausführen des Moduls mit einem Eingabeverzeichnis:

```
python -m spidercam_simulator -i input_dir/
```

- Ausführen des Moduls mit einem Ausgabeverzeichnis:

```
python -m spidercam_simulator -o output_dir/
```

⁵Dabei werden zusätzliche Informationen geloggt.

A.2.2 Fehlermeldungen

Folgende Fehlermeldungen können auftreten:

- `FileNotFoundException`: Eine angegebene Datei oder ein angegebenes Verzeichnis konnte nicht gefunden werden.
- `ValueError`: Eine Eingabedatei enthält ungültige Werte.

A.2.3 Skripte

Im Ordner `python/scripts/` befinden sich Skripte, die die Ausführung des Moduls erleichtern. Sie können z. B. mit `sh scripts/$name.sh` ausgeführt werden.

Folgende Skripte sind vorhanden und relevant:

- `generate_docs.sh`: Generiert die Entwicklerdokumentation.
- `generate.uml.sh`: Generiert die UML-Klassendiagramme, die hier verwendet wurden.
- `run.sh`: Führt das Modul mit den Standardwerten aus. Es werden alle Dateien aus dem Verzeichnis `python/input/` verarbeitet.
- `run_dev.sh`: Führt das Modul mit den Standardwerten aus. Es werden alle Dateien aus dem Verzeichnis `python/input/` verarbeitet. Zusätzlich wird der Debug-Modus aktiviert.

B Entwicklerdokumentation

Die Entwicklerdokumentation ist im Ordner `python/docs` zu finden.

Zum Generieren der Dokumentation wird das Tool `pdoc` verwendet. Dieses Tool generiert aus den Python-Dateien automatisch eine Dokumentation in HTML-Format mithilfe der Docstrings. Die Dokumentation kann mit dem Befehl in 28 generiert werden.⁶

Genauere Instruktionen sind in der Datei `python/README.md` zu finden.

```
1 pdoc spidercam_simulator -o docs --docformat 'google'
```

Abbildung 28: Generieren der Dokumentation

⁶Vorausgesetzt, dass das Tool `pdoc` installiert ist und im Ordner `python` ausgeführt wird.

C Hilfsmittel

Die genutzten Hilfsmittel sind in Abbildung 29 aufgeführt.

Sonstige benutzte Python-Module sind in der Datei `python/requirements.txt` zu finden.

Typ	Tool	Hinweise
CPU	AMD Ryzen 5 3600	6x 3.60 GHz
RAM	32 GB	DDR4-3200
Betriebssystem	Ubuntu	22.04.1 LTS
Kernel	Linux	5.15.74.2-microsoft-standard-WSL2
IDE	Visual Studio Code	1.73.1
Compiler	Python	3.10.8
Linter	pylint	2.15.7
Dokumentation	pdoc	12.3.0
UML-Diagramme	PlantUML	V1.2022.13

Abbildung 29: Hilfsmittel

D Testdateien

```
1 # Beispiel
2 dim 70 100 30
3 start 10 80 10
4 vmax 6
5 amax 2
6 freq 2
7 0 50 40 30
```

Listing 3: Testdatei IHK_1.txt

```
1 # Beispiel
2 dim 70 100 30
3 start 10 80 10
4 vmax 6
5 amax 2
6 freq 2
7 0 50 40 30          # 1:
8 20 10 80 10          # 2: Anweisung beginnt nach Ende der vorherigen
9 22 50 40 30          # 3: Die Bremsung der vorherigen Anweisung wird eingeleitet
10 23 35 50 30         # 4: Bremsung von Anweisung 2 noch nicht beendet -> Anweisung
   3 wird ignoriert
11 27.5 10 80 20      # 5: Anweisung 5 endete nicht zu den diskreten Zeitpunkten -> offset
   beachten
```

Listing 4: Testdatei IHK_2.txt

```
1 # Example which travels in an approximate spiral
2 dim 100 100 1000
3 start 50 50 0
4 vmax 50
5 amax 10
6 freq 2
7 # t x y z
8 0 50 0 20
9 5 100 50 40
10 10 50 100 60
11 15 0 50 80
12 20 50 0 100
13 25 100 50 120
14 30 50 100 140
15 35 0 50 160
16 40 50 0 180
17 45 100 50 200
18 50 50 100 220
19 55 0 50 240
20 60 50 0 260
21 65 100 50 280
22 70 50 100 300
23 75 0 50 320
24 80 50 0 340
25 85 100 50 360
```

```

26 90 50 100 380
27 95 0 50 400
28 100 50 0 420
29 105 100 50 440
30 110 50 100 460
31 115 0 50 480
32 120 50 0 500
33 125 100 50 520
34 130 50 100 540
35 135 0 50 560
36 140 50 0 580
37 145 100 50 600
38 150 50 100 620
39 155 0 50 640
40 160 50 0 660
41 165 100 50 680
42 170 50 100 700
43 175 0 50 720
44 180 50 0 740
45 185 100 50 760
46 190 50 100 780
47 195 0 50 800
48 200 50 0 820
49 205 100 50 840
50 210 50 100 860
51 215 0 50 880
52 220 50 0 900
53 225 100 50 920
54 230 50 100 940
55 235 0 50 960
56 240 50 0 980
57 245 100 50 1000

```

Listing 5: Testdatei test_spiral.txt

```

1 # Beispiel einer einzigen Instruktion
2 dim 100 100 50
3 start 0 0 0
4 vmax 25
5 amax 5
6 freq 10
7 0 90 50 10

```

Listing 6: Testdatei test_single.txt

```

1 # Example which travels diagonally
2 dim 100 200 50
3 start 0 0 0
4 vmax 50
5 amax 10
6 freq 5
7 # t x y z

```

```

8 20 100 200 50
9 40 0 200 0
10 60 100 0 50
11 80 0 0 0

```

Listing 7: Testdatei test_diagonals.txt

```

1 # Example which travels to each edge in the graph once after 10 seconds
2 dim 100 200 50
3 start 50 100 25
4 vmax 50
5 amax 10
6 freq 10
7 # t x y z
8 0 0 0 0
9 10 100 0 0
10 20 100 200 0
11 30 100 200 50
12 40 0 200 50
13 50 0 0 50
14 60 0 0 0

```

Listing 8: Testdatei test_edges.txt

```

1 # Example which has multiple instructions within a discrete timespan
2 dim 100 100 100
3 start 50 50 50
4 vmax 50
5 amax 10
6 freq 10
7 0 0 0 0
8 2.05 100 100 100 # opposite direction
9 2.1 0 0 0 # should continue in the previous direction
10 10 0 100 0
11 12.05 0 100 100 # slight change in direction
12 12.1 0 0 100 # should continue with this direction

```

Listing 9: Testdatei test_multiple_per_step.txt

```

1 # Example which has multiple instructions within a discrete timespan
2 dim 100 100 100
3 start 50 50 50
4 vmax 50
5 amax 10
6 freq 10
7 0 0 0 0
8 2.1 100 100 100 # opposite direction
9 2.1 0 0 0 # should continue in the previous direction
10 10 0 100 0
11 12.1 0 100 100 # slight change in direction
12 12.1 0 0 100 # should continue with this direction

