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

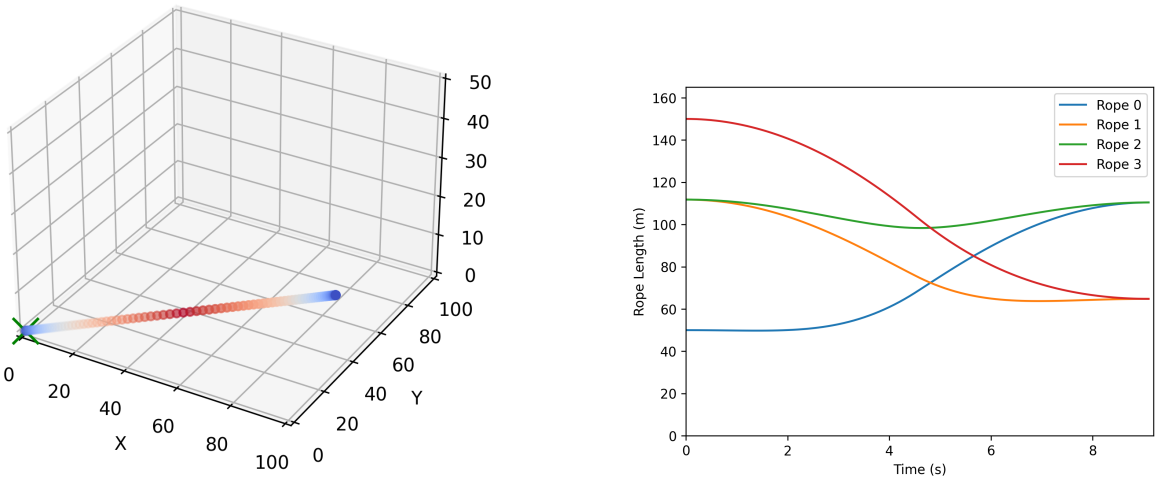
1 Aufgabenanalyse

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 muss[1],
- 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.

[1] Die 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.

```
# Beispiel
dim 70 100 30
start 10 80 10
vmax 6
amax 2
freq 2
0 50 40 30 # Instruktion 1
20 10 80 10 # Instruktion 2
22 50 40 30     # Instruktion 3
23 35 50 30     # Instruktion 4
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 eingelesen[2]:

- 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 Spielfelds[3] oder nicht vorhanden

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

[2] Von einer *syntaktisch* korrekten Eingabedatei kann ausgegangen werden.
[3] Von 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 [Hz $= s^{-1}$] 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.

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

Abbildung 3: Beispiel für eine Ausgabedatei 1