```

Listing 10: Testdatei test_multiple_simultaneously.txt

```

1 # Example where two consecutive instructions have the same destination
2 dim 100 100 50
3 start 0 0 0
4 vmax 25
5 amax 5
6 freq 10
7 0 90 50 10
8 10 90 50 10

```

Listing 11: Testdatei test_same_dest.txt

```

1 # Example to test invalid dimension
2 dim -1 100 30
3 start 10 80 10
4 vmax 6
5 amax 2
6 freq 2
7 0 50 40 30
8 20 10 80 10
9 22 50 40 30
10 23 35 50 30
11 27.5 10 80 20

```

Listing 12: Testdatei test_invalid_dim.txt

```

1 # Example to test velocity limit
2 dim 100 100 30
3 start 10 80 10
4 vmax -1
5 amax 2
6 freq 2
7 0 50 40 30
8 20 10 80 10
9 22 50 40 30
10 23 35 50 30
11 27.5 10 80 20

```

Listing 13: Testdatei test_invalid_vmax.txt

```

1 # Example to test acceleration limit
2 dim 100 100 30
3 start 10 80 10
4 vmax 6
5 amax -1
6 freq 2
7 0 50 40 30
8 20 10 80 10
9 22 50 40 30
10 23 35 50 30
11 27.5 10 80 20

```

Listing 14: Testdatei test_invalid_amax.txt

```
1 # Example to test invalid frequency
2 dim 100 100 30
3 start 10 80 10
4 vmax 6
5 amax 2
6 freq -1
7 0 50 40 30
8 20 10 80 10
9 22 50 40 30
10 23 35 50 30
11 27.5 10 80 20
```

Listing 15: Testdatei test_invalid_freq.txt

```
1 # Example to test instruction out of range
2 dim 100 100 30
3 start 10 80 10
4 vmax 6
5 amax 2
6 freq -1
7 0 50 40 30
8 20 10 80 40
9 22 50 40 30
10 23 35 50 30
11 27.5 10 80 20
```

Listing 16: Testdatei test_invalid_instruction.txt

```
1 # Example to test missing instruction
2 dim 100 100 30
3 start 10 80 10
4 vmax 6
5 amax 2
6 freq -1
```

Listing 17: Testdatei test_invalid_no_instruction.txt

E Quellcode

```
1 """  
2 .. include:: ../../README.md  
3 """  
  
4  
5 from .controller import *  
6 from .io import *  
7 from .movement import *  
8 from .phase import *  
9 from .spidercam import *  
10 from .plotter import *
```

Listing 18: Quellcode für /python/spidercam_simulator/__init__.py

```
1 import argparse  
2 import configparser  
3 import logging  
4 import os  
5 import sys  
6  
7 import spidercam_simulator  
8  
9  
10 def setup():  
11     """Sets up the logger and config"""  
12  
13     # reading the config file  
14     cfg = configparser.ConfigParser()  
15     # parent of os.path.dirname(__file__)  
16     cfg_file = os.path.join(os.path.dirname(os.path.dirname(__file__)), "config.ini")  
17     print(f"Config file: {cfg_file}")  
18     cfg.read(cfg_file)  
19  
20     # setting up the logger  
21     logging_dir = spidercam_simulator.find_location(cfg["io"]["logging_dir"])  
22  
23     logging.basicConfig(  
24         level=logging.INFO,  
25         format="%(asctime)s - %(name)s - %(levelname)s - %(message)s (%(filename)s:%(lineno)d)",  
26         datefmt="%Y/%m/%d %I:%M:%S %p",  
27         handlers=[  
28             logging.FileHandler(  
29                 filename=os.path.join(  
30                     os.path.dirname(os.path.dirname(os.path.abspath(__file__))),  
31                     "logs",  
32                     "spidercam_simulator.log",  
33                     ),  
34                 mode="w",  
35             ),  
36         ],
```

```

37 )
38
39 logging.info("Logging initialized")
40 print(f"Logging to {os.path.join(logging_dir, 'spidercam_simulator.log')}")
41
42 # Logging configuration
43 for section in cfg.sections():
44     for key, value in cfg[section].items():
45         logging.debug("Config: %s.%s = %s", section, key, value)
46
47 logging.debug("Config file read")
48
49 # Setup: Setting up Arguments
50 logging.info("Reading arguments")
51
52 parser = argparse.ArgumentParser()
53 parser.add_argument("--input", "-i", help="File or directory to parse")
54 parser.add_argument(
55     "--output", "-o", help="Output directory (needs to be different from input)"
56 )
57 parser.add_argument("--debug", "-d", help="Debug mode", action="store_true")
58 parser.add_argument(
59     "--no-plot", "-np", help="Disable plotting", action="store_true"
60 )
61 args = parser.parse_args()
62
63 if args.debug:
64     # logging.getLogger().addHandler(logging.StreamHandler())
65     logging.getLogger().setLevel(logging.DEBUG)
66
67     logging.info("Debug mode enabled")
68
69 logging.debug("Arguments read")
70
71 # logging configuration
72 for key, value in vars(args).items():
73     logging.debug("Argument: %s = %s", key, value)
74
75 # use defaults if no arguments are passed
76 if args.input is None:
77     args.input = cfg["io"]["input_dir"]
78
79 args.input = spidercam_simulator.find_location(args.input)
80
81 logging.debug("Input file/directory: %s", args.input)
82
83 if args.output is None:
84     args.output = cfg["io"]["output_dir"]
85
86 args.output = spidercam_simulator.find_location(args.output)
87

```

```

88     logging.debug("Output directory: %s", args.output)
89
90     # check if files/directories exist
91     if not os.path.exists(args.input):
92         logging.error("Input file/directory %s does not exist", args.input)
93         sys.exit(1)
94
95     if not os.path.exists(args.output):
96         logging.error(
97             "Output directory %s does not exist. Please create it first.", args.output
98         )
99         sys.exit(1)
100
101    # check if input and output are the same
102    if os.path.samefile(args.input, args.output):
103        logging.error("Input and output directories are the same")
104        sys.exit(1)
105
106    logging.info("Setup complete")
107    return cfg, args
108
109
110 config, args = setup()
111
112 logging.info("Starting spidercam_simulator")
113
114 input_queue = [args.input] if os.path.isfile(args.input) else os.listdir(args.input)
115
116 logging.info("Found %s files/directories to process", len(input_queue))
117
118 for file in input_queue:
119     if file == ".gitkeep":
120         continue
121
122     logging.info("Processing %s", file)
123
124     file_handler = spidercam_simulator.FileHandler(
125         os.path.join(args.input, file), args.output
126     )
127     contents = file_handler.read_file()
128
129     logging.debug("File contents: %s", contents)
130
131     # parse the file but skip if error
132     try:
133         input_dict = spidercam_simulator.Parser.parse_input(contents)
134     except ValueError as exc:
135         logging.error("Error parsing file: %s", exc)
136         print(f"Error parsing file {file}: {exc}")
137         continue
138