```
70,100,30
0.0,0.5,1.0,[...],12.5,13.0,13.5
10.0,10.166666666666666,10.666666666666666,[...],49.833333333333336,50.0
80.0,79.83333333333333,79.33333333333333,[...],40.166666666666664,40.0
10.0,10.083333333333334,10.333333333333334,[...],29.916666666666668,30.0
```

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

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

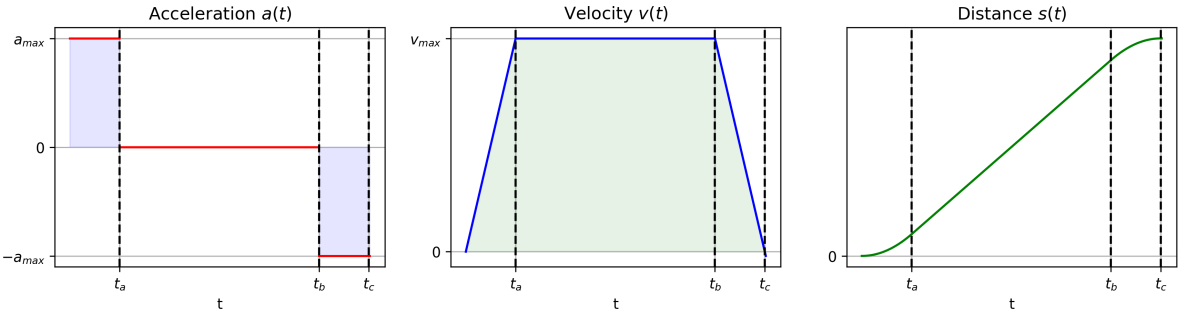


Abbildung 5: Bewegung der Spidercam in drei Phasen. t_a entspricht dem Zeitraum, in dem die Spidercam maximal beschleunigt wird. t_b ist der Zeitpunkt, an dem abgebremst wird und t_c ist der Zeitpunkt, an dem die Spidercam ihren Zielort erreicht hat.

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):

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

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

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

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

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

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.

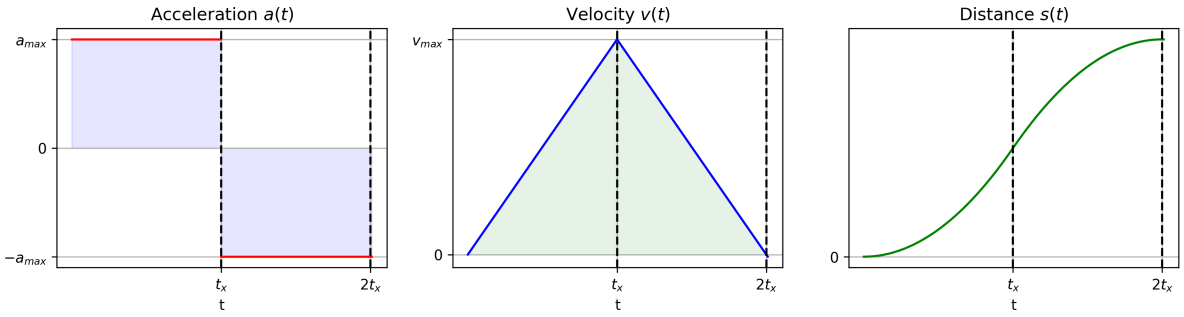


Abbildung 6: Bewegung der Spidercam in zwei Phasen. t_x ist der Zeitpunkt, an dem abgebremst wird und $2t_x$ ist der Zeitpunkt, an dem die Spidercam ihren Zielort erreicht hat.

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 bremsen[4]. 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)

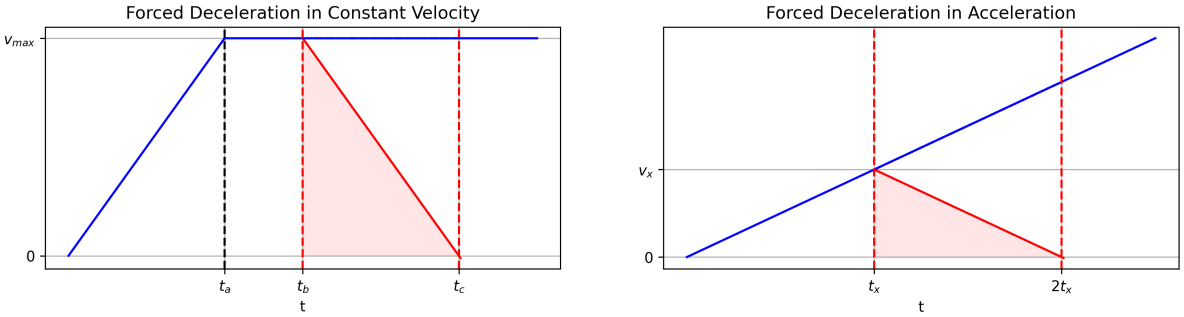


Abbildung 7: Berechnung der Bremsdistanz d_{DEC}. Die rote Fläche unter der Geschwindigkeitskurve ist die Bremsdistanz.

[4]Wenn 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 genannt 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`).

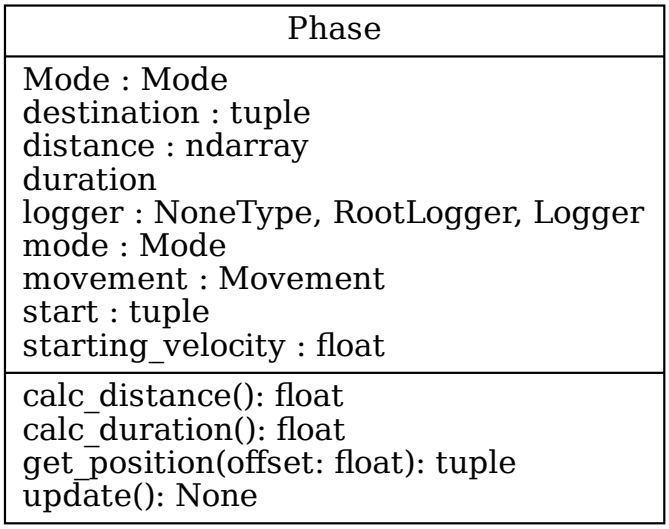


Abbildung 8: Klassendiagramm der Klasse `Phase`

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.

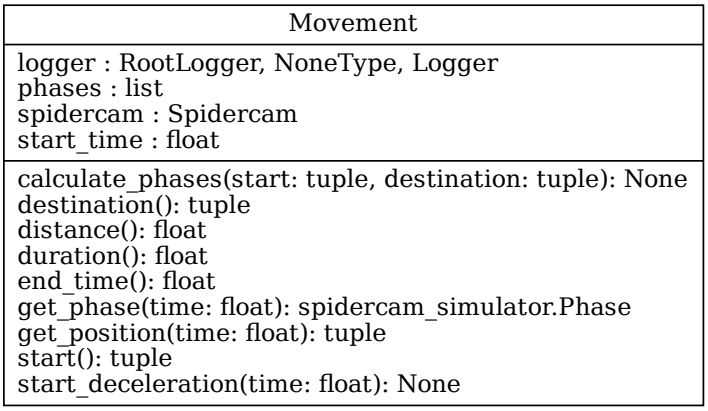


Abbildung 9: Klassendiagramm der Klasse `Movement`

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`).

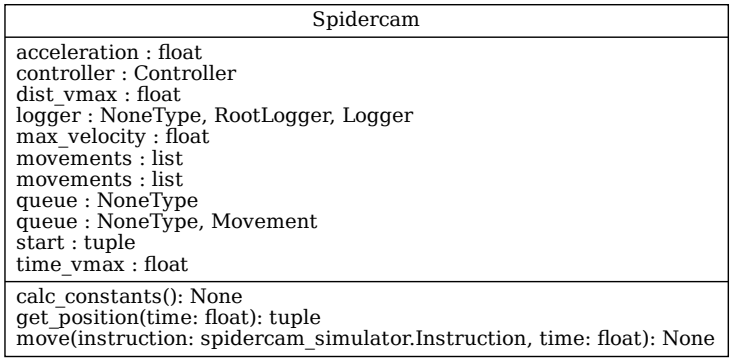


Abbildung 10: Klassendiagramm der Klasse `Spidercam`

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.

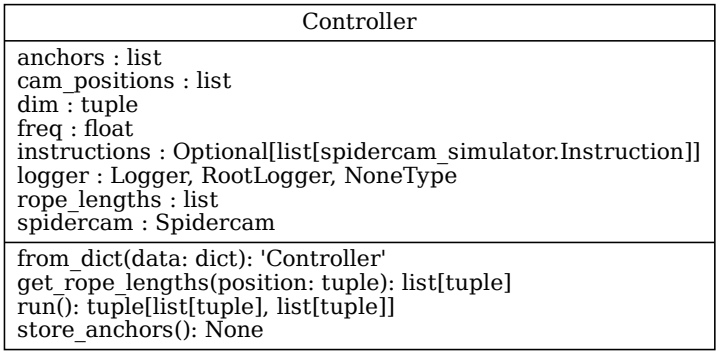


Abbildung 11: Klassendiagramm der Klasse `Controller`

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 instanziiert, indem ihr die Pfade zur Eingabe*datei* und zum Ausgabe*ordner* 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 instanziiert, 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.

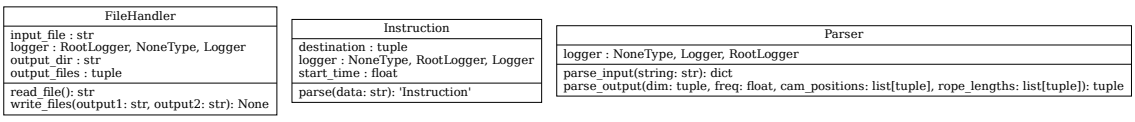


Abbildung 12: Klassendiagramme der Hilfsklassen `FileHandler`, `Instruction` und `Parser`

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 instanziiert, indem ihr analog zur Funktion `parse_output()` die entsprechenden Daten übergeben werden.

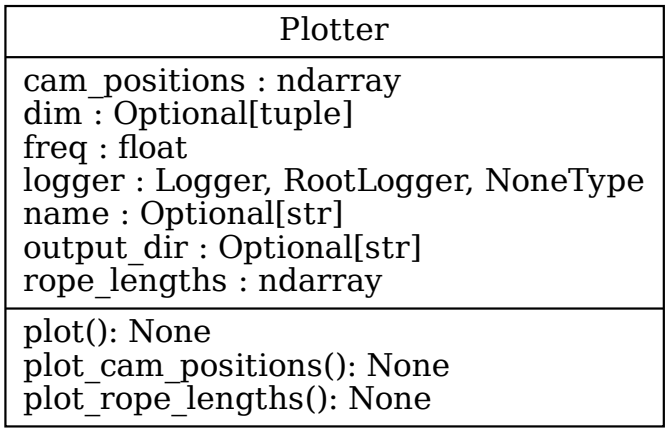


Abbildung 13: Klassendiagramm der Klasse `Plotter`

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.

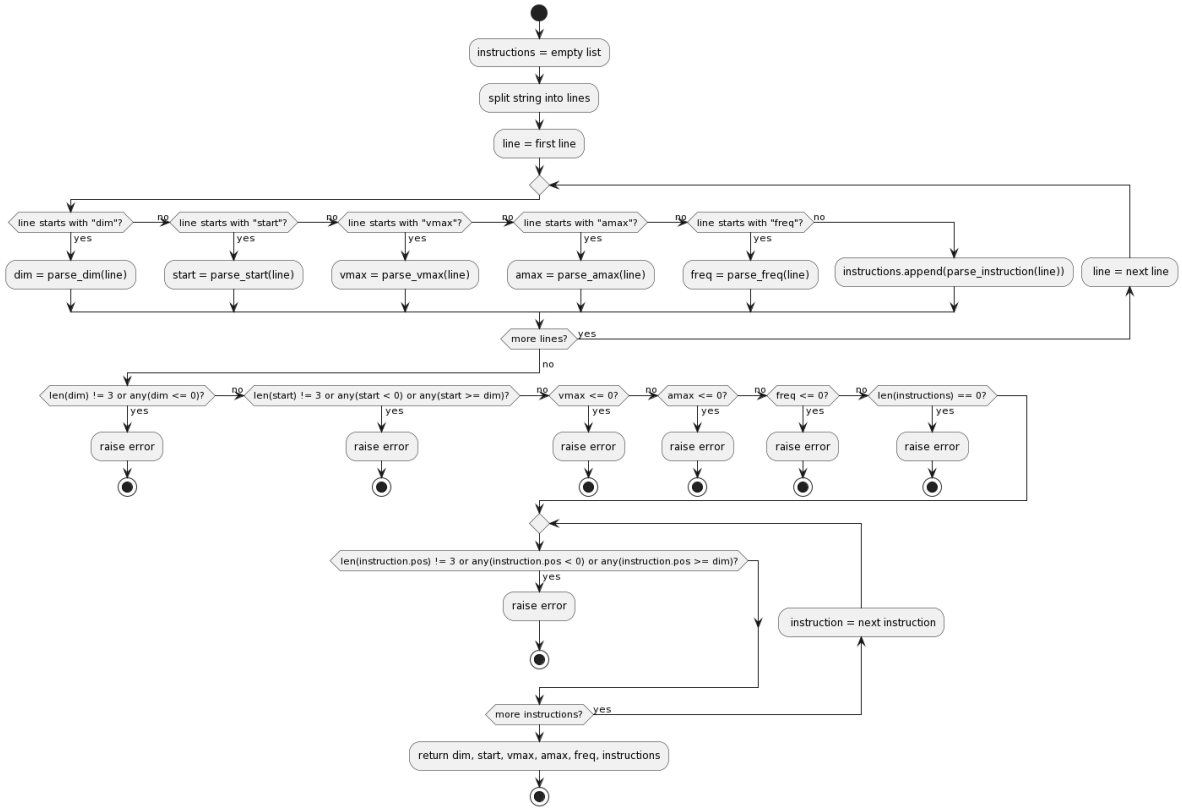


Abbildung 14: Aktivitätsdiagramm der Funktion `parse_input()` der Klasse `Parser`

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.

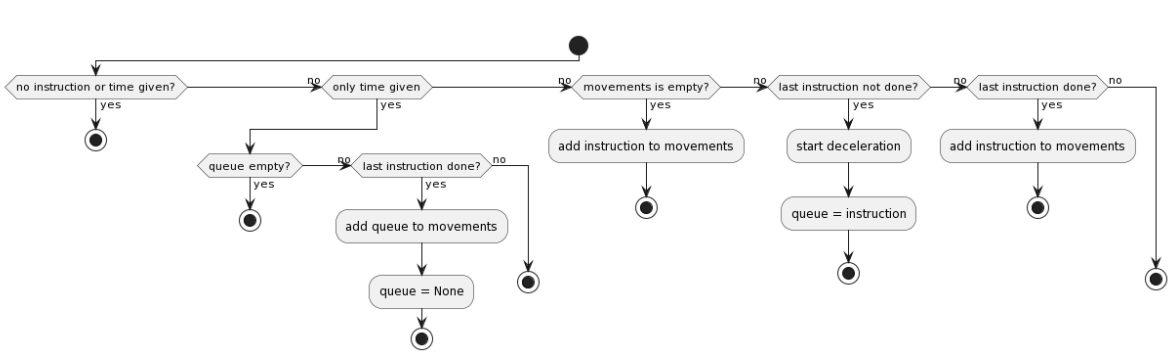


Abbildung 15: Aktivitätsdiagramm der Funktion `move()` der Klasse `Spidercam`

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.

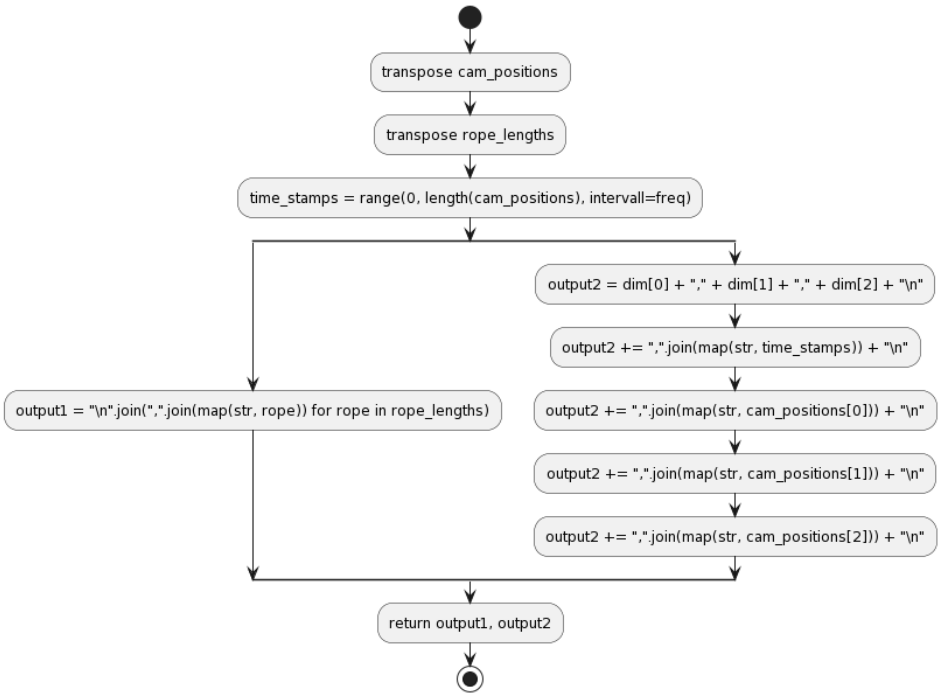


Abbildung 16: Aktivitätsdiagramm der Funktion `parse_output()`

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

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