```

```

139     logging.debug("Input dictionary: %s", input_dict)
140
141     controller = spidercam_simulator.Controller.from_dict(input_dict)
142
143     logging.info("Running controller %s", repr(controller))
144
145     cam_positions, rope_lengths = controller.run()
146
147     logging.debug("Cam positions: %s", cam_positions)
148     logging.debug("Rope lengths: %s", rope_lengths)
149
150     output1, output2 = spidercam_simulator.Parser.parse_output(
151         input_dict["dim"], input_dict["freq"], cam_positions, rope_lengths
152     )
153
154     file_handler.write_files(output1, output2)
155
156     if not args.no_plot:
157         plotter = spidercam_simulator.Plotter(
158             input_dict["dim"],
159             input_dict["freq"],
160             cam_positions,
161             rope_lengths,
162             args.output,
163             os.path.splitext(file)[0],
164         )
165
166         plotter.plot()
167
168     # cleanup
169     del file_handler
170     del contents
171     del input_dict
172     del controller
173     del cam_positions
174     del rope_lengths
175     del output1
176     del output2
177
178     logging.info("Finished processing %s", file)
179
180     # file_handler.write_file(parsed)
181
182     # Ending the program
183     logging.info("Ending spidercam_simulator")

```

Listing 19: Quellcode für /python/spidercam_simulator/__main__.py

```

1 from __future__ import annotations
2
3 import logging
4 import numpy as np

```

```

5
6 import spidercam_simulator
7
8
9 class Controller:
10     """ A class for initializing and controlling the spidercam"""
11
12     def __init__(
13         self,
14         dim: tuple = (1, 1, 1),
15         start: tuple = (0, 0, 0),
16         max_velocity: float = 1.0,
17         acceleration: float = 1.0,
18         freq: float = 1.0,
19         instructions: list[spidercam_simulator.Instruction] = None,
20     ) -> None:
21         """ Initializes the Controller class
22
23         Args:
24             dim (tuple, optional): The dimensions of the field. Defaults to (1, 1, 1).
25             start (tuple, optional): The starting position of the spidercam. Defaults
26             to (0, 0, 0).
27             max_velocity (float, optional): The maximum velocity of the spidercam.
28             Defaults to 1.0.
29             acceleration (float, optional): The maximum acceleration of the spidercam.
29             Defaults to 1.0.
30             freq (float, optional): The discrete time frequency. Defaults to 1.0.
31             instructions (list, optional): The instructions for the spidercam.
31             Defaults to [].
32
33         Returns:
34             None
35
36         self.logger = logging.getLogger(__name__)
37         self.logger.debug(
38             "Initializing Controller with dim=%s, start=%s, max_velocity=%s,
39             acceleration=%s, freq=%s, instructions=%s",
40             dim,
41             start,
42             max_velocity,
43             acceleration,
44             freq,
45             instructions,
46         )
47
48         self.dim = dim
49         self.spidercam = spidercam_simulator.Spidercam(
50             self, max_velocity, acceleration, start
51         )
52         self.freq = freq
53         self.instructions = instructions

```

```

51
52     self.cam_positions = []
53     self.rope_lengths = []
54
55     self.store_anchors()
56
57     def store_anchors(self) -> None:
58         """ Stores the anchor positions in regard to the dimensions
59
60         Returns:
61             None
62         """
63
64         # Anchors are at the top corners of the field
65         self.anchors = [
66             (0, 0, self.dim[2]),
67             (self.dim[0], 0, self.dim[2]),
68             (0, self.dim[1], self.dim[2]),
69             (self.dim[0], self.dim[1], self.dim[2]),
70         ]
71
72     @classmethod
73     def from_dict(cls, data: dict) -> "Controller":
74         """ Creates a Controller object from a dictionary
75
76         Args:
77             data (dict): The dictionary to create the Controller object from
78                 - dim (tuple): The dimensions of the field
79                 - start (tuple): The starting position of the spidercam
80                 - max_velocity (float): The maximum velocity of the spidercam
81                 - acceleration (float): The maximum acceleration of the spidercam
82                 - freq (float): The discrete time frequency
83                 - instructions (list): The instructions for the spidercam
84
85         Returns:
86             Controller: The Controller object
87         """
88
89         cls.logger = logging.getLogger(__name__)
90         cls.logger.debug("Creating Controller from dict %s", data)
91         return cls(**data)
92
93     def __repr__(self) -> str:
94         """ Returns a string representation of the Controller object
95
96         Returns:
97             str: The string representation of the Controller object
98         """
99         return f"Controller({self.dim=}, {self.spidercam.start=}, {self.spidercam.
max_velocity=}, {self.spidercam.acceleration=}, {self.freq=}, {self.instructions
=})"
```

```

100
101 def run(self) -> tuple[list[tuple], list[tuple]]:
102     """ Runs the instructions for the spidercam
103
104     Returns:
105         tuple[list[tuple], list[tuple]]: The positions and rope lengths for each
106         time step
107         """
108
109     current_time = 0
110
111     while True:
112         self.logger.info("Current time: %s", current_time)
113
114         if len(self.instructions) != 0:
115             # Check if there is a new instruction
116             if current_time >= self.instructions[0].start_time:
117                 instruction = self.instructions.pop(0)
118
119                 self.spidercam.move(instruction)
120
121             # Store cam_positions and rope_lengths
122             position = self.spidercam.get_position(current_time)
123             self.logger.info("Current position: %s", position)
124
125             self.cam_positions.append(position)
126             self.rope_lengths.append(self.get_rope_lengths(position))
127
128             # Check if the last instruction is finished
129             if (
130                 len(self.instructions) == 0
131                 and self.spidercam.movements[-1].end_time() < current_time
132                 and self.spidercam.queue is None
133             ):
134                 self.logger.info(
135                     "Last instruction finished and no more instructions left"
136                 )
137                 break
138
139             current_time += 1 / self.freq
140
141         return self.cam_positions, self.rope_lengths
142
143
144 def get_rope_lengths(self, position: tuple) -> list[tuple]:
145     """ Returns the rope lengths for a given position
146
147     Returns:
148         list[tuple]: The rope lengths
149         """
150
151         rope_lengths = [

```

```

150     np.linalg.norm(np.array(anchor) - np.array(position))
151     for anchor in self.anchors
152 ]
153
154     self.logger.debug("Rope lengths: %s", rope_lengths)
155
156     return rope_lengths

```

Listing 20: Quellcode für /python/spidercam_simulator/controller.py

```

1 import logging
2 import os
3
4 import numpy as np
5
6
7 class FileHandler:
8     """ A class for handling input and output"""
9
10    def __init__(self, input_file: str, output_dir: str) -> None:
11        """Initializes the FileHandler class
12
13        Args:
14            input_file (str): The path to the file to read
15            output_dir (str): The path to the directory to write
16
17        Returns:
18            None
19        """
20
21    self.logger = logging.getLogger(__name__)
22    self.logger.debug(
23        "Initializing FileHandler with input_file %s and output_dir %s",
24        input_file,
25        output_dir,
26    )
27
28    self.input_file = input_file
29    self.output_dir = output_dir
30    # removes file ending from input
31    self.output_files = (
32        os.path.join(
33            output_dir, os.path.basename(input_file).split(".")[0] + "_1.csv"
34        ),
35        os.path.join(
36            output_dir, os.path.basename(input_file).split(".")[0] + "_2.csv"
37        ),
38    )
39
40    def read_file(self) -> str:
41        """Reads the file
42

```

```

43     Returns:
44         str: The contents of the file
45         """
46
47     self.logger.info("Reading file %s", self.input_file)
48
49     with open(self.input_file, "r", encoding="utf-8") as file:
50         return file.read()
51
52 def write_files(self, output1: str, output2: str) -> None:
53     """Writes the output to two files
54
55     Args:
56         output1 (str): The contents to write to the first file
57         output2 (str): The contents to write to the second file
58
59     Returns:
60         None
61         """
62
63     self.logger.info(
64         "Writing output to files %s and %s",
65         self.output_files[0],
66         self.output_files[1],
67     )
68
69     with open(self.output_files[0], "w", encoding="utf-8") as file:
70         file.write(output1)
71
72     with open(self.output_files[1], "w", encoding="utf-8") as file:
73         file.write(output2)
74
75
76 def find_location(file: str) -> str:
77     """ Finds the location of the given file/directory
78     Looks in the following locations:
79         - The current working directory
80         - The directory of the script
81         - The directory of the script's parent
82         - The home directory
83         - The root directory
84
85     Args:
86         file (str): The path to the given file/directory
87
88     Returns:
89         str: The absolute path to the given file/directory
90         """
91
92     try:
93         logger = logging.getLogger(__name__)

```

```

94     logger.debug("Finding location of file/directory %s", file)
95 except NameError:
96     pass
97
98 locations = [
99     os.getcwd(), # Current working directory
100    os.path.dirname(os.path.realpath(__file__)), # Directory of the script
101    os.path.dirname(
102        os.path.dirname(os.path.realpath(__file__)))
103    ), # Directory of the script's parent
104    os.path.expanduser("~/"), # Home directory
105    os.path.sep, # Root directory
106 ]
107
108 for location in locations:
109     logger.debug("Checking location %s", location)
110     if os.path.exists(os.path.join(location, file)):
111         # log absolute path
112         logger.debug("Found file/directory at %s", os.path.join(location, file))
113         return os.path.join(location, file)
114
115 raise FileNotFoundError(f"Could not find file/directory {repr(file)}")
116
117
118 class Parser:
119     """ A class for parsing input strings"""
120
121     logger = logging.getLogger(__name__)
122
123     @staticmethod
124     def parse_input(string: str) -> dict:
125         """Parses the given string
126
127         Example file contents:
128         ~~~
129
130         # Beispiel 2
131         dim 70 100 30
132         start 10 80 10
133         vmax 6
134         amax 2
135         freq 2
136         0 50 40 30      # 1:
137         20 10 80 10     # 2: Anweisung beginnt nach Ende der vorherigen
138         22 50 40 30     # 3: Die Bremsung der vorherigen Anweisung wird eingeleitet
139         23 35 50 30     # 4: Bremsung von Anweisung 2 noch nicht beendet -> Anweisung
140         3 wird ignoriert
141         27.5 10 80 20   # 5: Anweisung 5 endete nicht zu den diskreten Zeitpunkten ->
142         offset beachten
143         ~~~
144
145         Args:

```

```

143     string (str): The string to parse
144
145     Returns:
146         dict: The parsed string
147             - dim (tuple): The dimensions of the field
148             - start (tuple): The starting position of the spidercam
149             - max_velocity (float): The maximum velocity of the spidercam
150             - acceleration (float): The maximum acceleration of the spidercam
151             - freq (float): The discrete time frequency
152             - instructions (list): The instructions for the spidercam
153     """
154
155     Parser.logger.debug("Parsing string %s", string)
156
157     lines = string.splitlines()
158     instructions = []
159
160     # Clean up the lines and remove comments
161     for i, line in enumerate(lines):
162         lines[i] = line.split("#")[0].strip()
163
164     # Remove empty lines
165     lines = list(filter(None, lines))
166
167     # Parse the lines
168     for i, line in enumerate(lines):
169         if line.startswith("dim"):
170             dim = tuple(map(int, line.split()[1:]))
171         elif line.startswith("start"):
172             start = tuple(map(int, line.split()[1:]))
173         elif line.startswith("vmax"):
174             max_velocity = float(line.split()[1])
175         elif line.startswith("amax"):
176             acceleration = float(line.split()[1])
177         elif line.startswith("freq"):
178             freq = float(line.split()[1])
179         else:
180             instructions.append(Instruction.parse(line))
181
182     # Check if the dimensions are valid
183     if len(dim) != 3 or any(d <= 0 for d in dim):
184         raise ValueError("The dimensions are invalid")
185
186     # Check if the starting position is valid
187     if len(start) != 3 or not all(0 <= i <= j for i, j in zip(start, dim)):
188         raise ValueError("The starting position is invalid")
189
190     # Check if the maximum velocity is valid
191     if max_velocity <= 0:
192         raise ValueError("The maximum velocity is invalid")
193

```

```

194     # Check if the maximum acceleration is valid
195     if acceleration <= 0:
196         raise ValueError("The maximum acceleration is invalid")
197
198     # Check if the discrete time frequency is valid
199     if freq <= 0:
200         raise ValueError("The discrete time frequency is invalid")
201
202     # Check if the instructions are valid
203     if len(instructions) == 0:
204         raise ValueError("No instructions found")
205
206     for i, instruction in enumerate(instructions):
207         if not all(0 <= j <= k for j, k in zip(instruction.destination, dim)):
208             raise ValueError(f"Instruction {i + 1} is invalid")
209
210     return {
211         "dim": dim,
212         "start": start,
213         "max_velocity": max_velocity,
214         "acceleration": acceleration,
215         "freq": freq,
216         "instructions": instructions,
217     }
218
219     # returns two strings
220     @staticmethod
221     def parse_output(
222         dim: tuple,
223         freq: float,
224         cam_positions: list[tuple],
225         rope_lengths: list[tuple],
226     ) -> tuple:
227         """Parses the output
228
229         Example file contents (first output: lengths of the ropes):
230         """
231         83.06623, 82.9059, [...], 64.0056, 64.0312 # Rope 1
232         101.9803, 101.9803, [...], 101.9803, 101.9803 # Rope 2
233         101.9803, 101.9803, [...], 101.9803, 101.9803 # Rope 3
234         101.9803, 101.9803, [...], 101.9803, 101.9803 # Rope 4
235         """
236
237         Example file contents (second output: dimensions, time stamps, positions of
238         the camera):
239         """
240         70, 100, 30 # Dimensions
241         0.0, 0.5, 1.0, [...], 20.5, 21.0 # Time stamps
242         10.0, 10.16, 10.66, [...], 20.0, 20.0 # x coordinates
243         80.0, 80.0, 80.0, [...], 80.0, 80.0 # y coordinates
244         10.0, 10.0, 10.0, [...], 10.0, 10.0 # z coordinates