```
# adding the dimensions to the output
output2 = f"{dim[0]},{dim[1]},{dim[2]}\n"
# adding the time stamps to the output
output2 += ",".join(map(str, time_stamps)) + "\n"
# adding the positions of the camera to the output
output2 += ",".join(map(str, cam_positions[0])) + "\n"
output2 += ",".join(map(str, cam_positions[1])) + "\n"
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 ließt eine Konfigurationsdatei `config.ini` und Argumente aus der Kommandozeile ein und überprüft diese auf Korrektheit.

Anschließend wird für alle definierten Eingabedateien ein `Controller`-Objekt instanziiert. 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.

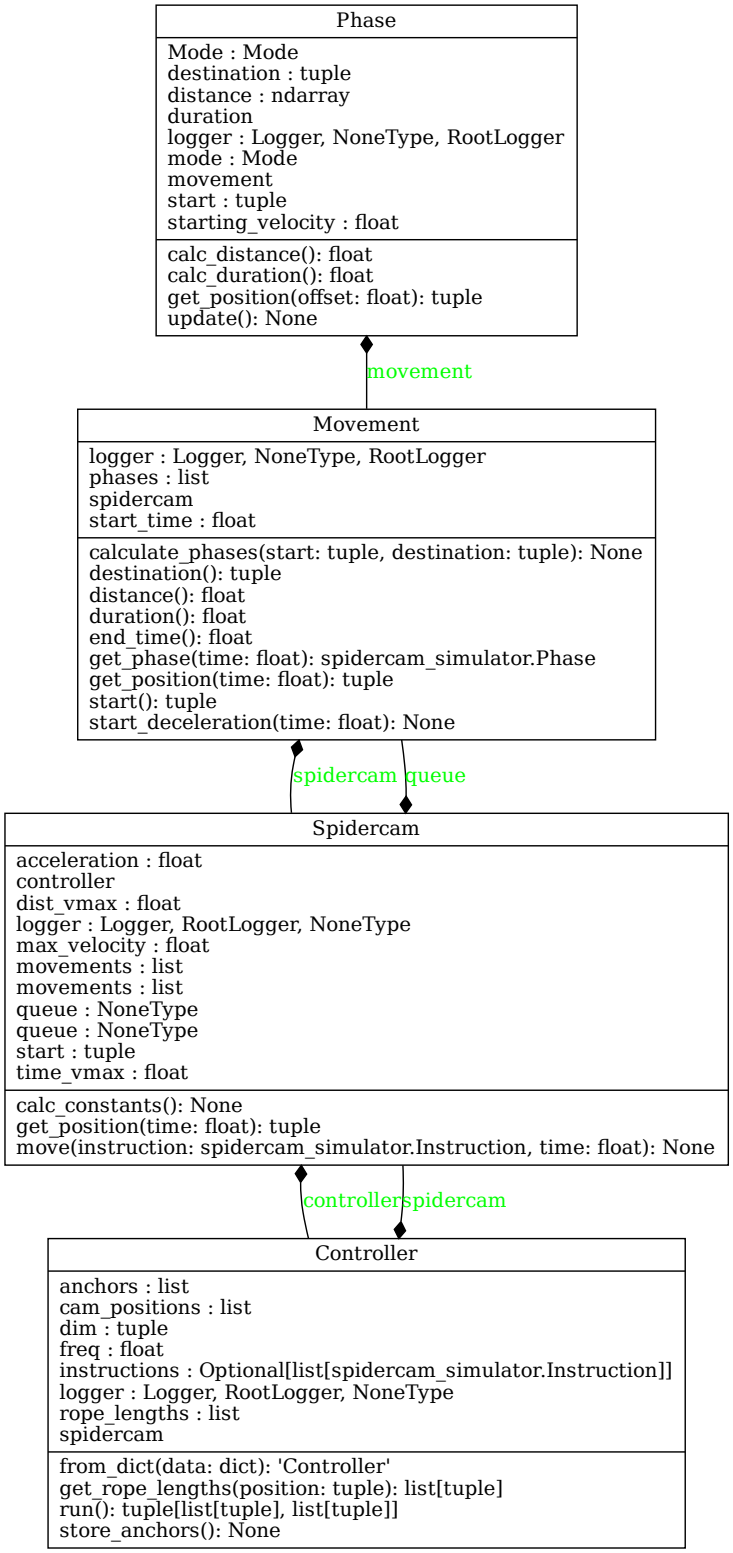


Abbildung 17: Klassendiagramm des Programms

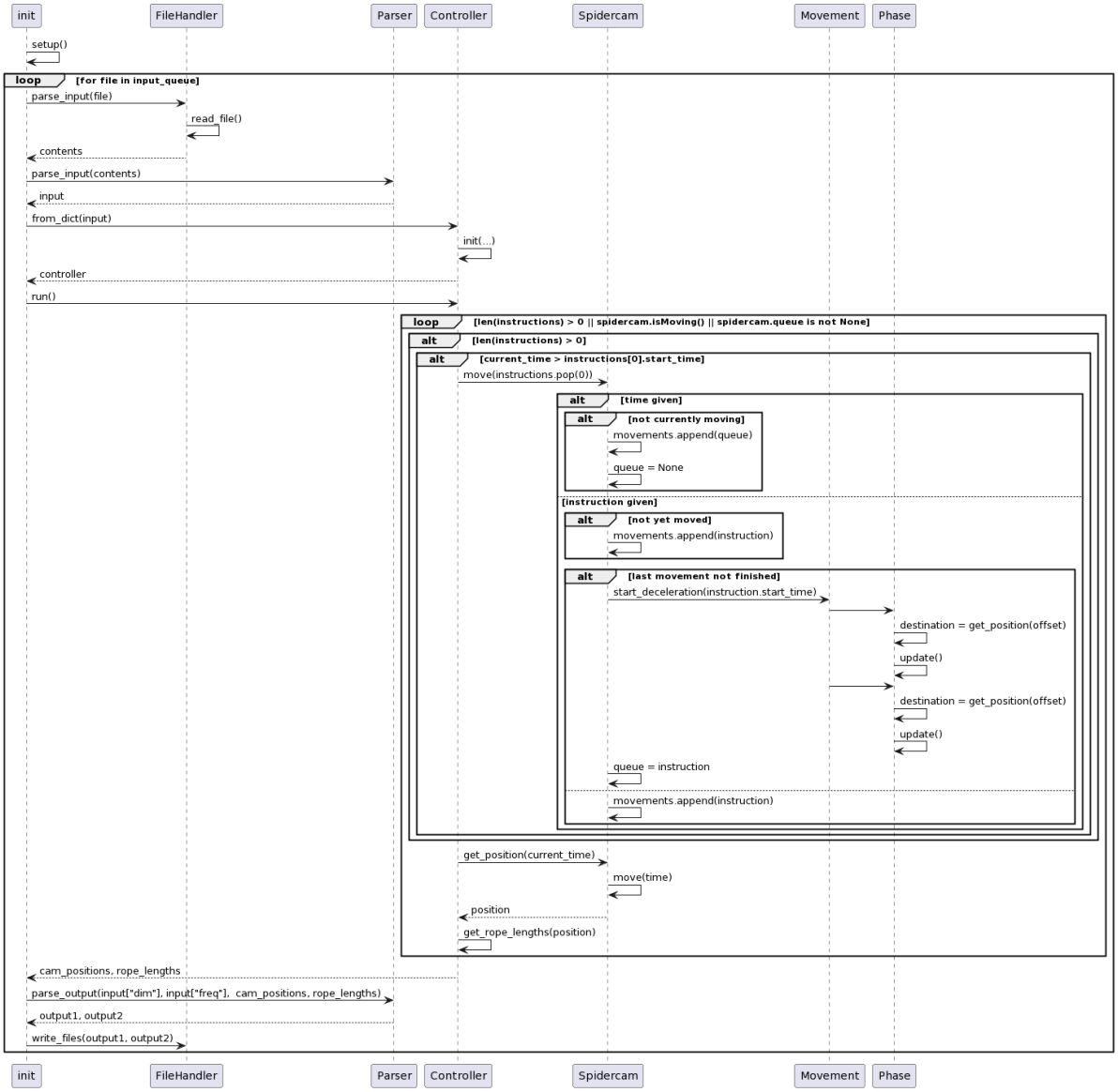


Abbildung 18: Aufrufhierarchie des Programms

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	

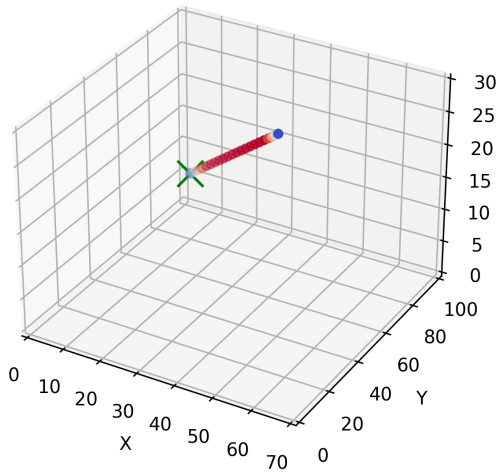
Abbildung 19: Testfälle

4.2 Beobachtungen

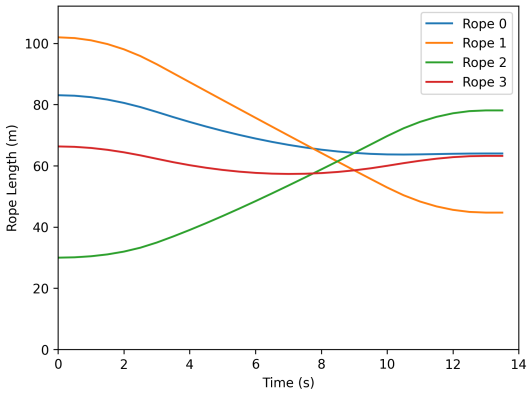
Für alle definierten Testfälle wurden die Ergebnisse mit den erwarteten Ergebnissen verglichen. Es konnte kein Fehlverhalten festgestellt werden. Das bedeutet:

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

Einige Ergebnisse sind in den folgenden Abbildungen dargestellt.