```

```

244     ~ ~ ~
245
246
247     Args:
248         dim (tuple): The dimensions of the field
249         freq (float): The discrete time frequency
250         cam_positions (list[tuple]): The positions of the camera
251         rope_lengths (list[tuple]): The lengths of the ropes
252
253     Returns:
254         str: The parsed output
255     """
256
257     Parser.logger.info(
258         "Parsing output with #cam_positions=%d and #rope_lengths=%d",
259         len(cam_positions),
260         len(rope_lengths),
261     )
262
263     # Using numpy to transform the lists into numpy arrays for easier handling
264     cam_positions = np.array(cam_positions)
265     rope_lengths = np.array(rope_lengths)
266
267     # transposing the arrays to get the correct shape
268     cam_positions = cam_positions.T
269     rope_lengths = rope_lengths.T
270
271     # creating the time stamps
272     time_stamps = np.arange(0, len(cam_positions[0])) / freq, 1 / freq
273
274     # creating the output strings
275     output1 = "\n".join(",".join(map(str, rope)) for rope in rope_lengths)
276
277     # adding the dimensions to the output
278     output2 = f"{dim[0]},{dim[1]},{dim[2]}\n"
279
280     # adding the time stamps to the output
281     output2 += ",".join(map(str, time_stamps)) + "\n"
282
283     # adding the positions of the camera to the output
284     output2 += ",".join(map(str, cam_positions[0])) + "\n"
285     output2 += ",".join(map(str, cam_positions[1])) + "\n"
286     output2 += ",".join(map(str, cam_positions[2])) + "\n"
287
288     # output2 = "\n".join(
289     #     ",".join(map(str, line)) for line in [dim, time_stamps, *cam_positions]
290     # )
291
292     Parser.logger.debug("Parsed output")
293
294     return output1, output2

```

```
295
296
297 class Instruction:
298     """ A class for defining an instruction"""
299
300     def __init__(self, start_time: float = 0.0, destination: tuple = (0, 0, 0)) ->
301         None:
302             """Initializes the Instruction class
303
304             Args:
305                 start_time (float, optional): The time to start the instruction. Defaults
306                 to 0.0.
307                 destination (tuple, optional): The destination of the instruction.
308                 Defaults to (0, 0, 0).
309
310             Returns:
311                 None
312             """
313
314         self.logger = logging.getLogger(__name__)
315         self.logger.debug(
316             "Initializing Instruction with start_time=%f and destination=%s",
317             start_time,
318             destination,
319         )
320         self.start_time = start_time
321         self.destination = destination
322
323     @classmethod
324     def parse(cls, data: str) -> "Instruction":
325         """Parses an instruction from a string
326
327             Args:
328                 data (str): The string to parse the instruction from
329                     - format: start_time x y z
330
331             Returns:
332                 Instruction: The instruction
333             """
334
335         cls.logger = logging.getLogger(__name__)
336         cls.logger.debug("Parsing instruction from %s", data)
337
338         try:
339             start_time, x, y, z = data.split(" ")
340         except ValueError as exp:
341             raise ValueError(
342                 f"Invalid instruction format (expected: start_time x y z), got: {data}"
343             )
344
345             ) from exp
346
347
348         return cls(float(start_time), (float(x), float(y), float(z)))
349
350
```

```

342     def __repr__(self) -> str:
343         """Returns the representation of the instruction
344
345         Args:
346             None
347
348         Returns:
349             str: The representation of the instruction
350             """
351
352         return (
353             f"Instruction(start_time={self.start_time}, destination={self.destination
354             })"
355         )

```

Listing 21: Quellcode für /python/spidercam_simulator/io.py

```

1  from __future__ import annotations
2
3  import logging
4
5  import numpy as np
6  import spidercam_simulator
7
8
9  class Movement:
10     """ A class for defining a movement"""
11
12     def __init__(
13         self,
14         spidercam: spidercam_simulator.Spidercam,
15         start_time: float = 0,
16         start: tuple = (0, 0, 0),
17         destination: tuple = (0, 0, 0),
18     ) -> None:
19         """ Initializes the Movement class
20
21         Args:
22             spidercam (Spidercam): The spidercam instance
23             start_time (float, optional): The time to start the movement. Defaults to
24             0.
25             start (tuple, optional): The starting position. Defaults to (0, 0, 0).
26             destination (tuple, optional): The destination position. Defaults to (0,
27             0, 0).
28
29         Returns:
30             None
31             """
32
33         self.logger = logging.getLogger(__name__)
34         self.logger.info(
35             "Initializing Movement at %s with start %s and destination %s",
36             start_time,
37             start,
38             destination,

```

```

35
)
36
37     self.spidercam = spidercam
38     self.start_time = start_time
39
40     self.phases = []
41
42     self.calculate_phases(start, destination)
43
44     def start(self) -> tuple:
45         """ Returns the start of the movement
46
47         Returns:
48             tuple: The start of the movement
49             """
50
51         return self.phases[0].start
52
53     def destination(self) -> tuple:
54         """ Returns the destination of the movement
55
56         Returns:
57             tuple: The destination of the movement
58             """
59
60         return self.phases[-1].destination
61
62     def duration(self) -> float:
63         """ Returns the duration of the movement
64
65         Returns:
66             float: The duration of the movement
67             """
68
69         return sum(phase.duration for phase in self.phases)
70
71     def distance(self) -> float:
72         """ Returns the distance of the movement
73
74         Returns:
75             float: The distance of the movement
76             """
77
78         return sum(phase.distance for phase in self.phases)
79
80     def end_time(self) -> float:
81         """ Returns the end time of the movement
82
83         Returns:
84             float: The end time of the movement
85             """
86
87         return self.start_time + self.duration()
88
89     def calculate_phases(self, start: tuple, destination: tuple) -> None:
90         """ Calculates the phases of the movement

```

```

86
87     Args:
88         start (tuple): The starting position
89         destination (tuple): The destination position
90
91     Returns:
92         None
93     """
94
95     # HERE 3
96     self.logger.info("Calculating phases")
97
98     distance = np.linalg.norm(np.array(destination) - np.array(start))
99
100    # Check how many phases are needed
101    if distance <= 2 * self.spidercam.dist_vmax:
102        # Only acceleration and deceleration
103        self.logger.debug("Only acceleration and deceleration needed")
104
105        # Calculate middle point
106        middle_point = start + (np.array(destination) - np.array(start)) / 2
107        self.phases = [
108            spidercam_simulator.Phase(
109                self,
110                spidercam_simulator.Phase.Mode.ACCELERATION,
111                start,
112                middle_point,
113                0.0,
114            ),
115            spidercam_simulator.Phase(
116                self,
117                spidercam_simulator.Phase.Mode.DECCELERATION,
118                middle_point,
119                destination,
120                np.sqrt(self.spidercam.acceleration * distance),
121            ),
122        ]
123
124    else:
125        # Acceleration, constant velocity and deceleration
126        self.logger.debug("Acceleration, constant velocity and deceleration needed")
127
128        # Calculate needed points
129        point_a = (
130            np.array(start)
131            + (np.array(destination) - np.array(start))
132            * self.spidercam.dist_vmax
133            / distance
134        )
135
136        point_b = (

```

```

136         np.array(destination)
137         - (np.array(destination) - np.array(start))
138         * self.spidercam.dist_vmax
139         / distance
140     )
141
142     self.phases = [
143         spidercam_simulator.Phase(
144             self,
145             spidercam_simulator.Phase.Mode.ACCELERATION,
146             start,
147             point_a,
148             0.0,
149         ),
150         spidercam_simulator.Phase(
151             self,
152             spidercam_simulator.Phase.Mode.CONSTANT_VELOCITY,
153             point_a,
154             point_b,
155             self.spidercam.max_velocity,
156         ),
157         spidercam_simulator.Phase(
158             self,
159             spidercam_simulator.Phase.Mode.DECCELERATION,
160             point_b,
161             destination,
162             self.spidercam.max_velocity,
163         ),
164     ]
165
166     def start_deceleration(self, time: float) -> None:
167         """ Starts deceleration
168
169         Args:
170             time (float): The time to start deceleration
171
172         Returns:
173             None
174         """
175
176         self.logger.info("Starting deceleration at %s", time)
177
178         # Getting the phase that is active at the time
179         # and the time offset of the phase
180         # update all phases after the phase
181
182         time_sum = self.start_time
183
184         for phase in self.phases:
185             # Skip phases before the time
186             if time_sum + phase.duration < time:

```

```

187         time_sum += phase.duration
188         continue
189
190     # Nothing to do if already decelerating
191     if phase.mode == spidercam_simulator.Phase.Mode.DECCELERATION:
192         self.logger.debug("Already decelerating, nothing to do")
193         return
194
195     offset = time - time_sum
196
197     # Found phase should end at offset
198     phase.destination = phase.get_position(offset)
199     phase.update()
200
201     # Update next phases
202     # There always is a next phase because the final phase is deceleration and
203     # the previous phase was not
204     next_phase = self.phases[self.phases.index(phase) + 1]
205
206     next_phase.start = phase.destination
207     next_phase.starting_velocity = (
208         self.spidercam.max_velocity
209         if phase.mode is spidercam_simulator.Phase.Mode.CONSTANT_VELOCITY
210         else self.spidercam.acceleration * offset
211     )
212     next_phase.mode = spidercam_simulator.Phase.Mode.DECCELERATION
213     next_phase.destination = next_phase.get_position(
214         next_phase.starting_velocity / self.spidercam.acceleration
215     )
216     next_phase.update()
217
218     # If there is a phase after the next phase, pop it
219     if len(self.phases) > self.phases.index(next_phase) + 1:
220         self.phases.pop(self.phases.index(next_phase) + 1)
221
222     break
223
224 def get_phase(self, time: float) -> spidercam_simulator.Phase:
225     """ Returns the phase at the given time
226
227     Args:
228         time (float): The time to get the phase at
229
230     Returns:
231         Phase: The phase at the given time
232
233     time_sum = self.start_time
234
235     for phase in self.phases:
236         if time_sum + phase.duration > time:
237             return phase

```

```

237         time_sum += phase.duration
238
239     return self.phases[-1]
240
241 def __repr__(self):
242     return f"Movement(start={repr(self.start())}, destination={repr(self.destination())}, start_time={self.start_time}, duration={self.duration()}, distance={self.distance()})"
243
244 def get_position(self, time: float) -> tuple:
245     """ Returns the position of the movement at the time
246
247     Args:
248         time (float): The time to get the position for
249
250     Returns:
251         tuple: The position of the movement at the time
252     """
253
254     self.logger.info("Getting position for time %s", time)
255
256     # Getting the phase that is active at the time
257     # and the time offset of the phase
258     time_sum = self.start_time
259
260     for phase in self.phases:
261         if time_sum + phase.duration >= time:
262             return phase.get_position(time - time_sum)
263
264         time_sum += phase.duration
265
266     # Just for debugging
267     raise Exception("No phase found")

```

Listing 22: Quellcode für /python/spidercam_simulator/movement.py

```

1 from __future__ import annotations
2
3 import logging
4 from enum import Enum
5
6 import numpy as np
7 import spidercam_simulator
8
9
10 class Phase:
11     """ A class for defining a phase"""
12
13     Mode = Enum("Mode", ["ACCELERATION", "CONSTANT_VELOCITY", "DECELERATION"])
14
15     def __init__(
16         self,

```

```

17     movement: spidercam_simulator.Movement,
18     mode: Mode,
19     start: tuple = (0, 0, 0),
20     destination: tuple = (0, 0, 0),
21     starting_velocity: float = 0,
22 ) -> None:
23     """ Initializes the Phase class
24
25     Args:
26         movement (Movement): The movement instance
27         mode (Mode): The type of phase. Can be ACCELERATION, CONSTANT_VELOCITY, or
28         DECELERATION
29         start (tuple, optional): The starting position. Defaults to (0, 0, 0).
30         destination (tuple, optional): The destination position. Defaults to (0,
31         0, 0).
32         starting_velocity (float, optional): The starting velocity. Defaults to 0.
33
34     Returns:
35         None
36     """
37     self.logger = logging.getLogger(__name__)
38     self.logger.info(
39         "Initializing Phase with mode %s, start %s, destination %s, and starting
40         velocity %s",
41         mode,
42         start,
43         destination,
44         starting_velocity,
45     )
46
47     self.movement = movement
48     self.mode = mode
49     self.start = start
50     self.destination = destination
51     self.starting_velocity = starting_velocity
52
53     self.update()
54
55     def update(self) -> None:
56         """ Updates the phase
57
58         Args:
59             None
60
61         Returns:
62             None
63         """
64         self.logger.debug("Updating phase %s", self.mode)
65
66         self.distance = self.calc_distance()
67         self.duration = self.calc_duration()

```

```

65     self.logger.debug(
66         "Updated phase with distance %s and duration %s",
67         self.distance,
68         self.duration,
69     )
70
71
72     def calc_distance(self) -> float:
73         """ Returns the distance of the phase
74
75         Returns:
76             float: The distance of the phase
77         """
78
79         return np.linalg.norm(np.array(self.destination) - np.array(self.start))
80
81
82     def calc_duration(self) -> float:
83         """ Returns the duration of the phase
84
85         Returns:
86             float: The duration of the phase
87         """
88
89
90         if self.mode == Phase.Mode.ACCELERATION:
91             # t = sqrt(2 * d / a)
92             return np.sqrt((2 * self.distance) / self.movement.spidercam.acceleration)
93         elif self.mode == Phase.Mode.CONSTANT_VELOCITY:
94             # t = d / v
95             return self.distance / self.starting_velocity
96         elif self.mode == Phase.Mode.DECCELERATION:
97             # t = v0 / a, because final velocity is 0
98             return self.starting_velocity / self.movement.spidercam.acceleration
99
100
101    def get_position(self, offset: float) -> tuple:
102        """ Returns the position of the phase after a given offset
103
104        Args:
105            offset (float): The offset to get the position after
106
107        Returns:
108            tuple: The position of the phase after the given offset
109        """
110
111
112        self.logger.debug(
113            "Getting position after %s seconds, mode %s", offset, self.mode
114        )
115
116
117        # if offset > self.duration:
118        #     self.logger.debug(
119        #         f"Offset {offset} is greater than duration {self.duration},"
120        returning destination {repr(self.destination)}"
121        #     )