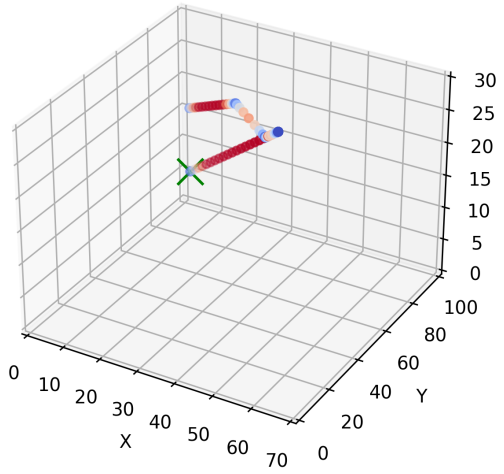


(a) Bewegung der Spidercam für Test `IHK_1`

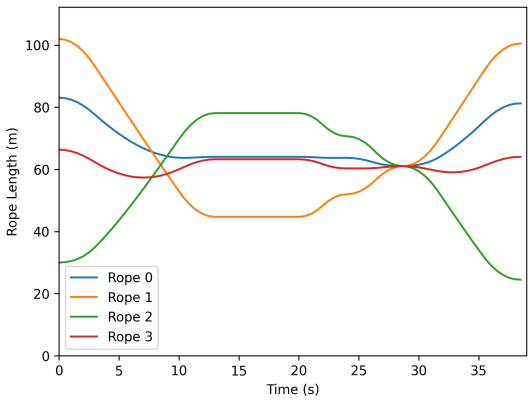


(b) Veränderung der Seillängen für Test `IHK_1`

Abbildung 20: Ergebnisse für Test `IHK_1`

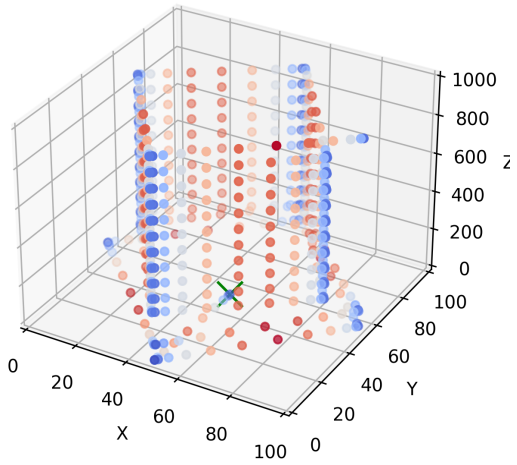


(a) Bewegung der Spidercam für Test `IHK_2`

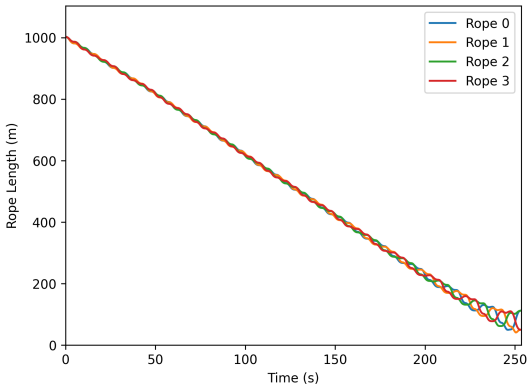


(b) Veränderung der Seillängen für Test `IHK_2`

Abbildung 21: Ergebnisse für Test `IHK_2`

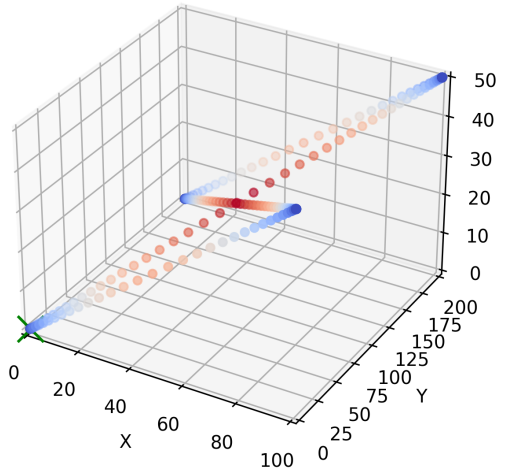


(a) Bewegung der Spidercam für Test `spiral`

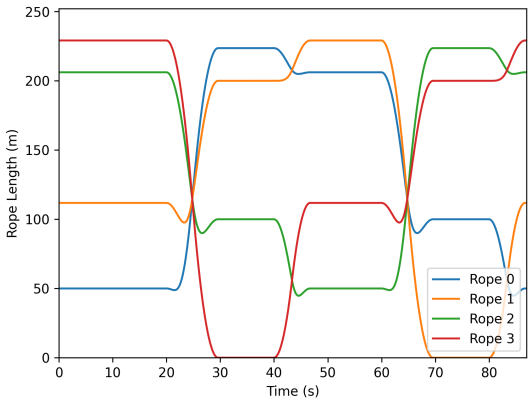


(b) Veränderung der Seillängen für Test `spiral`

Abbildung 22: Ergebnisse für Test `spiral`

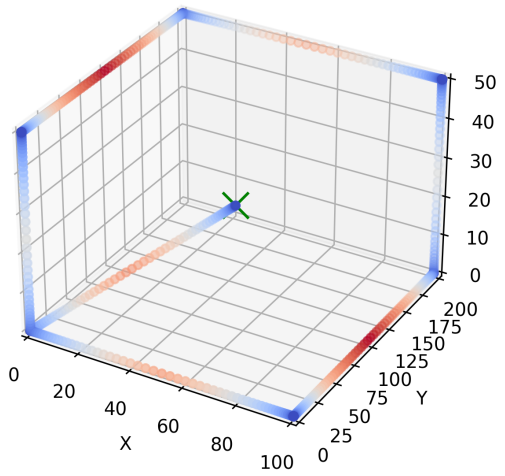


(a) Bewegung der Spidercam für Test `diagonals`

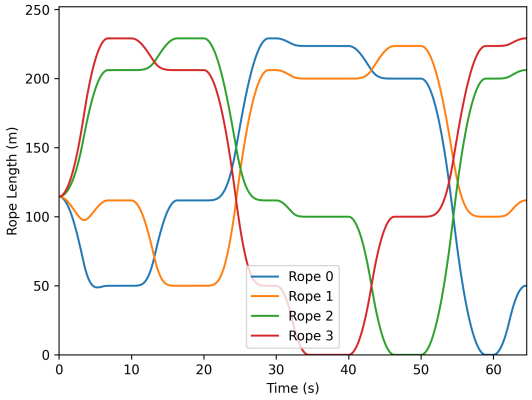


(b) Veränderung der Seillängen für Test `diagonals`

Abbildung 23: Ergebnisse für Test `diagonals`

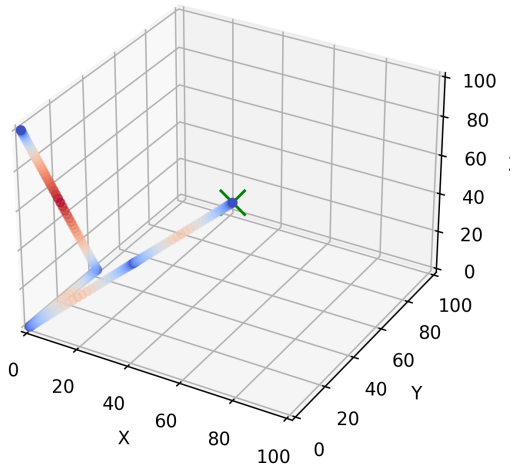


(a) Bewegung der Spidercam für Test `edges`

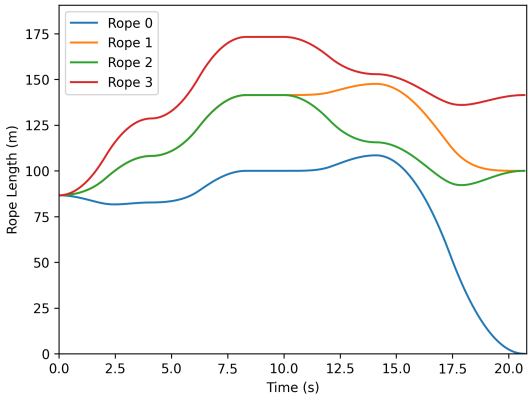


(b) Veränderung der Seillängen für Test `edges`

Abbildung 24: Ergebnisse für Test `edges`

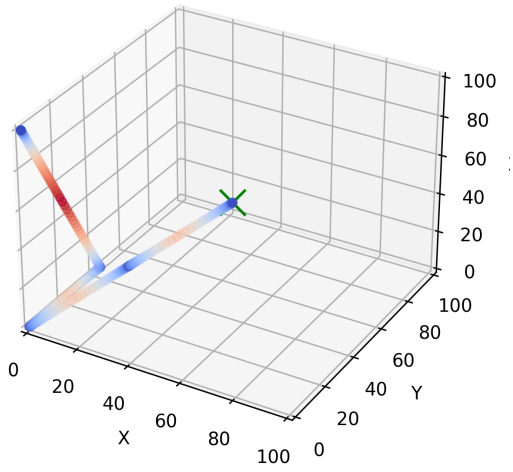


(a) Bewegung der Spidercam für Test `multiple_per_step`

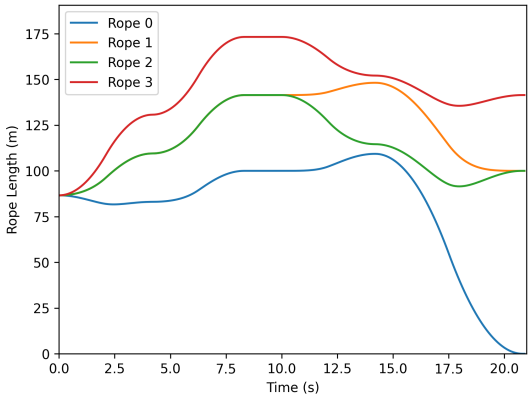


(b) Veränderung der Seillängen für Test `multiple_per_step`

Abbildung 25: Ergebnisse für Test `multiple_per_step`

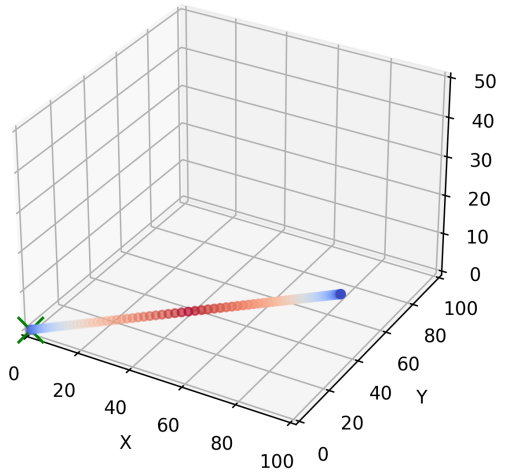


(a) Bewegung der Spidercam für Test `multiple_simultaneously`

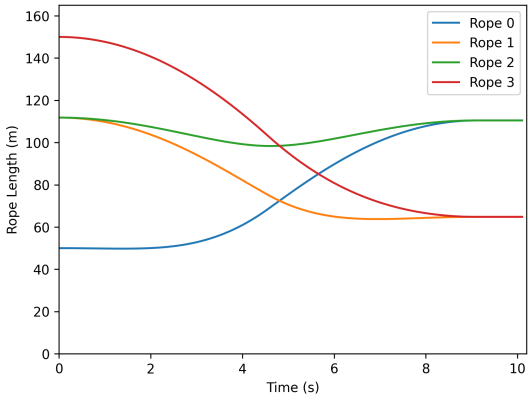


(b) Veränderung der Seillängen für Test `multiple_simultaneously`

Abbildung 26: Ergebnisse für Test `multiple_simultaneously`



(a) Bewegung der Spidercam für Test `test_same_dest`



(b) Veränderung der Seillängen für Test `test_same_dest`

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 usprünglichen 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/
  ```

[5] Dabei werden zusätzliche Informationen geloggt.

A.2.2 Fehlermeldungen

Folgende Fehlermeldungen können auftreten:

- `FileNotFoundError`: 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

[6]Vorausgesetzt, 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

```
# Beispiel
dim 70 100 30
start 10 80 10
vmax 6
amax 2
freq 2
0 50 40 30
```

Listing 3: Testdatei `IHK_1.txt`

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

Listing 4: Testdatei `IHK_2.txt`

```
# Example which travels in an approximate spiral
dim 100 100 1000
start 50 50 0
vmax 50
amax 10
freq 2
# t x y z
0 50 0 20
5 100 50 40
10 50 100 60
15 0 50 80
20 50 0 100
25 100 50 120
30 50 100 140
35 0 50 160
40 50 0 180
45 100 50 200
50 50 100 220
55 0 50 240
60 50 0 260
65 100 50 280
70 50 100 300
75 0 50 320
80 50 0 340
85 100 50 360
```

```
90 50 100 380
95 0 50 400
100 50 0 420
105 100 50 440
110 50 100 460
115 0 50 480
120 50 0 500
125 100 50 520
130 50 100 540
135 0 50 560
140 50 0 580
145 100 50 600
150 50 100 620
155 0 50 640
160 50 0 660
165 100 50 680
170 50 100 700
175 0 50 720
180 50 0 740
185 100 50 760
190 50 100 780
195 0 50 800
200 50 0 820
205 100 50 840
210 50 100 860
215 0 50 880
220 50 0 900
225 100 50 920
230 50 100 940
235 0 50 960
240 50 0 980
245 100 50 1000
```

Listing 5: Testdatei `test_spiral.txt`

```
# Beispiel einer einzigen Instruktion
dim 100 100 50
start 0 0 0
vmax 25
amax 5
freq 10
0 90 50 10
```

Listing 6: Testdatei `test_single.txt`

```
# Example which travels diagonally
dim 100 200 50
start 0 0 0
vmax 50
amax 10
freq 5
# t x y z
```

```
20 100 200 50
40 0 200 0
60 100 0 50
80 0 0 0
```

Listing 7: Testdatei `test_diagonals.txt`

```
# Example which travels to each edge in the graph once after 10 seconds
dim 100 200 50
start 50 100 25
vmax 50
amax 10
freq 10
# t x y z
0 0 0 0
10 100 0 0
20 100 200 0
30 100 200 50
40 0 200 50
50 0 0 50
60 0 0 0
```

Listing 8: Testdatei `test_edges.txt`

```
# Example which has multiple instructions within a discrete timespan
dim 100 100 100
start 50 50 50
vmax 50
amax 10
freq 10
0 0 0 0
2.05 100 100 100 # opposite direction
2.1 0 0 0 # should continue in the previous direction
10 0 100 0
12.05 0 100 100 # slight change in direction
12.1 0 0 100 # should continue with this direction
```

Listing 9: Testdatei `test_multiple_per_step.txt`

```
# Example which has multiple instructions within a discrete timespan
dim 100 100 100
start 50 50 50
vmax 50
amax 10
freq 10
0 0 0 0
2.1 100 100 100 # opposite direction
2.1 0 0 0 # should continue in the previous direction
10 0 100 0
12.1 0 100 100 # slight change in direction
12.1 0 0 100 # should continue with this direction
```

Listing 10: Testdatei `test_multiple_simultaneously.txt`

```
# Example where two consecutive instructions have the same destination
dim 100 100 50
start 0 0 0
vmax 25
amax 5
freq 10
0 90 50 10
10 90 50 10
```

Listing 11: Testdatei `test_same_dest.txt`

```
# Example to test invalid dimension
dim -1 100 30
start 10 80 10
vmax 6
amax 2
freq 2
0 50 40 30
20 10 80 10
22 50 40 30
23 35 50 30
27.5 10 80 20
```

Listing 12: Testdatei `test_invalid_dim.txt`

```
# Example to test velocity limit
dim 100 100 30
start 10 80 10
vmax -1
amax 2
freq 2
0 50 40 30
20 10 80 10
22 50 40 30
23 35 50 30
27.5 10 80 20
```

Listing 13: Testdatei `test_invalid_vmax.txt`

```
# Example to test acceleration limit
dim 100 100 30
start 10 80 10
vmax 6
amax -1
freq 2
0 50 40 30
20 10 80 10
22 50 40 30
23 35 50 30
27.5 10 80 20
```

Listing 14: Testdatei `test_invalid_amax.txt`

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

Listing 15: Testdatei `test_invalid_freq.txt`

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

Listing 16: Testdatei `test_invalid_instruction.txt`

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

Listing 17: Testdatei `test_invalid_no_instruction.txt`

E Quellcode

```
"""
.. include:: ../README.md
"""

from .controller import *
from .io import *
from .movement import *
from .phase import *
from .spidercam import *
from .plotter import *
```

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

```
import argparse
import configparser
import logging
import os
import sys

import spidercam_simulator


def setup():
    """Sets up the logger and config"""

    # reading the config file
    cfg = configparser.ConfigParser()
    # parent of os.path.dirname(__file__)
    cfg_file = os.path.join(os.path.dirname(os.path.dirname(__file__)), "config.ini")
    print(f"Config file: {cfg_file}")
    cfg.read(cfg_file)

    # setting up the logger
    logging_dir = spidercam_simulator.find_location(cfg["io"]["logging_dir"])

    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s - %(name)s - %(levelname)s - %(message)s (%(filename)s:%(
    lineno)d)",
        datefmt="%Y/%m/%d %I:%M:%S %p",
        handlers=[
            logging.FileHandler(
                filename=os.path.join(
                    os.path.dirname(os.path.dirname(os.path.abspath(__file__))),
                    "logs",
                    "spidercam_simulator.log",
                ),
                mode="w",
            ),
        ],
```

```
    )

    logging.info("Logging initialized")
    print(f"Logging to {os.path.join(logging_dir, 'spidercam_simulator.log')}")

    # Logging configuration
    for section in cfg.sections():
        for key, value in cfg[section].items():
            logging.debug("Config: %s.%s = %s", section, key, value)

    logging.debug("Config file read")

    # Setup: Setting up Arguments
    logging.info("Reading arguments")

    parser = argparse.ArgumentParser()
    parser.add_argument("--input", "-i", help="File or directory to parse")
    parser.add_argument(
        "--output", "-o", help="Output directory (needs to be different from input)"
    )
    parser.add_argument("--debug", "-d", help="Debug mode", action="store_true")
    parser.add_argument(
        "--no-plot", "-np", help="Disable plotting", action="store_true"
    )
    ags = parser.parse_args()

    if ags.debug:
        # logging.getLogger().addHandler(logging.StreamHandler())
        logging.getLogger().setLevel(logging.DEBUG)

        logging.info("Debug mode enabled")

    logging.debug("Arguments read")

    # logging configuration
    for key, value in vars(ags).items():
        logging.debug("Argument: %s = %s", key, value)

    # use defaults if no arguments are passed
    if ags.input is None:
        ags.input = cfg["io"]["input_dir"]

    ags.input = spidercam_simulator.find_location(ags.input)

    logging.debug("Input file/directory: %s", ags.input)

    if ags.output is None:
        ags.output = cfg["io"]["output_dir"]

    ags.output = spidercam_simulator.find_location(ags.output)

```

```
    logging.debug("Output directory: %s", ags.output)

    # check if files/directories exist
    if not os.path.exists(ags.input):
        logging.error("Input file/directory %s does not exist", ags.input)
        sys.exit(1)

    if not os.path.exists(ags.output):
        logging.error(
            "Output directory %s does not exist. Please create it first.", ags.output
        )
        sys.exit(1)

    # check if input and output are the same
    if os.path.samefile(ags.input, ags.output):
        logging.error("Input and output directories are the same")
        sys.exit(1)

    logging.info("Setup complete")
    return cfg, ags


config, args = setup()

logging.info("Starting spidercam_simulator")

input_queue = [args.input] if os.path.isfile(args.input) else os.listdir(args.input)

logging.info("Found %s files/directories to process", len(input_queue))

for file in input_queue:
    if file == ".gitkeep":
        continue

    logging.info("Processing %s", file)

    file_handler = spidercam_simulator.FileHandler(
        os.path.join(args.input, file), args.output
    )
    contents = file_handler.read_file()

    logging.debug("File contents: %s", contents)

    # parse the file but skip if error
    try:
        input_dict = spidercam_simulator.Parser.parse_input(contents)
    except ValueError as exc:
        logging.error("Error parsing file: %s", exc)
        print(f"Error parsing file {file}: {exc}")
        continue

```

```
    logging.debug("Input dictionary: %s", input_dict)

    controller = spidercam_simulator.Controller.from_dict(input_dict)

    logging.info("Running controller %s", repr(controller))

    cam_positions, rope_lengths = controller.run()

    logging.debug("Cam positions: %s", cam_positions)
    logging.debug("Rope lengths: %s", rope_lengths)

    output1, output2 = spidercam_simulator.Parser.parse_output(
        input_dict["dim"], input_dict["freq"], cam_positions, rope_lengths
    )

    file_handler.write_files(output1, output2)

    if not args.no_plot:
        plotter = spidercam_simulator.Plotter(
            input_dict["dim"],
            input_dict["freq"],
            cam_positions,
            rope_lengths,
            args.output,
            os.path.splitext(file)[0],
        )

        plotter.plot()

    # cleanup
    del file_handler
    del contents
    del input_dict
    del controller
    del cam_positions
    del rope_lengths
    del output1
    del output2

    logging.info("Finished processing %s", file)

    # file_handler.write_file(parsed)

# Ending the program
logging.info("Ending spidercam_simulator")
```

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

```
from __future__ import annotations

import logging
import numpy as np
```

```python

import spidercam_simulator


class Controller:
    """ A class for initializing and controlling the spidercam"""

    def __init__(
        self,
        dim: tuple = (1, 1, 1),
        start: tuple = (0, 0, 0),
        max_velocity: float = 1.0,
        acceleration: float = 1.0,
        freq: float = 1.0,
        instructions: list[spidercam_simulator.Instruction] = None,
    ) -> None:
        """ Initializes the Controller class

        Args:
            dim (tuple, optional): The dimensions of the field. Defaults to (1, 1, 1).
            start (tuple, optional): The starting position of the spidercam. Defaults to (0, 0, 0).
            max_velocity (float, optional): The maximum velocity of the spidercam. Defaults to 1.0.
            acceleration (float, optional): The maximum acceleration of the spidercam. Defaults to 1.0.
            freq (float, optional): The discrete time frequency. Defaults to 1.0.
            instructions (list, optional): The instructions for the spidercam. Defaults to [].

        Returns:
            None
        """
        self.logger = logging.getLogger(__name__)
        self.logger.debug(
            "Initializing Controller with dim=%s, start=%s, max_velocity=%s, acceleration=%s, freq=%s, instructions=%s",
            dim,
            start,
            max_velocity,
            acceleration,
            freq,
            instructions,
        )

        self.dim = dim
        self.spidercam = spidercam_simulator.Spidercam(
            self, max_velocity, acceleration, start
        )
        self.freq = freq
        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
    =})"
```

```python

    def run(self) -> tuple[list[tuple], list[tuple]]:
        """ Runs the instructions for the spidercam

        Returns:
            tuple[list[tuple], list[tuple]]: The positions and rope lengths for each time step
        """

        current_time = 0

        while True:
            self.logger.info("Current time: %s", current_time)

            if len(self.instructions) != 0:
                # Check if there is a new instruction
                if current_time >= self.instructions[0].start_time:
                    instruction = self.instructions.pop(0)

                    self.spidercam.move(instruction)

            # Store cam_positions and rope_lengths
            position = self.spidercam.get_position(current_time)
            self.logger.info("Current position: %s", position)

            self.cam_positions.append(position)
            self.rope_lengths.append(self.get_rope_lengths(position))

            # Check if the last instruction is finished
            if (
                len(self.instructions) == 0
                and self.spidercam.movements[-1].end_time() < current_time
                and self.spidercam.queue is None
            ):
                self.logger.info(
                    "Last instruction finished and no more instructions left"
                )
                break

            current_time += 1 / self.freq

        return self.cam_positions, self.rope_lengths

    def get_rope_lengths(self, position: tuple) -> list[tuple]:
        """ Returns the rope lengths for a given position

        Returns:
            list[tuple]: The rope lengths
        """

        rope_lengths = [
```

```
            np.linalg.norm(np.array(anchor) - np.array(position))
            for anchor in self.anchors
        ]

        self.logger.debug("Rope lengths: %s", rope_lengths)

        return rope_lengths
```

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

```
import logging
import os

import numpy as np


class FileHandler:
    """ A class for handling input and output"""

    def __init__(self, input_file: str, output_dir: str) -> None:
        """Initializes the FileHandler class

        Args:
            input_file (str): The path to the file to read
            output_dir (str): The path to the directory to write

        Returns:
            None
        """

        self.logger = logging.getLogger(__name__)
        self.logger.debug(
            "Initializing FileHandler with input_file %s and output_dir %s",
            input_file,
            output_dir,
        )

        self.input_file = input_file
        self.output_dir = output_dir
        # removes file ending from input
        self.output_files = (
            os.path.join(
                output_dir, os.path.basename(input_file).split(".")[0] + "_1.csv"
            ),
            os.path.join(
                output_dir, os.path.basename(input_file).split(".")[0] + "_2.csv"
            ),
        )

    def read_file(self) -> str:
        """Reads the file

```

```
        Returns:
            str: The contents of the file
        """

        self.logger.info("Reading file %s", self.input_file)

        with open(self.input_file, "r", encoding="utf-8") as file:
            return file.read()

    def write_files(self, output1: str, output2: str) -> None:
        """Writes the output to two files