```

```

115     #     return self.destination
116
117     if self.mode == Phase.Mode.ACCELERATION:
118         distance = self.movement.spidercam.acceleration * offset**2 / 2
119     elif self.mode == Phase.Mode.CONSTANT_VELOCITY:
120         distance = self.starting_velocity * offset
121     elif self.mode == Phase.Mode.DECCELERATION:
122         distance = (
123             self.starting_velocity * offset
124             - self.movement.spidercam.acceleration * offset**2 / 2
125         )
126
127     self.logger.debug("Phase distance: %s", distance)
128
129     # if distance almost 0, return start
130     if np.isclose(distance, 0):
131         self.logger.debug(
132             "Distance is almost 0, returning start %s", repr(self.start)
133         )
134         return self.start
135
136     position = np.array(self.start) + (
137         np.array(self.destination) - np.array(self.start)
138     ) * distance / np.linalg.norm(np.array(self.destination) - np.array(self.start)
139 ))
140
141     self.logger.debug("Phase position: %s", position)
142
143     return tuple(position)

```

Listing 23: Quellcode für /python/spidercam_simulator/phase.py

```

1 from __future__ import annotations
2 import logging
3 import os
4 import matplotlib.pyplot as plt
5 import numpy as np
6
7
8 class Plotter:
9     """ A class for plotting camera positions and rope lengths"""
10
11     def __init__(
12         self,
13         dim: tuple = None,
14         freq: float = 1.0,
15         cam_positions: list = None,
16         rope_lengths: list = None,
17         output_dir: str = None,
18         name: str = None,
19     ) -> None:
20         """ Initializes the Plotter class

```

```

21
22     Args:
23         dim (tuple, optional): The dimensions of the field. Defaults to None.
24         freq (float, optional): The discrete time frequency. Defaults to 1.0.
25         cam_positions (list, optional): The camera positions. Defaults to None.
26         rope_lengths (list, optional): The rope lengths. Defaults to None.
27         output_dir (str, optional): The output directory. Defaults to None.
28         name (str, optional): The name of the plot. Defaults to None.
29
30     Returns:
31         None
32     """
33
34     self.logger = logging.getLogger(__name__)
35     # Setting log level info to suppress matplotlib font manager warnings
36     self.logger.setLevel(logging.INFO)
37
38     self.logger.info("Initializing Plotter for %s", name)
39
40     self.dim = dim
41     self.freq = freq
42     self.cam_positions = np.array(cam_positions)
43     self.rope_lengths = np.array(rope_lengths)
44     self.output_dir = output_dir
45     self.name = name
46
47     def plot_cam_positions(self) -> None:
48         """ Plots the camera positions
49
50         Returns:
51             None
52         """
53
54         self.logger.info("Plotting camera positions for %s", self.name)
55
56         # If output directory is not specified, plot to screen
57         if self.output_dir is None:
58             plt.io()
59
60         # Plot camera positions projection = 3d
61         fig = plt.figure()
62         ax = fig.add_subplot(111, projection="3d")
63
64         # Set axis labels
65         ax.set_xlabel("X")
66         ax.set_ylabel("Y")
67         ax.set_zlabel("Z")
68
69         # Set axis limits
70         ax.set_xlim3d(0, self.dim[0])
71         ax.set_ylim3d(0, self.dim[1])

```

```

72     ax.set_zlim3d(0, self.dim[2])
73
74     # Rotate so that origin is in the bottom left and z is up, angle is normal
75     # ax.view_init(azim=0, elev=90)
76
77     # Plot camera positions
78     xline = self.cam_positions[:, 0]
79     yline = self.cam_positions[:, 1]
80     zline = self.cam_positions[:, 2]
81
82     # Define color as distance from one point to the previous
83     color = np.zeros(len(xline))
84     for i in range(1, len(xline)):
85         color[i] = np.linalg.norm(
86             np.array([xline[i], yline[i], zline[i]]) -
87             np.array([xline[i - 1], yline[i - 1], zline[i - 1]]))
88     )
89
90     # Plot camera positions
91     ax.scatter(xline, yline, zline, c=color, cmap="coolwarm")
92
93     # Plot start as big green x
94     ax.scatter(xline[0], yline[0], zline[0], c="green", s=200, marker="x")
95
96     # Set title for whole figure
97     # fig.suptitle("Camera Positions for " + self.name)
98
99     # Save or show plot
100    if self.output_dir is None:
101        plt.show()
102    else:
103        # Save at output + name + cam_pos.png
104        # only get base name of file
105        plt.savefig(
106            f"{self.output_dir}/{os.path.basename(self.name)}_cam_pos.png",
107            bbox_inches="tight",
108            dpi=300,
109        )
110
111    def plot_rope_lengths(self) -> None:
112        """ Plots the rope lengths
113
114        Returns:
115            None
116        """
117
118        self.logger.info("Plotting rope lengths for %s", self.name)
119
120        # If output directory is not specified, plot to screen
121        if self.output_dir is None:
122            plt.io()
```

```

123
124     # Plot rope lengths as a function of time (discrete)
125     fig = plt.figure()
126
127     # Set axis labels
128     plt.xlabel("Time (s)")
129     plt.ylabel("Rope Length (m)")
130
131     # Set axis limits
132     plt.xlim(0, len(self.rope_lengths) / self.freq)
133     plt.ylim(0, np.max(self.rope_lengths) * 1.1)
134
135     # Plot rope lengths
136     plt.plot(
137         np.arange(0, len(self.rope_lengths) / self.freq, 1 / self.freq),
138         self.rope_lengths,
139     )
140
141     # Legend of Rope i
142     plt.legend([f"Rope {i}" for i in range(len(self.rope_lengths[0]))])
143
144     # Set title for whole figure
145     # fig.suptitle("Rope Lengths for " + self.name)
146
147     # Save or show plot
148     if self.output_dir is None:
149         plt.show()
150     else:
151         # Save at output + name + rope_lengths.png
152         plt.savefig(
153             f"{self.output_dir}/{os.path.basename(self.name)}_rope_lengths.png",
154             bbox_inches="tight",
155             dpi=300,
156         )
157
158     def plot(self) -> None:
159         """ Plots the camera positions and rope lengths
160
161         Returns:
162             None
163         """
164
165         self.logger.info("Plotting camera positions and rope lengths for %s", self.
166         name)
167
168         self.plot_cam_positions()
169         self.plot_rope_lengths()