        Args:
            output1 (str): The contents to write to the first file
            output2 (str): The contents to write to the second file

        Returns:
            None
        """

        self.logger.info(
            "Writing output to files %s and %s",
            self.output_files[0],
            self.output_files[1],
        )

        with open(self.output_files[0], "w", encoding="utf-8") as file:
            file.write(output1)

        with open(self.output_files[1], "w", encoding="utf-8") as file:
            file.write(output2)


def find_location(file: str) -> str:
    """ Finds the location of the given file/directory
    Looks in the following locations:
        - The current working directory
        - The directory of the script
        - The directory of the script's parent
        - The home directory
        - The root directory

    Args:
        file (str): The path to the given file/directory

    Returns:
        str: The absolute path to the given file/directory
    """

    try:
        logger = logging.getLogger(__name__)
```

```
        logger.debug("Finding location of file/directory %s", file)
    except NameError:
        pass

    locations = [
        os.getcwd(),  # Current working directory
        os.path.dirname(os.path.realpath(__file__)),  # Directory of the script
        os.path.dirname(
            os.path.dirname(os.path.realpath(__file__))
        ),  # Directory of the script's parent
        os.path.expanduser("~"),  # Home directory
        os.path.sep,  # Root directory
    ]

    for location in locations:
        logger.debug("Checking location %s", location)
        if os.path.exists(os.path.join(location, file)):
            # log absolute path
            logger.debug("Found file/directory at %s", os.path.join(location, file))
            return os.path.join(location, file)

    raise FileNotFoundError(f"Could not find file/directory {repr(file)}")


class Parser:
    """ A class for parsing input strings"""

    logger = logging.getLogger(__name__)

    @staticmethod
    def parse_input(string: str) -> dict:
        """Parses the given string

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

        Args:
```

```
            string (str): The string to parse

        Returns:
            dict: The parsed string
                - dim (tuple): The dimensions of the field
                - start (tuple): The starting position of the spidercam
                - max_velocity (float): The maximum velocity of the spidercam
                - acceleration (float): The maximum acceleration of the spidercam
                - freq (float): The discrete time frequency
                - instructions (list): The instructions for the spidercam
        """

        Parser.logger.debug("Parsing string %s", string)

        lines = string.splitlines()
        instructions = []

        # Clean up the lines and remove comments
        for i, line in enumerate(lines):
            lines[i] = line.split("#")[0].strip()

        # Remove empty lines
        lines = list(filter(None, lines))

        # Parse the lines
        for i, line in enumerate(lines):
            if line.startswith("dim"):
                dim = tuple(map(int, line.split()[1:]))
            elif line.startswith("start"):
                start = tuple(map(int, line.split()[1:]))
            elif line.startswith("vmax"):
                max_velocity = float(line.split()[1])
            elif line.startswith("amax"):
                acceleration = float(line.split()[1])
            elif line.startswith("freq"):
                freq = float(line.split()[1])
            else:
                instructions.append(Instruction.parse(line))

        # Check if the dimensions are valid
        if len(dim) != 3 or any(d <= 0 for d in dim):
            raise ValueError("The dimensions are invalid")

        # Check if the starting position is valid
        if len(start) != 3 or not all(0 <= i <= j for i, j in zip(start, dim)):
            raise ValueError("The starting position is invalid")

        # Check if the maximum velocity is valid
        if max_velocity <= 0:
            raise ValueError("The maximum velocity is invalid")

```

```
        # Check if the maximum acceleration is valid
        if acceleration <= 0:
            raise ValueError("The maximum acceleration is invalid")

        # Check if the discrete time frequency is valid
        if freq <= 0:
            raise ValueError("The discrete time frequency is invalid")

        # Check if the instructions are valid
        if len(instructions) == 0:
            raise ValueError("No instructions found")

        for i, instruction in enumerate(instructions):
            if not all(0 <= j <= k for j, k in zip(instruction.destination, dim)):
                raise ValueError(f"Instruction {i + 1} is invalid")

        return {
            "dim": dim,
            "start": start,
            "max_velocity": max_velocity,
            "acceleration": acceleration,
            "freq": freq,
            "instructions": instructions,
        }

    # returns two strings
    @staticmethod
    def parse_output(
        dim: tuple,
        freq: float,
        cam_positions: list[tuple],
        rope_lengths: list[tuple],
    ) -> tuple:
        """Parses the output

        Example file contents (first output: lengths of the ropes):
        ```
        83.06623, 82.9059, [...], 64.0056, 64.0312 # Rope 1
        101.9803, 101.9803, [...], 101.9803, 101.9803 # Rope 2
        101.9803, 101.9803, [...], 101.9803, 101.9803 # Rope 3
        101.9803, 101.9803, [...], 101.9803, 101.9803 # Rope 4
        ```

        Example file contents (second output: dimensions, time stamps, positions of
    the camera):
        ```
        70, 100, 30 # Dimensions
        0.0, 0.5, 1.0, [...], 20.5, 21.0 # Time stamps
        10.0, 10.16, 10.66, [...], 20.0, 20.0 # x coordinates
        80.0, 80.0, 80.0, [...], 80.0, 80.0 # y coordinates
        10.0, 10.0, 10.0, [...], 10.0, 10.0 # z coordinates
```
```

```
        ```


        Args:
            dim (tuple): The dimensions of the field
            freq (float): The discrete time frequency
            cam_positions (list[tuple]): The positions of the camera
            rope_lengths (list[tuple]): The lengths of the ropes

        Returns:
            str: The parsed output
        """

        Parser.logger.info(
            "Parsing output with #cam_positions=%d and #rope_lengths=%d",
            len(cam_positions),
            len(rope_lengths),
)

        # Using numpy to transform the lists into numpy arrays for easier handling
        cam_positions = np.array(cam_positions)
        rope_lengths = np.array(rope_lengths)

        # transposing the arrays to get the correct shape
        cam_positions = cam_positions.T
        rope_lengths = rope_lengths.T

        # creating the time stamps
        time_stamps = np.arange(0, len(cam_positions[0]) / freq, 1 / freq)

        # creating the output strings
        output1 = "\n".join(",".join(map(str, rope)) for rope in rope_lengths)

        # adding the dimensions to the output
        output2 = f"{dim[0]},{dim[1]},{dim[2]}\n"

        # adding the time stamps to the output
        output2 += ",".join(map(str, time_stamps)) + "\n"

        # adding the positions of the camera to the output
        output2 += ",".join(map(str, cam_positions[0])) + "\n"
        output2 += ",".join(map(str, cam_positions[1])) + "\n"
        output2 += ",".join(map(str, cam_positions[2])) + "\n"

        # output2 = "\n".join(
        #   ",".join(map(str, line)) for line in [dim, time_stamps, *cam_positions]
        #)

        Parser.logger.debug("Parsed output")

        return output1, output2
```
```

```python


class Instruction:
    """ A class for defining an instruction"""

    def __init__(self, start_time: float = 0.0, destination: tuple = (0, 0, 0)) ->
    None:
        """Initializes the Instruction class

        Args:
            start_time (float, optional): The time to start the instruction. Defaults
    to 0.0.
            destination (tuple, optional): The destination of the instruction.
    Defaults to (0, 0, 0).

        Returns:
            None
        """
        self.logger = logging.getLogger(__name__)
        self.logger.debug(
            "Initializing Instruction with start_time=%f and destination=%s",
            start_time,
            destination,
        )
        self.start_time = start_time
        self.destination = destination

    @classmethod
    def parse(cls, data: str) -> "Instruction":
        """Parses an instruction from a string

        Args:
            data (str): The string to parse the instruction from
                - format: start_time x y z

        Returns:
            Instruction: The instruction
        """
        cls.logger = logging.getLogger(__name__)
        cls.logger.debug("Parsing instruction from %s", data)

        try:
            start_time, x, y, z = data.split(" ")
        except ValueError as exp:
            raise ValueError(
                f"Invalid instruction format (expected: start_time x y z), got: {data}
    "
            ) from exp