```

Listing 24: Quellcode für /python/spidercam_simulator/plotter.py

```

1 from __future__ import annotations
2

```

```

3 import logging
4
5 import spidercam_simulator
6
7
8 class Spidercam:
9     """ A class for controlling the spidercam"""
10
11     movements = []
12     queue = None
13
14     def __init__(
15         self,
16         controller: spidercam_simulator.Controller,
17         max_velocity: float = 1.0,
18         acceleration: float = 1.0,
19         start: tuple = (0, 0, 0),
20     ) -> None:
21         """ Initializes the Spidercam class
22
23         Args:
24             controller (Controller): The controller instance
25             max_velocity (float, optional): The maximum velocity of the spidercam.
26             Defaults to 1.0.
27             acceleration (float, optional): The maximum acceleration of the spidercam.
28             Defaults to 1.0.
29             start (tuple, optional): The starting position of the spidercam. Defaults
27             to (0, 0, 0).
30
31         Returns:
32             None
33
34         """
35
36         self.logger = logging.getLogger(__name__)
37         self.logger.debug(
38             "Initializing Spidercam with max_velocity=%s, acceleration=%s, start=%s",
39             max_velocity,
40             acceleration,
41             start,
42         )
43
44         self.controller = controller
45         self.max_velocity = max_velocity
46         self.acceleration = acceleration
47         self.start = start
48
49         self.movements = []
50         self.queue = None
51
52         self.calc_constants()

```

```

51     def calc_constants(self) -> None:
52         """ Calculates the following constants for the spidercam:
53
54         - time_vmax: The time it takes to reach the maximum velocity
55         
$$t_{\max} = \frac{v_{\max}}{a_{\max}}$$

56
57         - dist_vmax: The distance it takes to reach the maximum velocity
58         
$$d_{\max} = \frac{v_{\max}^2}{2a_{\max}}$$

59
60     Returns:
61         None
62         """
63
64     self.logger.debug("Calculating constraints for spidercam")
65
66     self.time_vmax = self.max_velocity / self.acceleration
67     self.dist_vmax = self.acceleration * self.time_vmax**2 / 2
68
69     def move(
70         self, instruction: spidercam_simulator.Instruction = None, time: float = -1.0
71     ) -> None:
72         """ Moves the spidercam to a given position or updates the current movement at
73         a given time
74
75         Args:
76             Instruction (Instruction): The instruction to move the spidercam. Defaults
77             to None.
78             time (float): The time to update the current movement at. Defaults to
79             -1.0.
80
81         Returns:
82             None
83             """
84
85         # Check if parameters are valid
86         if instruction is None and time == -1.0:
87             self.logger.debug("No instruction or time given, returning")
88             return
89
90         # No instruction given, update the current movement
91         if instruction is None and time != -1.0:
92             self.logger.debug("Updating movement at time %s", time)
93
94         # REDUNDANT?
95         # If there is no movement in the queue, return
96         if self.queue is None:
97             self.logger.debug("No movement in queue, returning")
98             return
99
100
101         # If the last movement is finished, add the movement in the queue to the
102         movements list

```

```

98     if self.movements[-1].end_time() <= time:
99         self.logger.debug(
100            "Last movement is finished, adding queue to movements list"
101        )
102         self.movements.append(self.queue)
103         self.queue = None
104         return
105
106     return
107
108 # Instruction given, apply the instruction
109 self.logger.debug("Moving spidercam with instruction %s", instruction)
110
111 # First movement is always possible
112 if len(self.movements) == 0:
113     self.logger.debug("First movement registered")
114
115     self.movements.append(
116         spidercam_simulator.Movement(
117             self, instruction.start_time, self.start, instruction.destination
118         )
119     )
120     return
121
122 self.logger.debug("Checking if last movement is finished")
123 self.logger.debug("Last movement: %s", self.movements[-1])
124
125 # If the last movement is not finished, decelerate and update the queue
126 if self.movements[-1].end_time() > instruction.start_time:
127     self.logger.debug("Last movement is not yet finished")
128
129     self.logger.debug("Queueing instruction %s", instruction)
130
131     # Decelerate
132     self.movements[-1].start_deceleration(instruction.start_time)
133
134     # Overwrite queue
135     self.queue = spidercam_simulator.Movement(
136         self,
137         self.movements[-1].end_time(),
138         self.movements[-1].destination(),
139         instruction.destination,
140     )
141
142     return
143
144 # REDUNDANT ?
145 # If the last movement is finished and there is a queue, move to the queue
146 if self.queue is not None:
147     self.logger.debug("Last movement is finished and there is a queue")
148     self.logger.debug("Executing queue")

```

```

149
150     self.movements.append(self.queue)
151     self.queue = None
152
153     # Move to the destination after the queue
154     self.move(instruction)
155     return
156
157     # If the last movement is finished and there is no queue, move to the
158     # destination
159     self.logger.debug("Last movement is finished and there is no queue")
160
161     self.movements.append(
162         spidercam_simulator.Movement(
163             self,
164             max(self.movements[-1].end_time(), instruction.start_time),
165             self.movements[-1].destination(),
166             instruction.destination,
167         )
168     )
169
170     def get_position(self, time: float) -> tuple:
171         """ Gets the position of the spidercam at a given time
172
173         Args:
174             time (float): The time to get the position at
175
176         Returns:
177             tuple: The position of the spidercam at the given time
178
179         """
180
181         self.logger.debug("Getting position at time %s", time)
182
183         self.move(time=time)
184
185         # If there are no movements, return the start position
186         if len(self.movements) == 0:
187             self.logger.debug("No movements registered, returning start position")
188             return self.start
189
190         # If the time is before the first movement, return the start position
191         if time < self.movements[0].start_time:
192             self.logger.debug("Time is before first movement, returning start position")
193         return self.start
194
195         # If the time is after the last movement, return the destination of the last
196         # movement
197         if time > self.movements[-1].end_time():
198             self.logger.debug(
199                 "Time is after last movement, returning destination of last movement"

```

```

197     )
198     return self.movements[-1].destination()
199
200     # If the time is during a movement, return the position of that movement
201     for movement in self.movements:
202         if movement.start_time <= time <= movement.end_time():
203             self.logger.debug(
204                 "Time is during movement %s, returning position of movement",
205                 movement,
206             )
207             return movement.get_position(time)
208
209     # If the time is between movements, return the destination of the previous
210     # movement
211     for i in range(len(self.movements) - 1):
212         if self.movements[i].end_time() < time < self.movements[i + 1].start_time:
213             self.logger.debug(
214                 "Time is between movements %s and %s, returning destination of
215                 previous movement",
216                 self.movements[i],
217                 self.movements[i + 1],
218             )
219             return self.movements[i].destination()
220
221     # If the time is not in any of the above cases, return the start position
222     self.logger.debug(
223         "Time is not in any of the above cases, returning start position"
224     )

```

Listing 25: Quellcode für /python/spidercam_simulator/spidercam.py