        return cls(float(start_time), (float(x), float(y), float(z)))

```

```
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         return (
352             f"Instruction(start_time={self.start_time}, destination={self.destination})"
353         )
```

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 0.
24             start (tuple, optional): The starting position. Defaults to (0, 0, 0).
25             destination (tuple, optional): The destination position. Defaults to (0, 0, 0).
26         Returns:
27             None
28         """
29         self.logger = logging.getLogger(__name__)
30         self.logger.info(
31             "Initializing Movement at %s with start %s and destination %s",
32             start_time,
33             start,
34             destination,
```

```
        )

        self.spidercam = spidercam
        self.start_time = start_time

        self.phases = []

        self.calculate_phases(start, destination)

    def start(self) -> tuple:
        """ Returns the start of the movement

        Returns:
            tuple: The start of the movement
        """
        return self.phases[0].start

    def destination(self) -> tuple:
        """ Returns the destination of the movement

        Returns:
            tuple: The destination of the movement
        """
        return self.phases[-1].destination

    def duration(self) -> float:
        """ Returns the duration of the movement

        Returns:
            float: The duration of the movement
        """
        return sum(phase.duration for phase in self.phases)

    def distance(self) -> float:
        """ Returns the distance of the movement

        Returns:
            float: The distance of the movement
        """
        return sum(phase.distance for phase in self.phases)

    def end_time(self) -> float:
        """ Returns the end time of the movement

        Returns:
            float: The end time of the movement
        """
        return self.start_time + self.duration()

    def calculate_phases(self, start: tuple, destination: tuple) -> None:
        """ Calculates the phases of the movement
```

```
        Args:
            start (tuple): The starting position
            destination (tuple): The destination position

        Returns:
            None
        """
        # HERE 3
        self.logger.info("Calculating phases")

        distance = np.linalg.norm(np.array(destination) - np.array(start))

        # Check how many phases are needed
        if distance <= 2 * self.spidercam.dist_vmax:
            # Only acceleration and deceleration
            self.logger.debug("Only acceleration and deceleration needed")

            # Calculate middle point
            middle_point = start + (np.array(destination) - np.array(start)) / 2
            self.phases = [
                spidercam_simulator.Phase(
                    self,
                    spidercam_simulator.Phase.Mode.ACCELERATION,
                    start,
                    middle_point,
                    0.0,
                ),
                spidercam_simulator.Phase(
                    self,
                    spidercam_simulator.Phase.Mode.DECELERATION,
                    middle_point,
                    destination,
                    np.sqrt(self.spidercam.acceleration * distance),
                ),
            ]

        else:
            # Acceleration, constant velocity and deceleration
            self.logger.debug("Acceleration, constant velocity and deceleration needed
    ")

            # Calculate needed points
            point_a = (
                np.array(start)
                + (np.array(destination) - np.array(start))
                * self.spidercam.dist_vmax
                / distance
            )

            point_b = (
```

```
                np.array(destination)
                - (np.array(destination) - np.array(start))
                * self.spidercam.dist_vmax
                / distance
            )

            self.phases = [
                spidercam_simulator.Phase(
                    self,
                    spidercam_simulator.Phase.Mode.ACCELERATION,
                    start,
                    point_a,
                    0.0,
                ),
                spidercam_simulator.Phase(
                    self,
                    spidercam_simulator.Phase.Mode.CONSTANT_VELOCITY,
                    point_a,
                    point_b,
                    self.spidercam.max_velocity,
                ),
                spidercam_simulator.Phase(
                    self,
                    spidercam_simulator.Phase.Mode.DECELERATION,
                    point_b,
                    destination,
                    self.spidercam.max_velocity,
                ),
            ]

    def start_deceleration(self, time: float) -> None:
        """ Starts deceleration

        Args:
            time (float): The time to start deceleration

        Returns:
            None
        """

        self.logger.info("Starting deceleration at %s", time)

        # Getting the phase that is active at the time
        # and the time offset of the phase
        # update all phases after the phase

        time_sum = self.start_time

        for phase in self.phases:
            # Skip phases before the time
            if time_sum + phase.duration < time:
```

```
                time_sum += phase.duration
                continue

            # Nothing to do if already decelerating
            if phase.mode == spidercam_simulator.Phase.Mode.DECELERATION:
                self.logger.debug("Already decelerating, nothing to do")
                return

            offset = time - time_sum

            # Found phase should end at offset
            phase.destination = phase.get_position(offset)
            phase.update()

            # Update next phases
            # There always is a next phase because the final phase is deceleration and the previous phase was not
            next_phase = self.phases[self.phases.index(phase) + 1]

            next_phase.start = phase.destination
            next_phase.starting_velocity = (
                self.spidercam.max_velocity
                if phase.mode is spidercam_simulator.Phase.Mode.CONSTANT_VELOCITY
                else self.spidercam.acceleration * offset
            )
            next_phase.mode = spidercam_simulator.Phase.Mode.DECELERATION
            next_phase.destination = next_phase.get_position(
                next_phase.starting_velocity / self.spidercam.acceleration
            )
            next_phase.update()

            # If there is a phase after the next phase, pop it
            if len(self.phases) > self.phases.index(next_phase) + 1:
                self.phases.pop(self.phases.index(next_phase) + 1)

            break

    def get_phase(self, time: float) -> spidercam_simulator.Phase:
        """ Returns the phase at the given time

        Args:
            time (float): The time to get the phase at

        Returns:
            Phase: The phase at the given time
        """
        time_sum = self.start_time

        for phase in self.phases:
            if time_sum + phase.duration > time:
                return phase
```

```
            time_sum += phase.duration

        return self.phases[-1]

    def __repr__(self):
        return f"Movement(start={repr(self.start())}, destination={repr(self.
    destination())}, start_time={self.start_time}, duration={self.duration()},
    distance={self.distance()})"

    def get_position(self, time: float) -> tuple:
        """ Returns the position of the movement at the time

        Args:
            time (float): The time to get the position for

        Returns:
            tuple: The position of the movement at the time
        """

        self.logger.info("Getting position for time %s", time)

        # Getting the phase that is active at the time
        # and the time offset of the phase
        time_sum = self.start_time

        for phase in self.phases:
            if time_sum + phase.duration >= time:
                return phase.get_position(time - time_sum)

            time_sum += phase.duration

        # Just for debugging
        raise Exception("No phase found")
```

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

```
from __future__ import annotations

import logging
from enum import Enum

import numpy as np
import spidercam_simulator


class Phase:
    """ A class for defining a phase"""

    Mode = Enum("Mode", ["ACCELERATION", "CONSTANT_VELOCITY", "DECELERATION"])

    def __init__(
        self,
```

```
        movement: spidercam_simulator.Movement,
        mode: Mode,
        start: tuple = (0, 0, 0),
        destination: tuple = (0, 0, 0),
        starting_velocity: float = 0,
    ) -> None:
        """ Initializes the Phase class

        Args:
            movement (Movement): The movement instance
            mode (Mode): The type of phase. Can be ACCELERATION, CONSTANT_VELOCITY, or
     DECELERATION
            start (tuple, optional): The starting position. Defaults to (0, 0, 0).
            destination (tuple, optional): The destination position. Defaults to (0,
    0, 0).
            starting_velocity (float, optional): The starting velocity. Defaults to 0.

        Returns:
            None
        """
        self.logger = logging.getLogger(__name__)
        self.logger.info(
            "Initializing Phase with mode %s, start %s, destination %s, and starting
    velocity %s",
            mode,
            start,
            destination,
            starting_velocity,
        )

        self.movement = movement
        self.mode = mode
        self.start = start
        self.destination = destination
        self.starting_velocity = starting_velocity

        self.update()

    def update(self) -> None:
        """ Updates the phase

        Args:
            None

        Returns:
            None
        """
        self.logger.debug("Updating phase %s", self.mode)

        self.distance = self.calc_distance()
        self.duration = self.calc_duration()
```

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

```
        #     return self.destination

        if self.mode == Phase.Mode.ACCELERATION:
            distance = self.movement.spidercam.acceleration * offset**2 / 2
        elif self.mode == Phase.Mode.CONSTANT_VELOCITY:
            distance = self.starting_velocity * offset
        elif self.mode == Phase.Mode.DECELERATION:
            distance = (
                self.starting_velocity * offset
                - self.movement.spidercam.acceleration * offset**2 / 2
            )

        self.logger.debug("Phase distance: %s", distance)

        # if distance almost 0, return start
        if np.isclose(distance, 0):
            self.logger.debug(
                "Distance is almost 0, returning start %s", repr(self.start)
            )
            return self.start

        position = np.array(self.start) + (
            np.array(self.destination) - np.array(self.start)
        ) * distance / np.linalg.norm(np.array(self.destination) - np.array(self.start
    ))

        self.logger.debug("Phase position: %s", position)

        return tuple(position)
```

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

```
from __future__ import annotations
import logging
import os
import matplotlib.pyplot as plt
import numpy as np


class Plotter:
    """ A class for plotting camera positions and rope lengths"""

    def __init__(
        self,
        dim: tuple = None,
        freq: float = 1.0,
        cam_positions: list = None,
        rope_lengths: list = None,
        output_dir: str = None,
        name: str = None,
    ) -> None:
        """ Initializes the Plotter class
```

```
        Args:
            dim (tuple, optional): The dimensions of the field. Defaults to None.
            freq (float, optional): The discrete time frequency. Defaults to 1.0.
            cam_positions (list, optional): The camera positions. Defaults to None.
            rope_lengths (list, optional): The rope lengths. Defaults to None.
            output_dir (str, optional): The output directory. Defaults to None.
            name (str, optional): The name of the plot. Defaults to None.

        Returns:
            None
        """

        self.logger = logging.getLogger(__name__)
        # Setting log level info to suppress matplotlib font manager warnings
        self.logger.setLevel(logging.INFO)

        self.logger.info("Initializing Plotter for %s", name)

        self.dim = dim
        self.freq = freq
        self.cam_positions = np.array(cam_positions)
        self.rope_lengths = np.array(rope_lengths)
        self.output_dir = output_dir
        self.name = name

    def plot_cam_positions(self) -> None:
        """ Plots the camera positions

        Returns:
            None
        """

        self.logger.info("Plotting camera positions for %s", self.name)

        # If output directory is not specified, plot to screen
        if self.output_dir is None:
            plt.ion()

        # Plot camera positions projection = 3d
        fig = plt.figure()
        ax = fig.add_subplot(111, projection="3d")

        # Set axis labels
        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.set_zlabel("Z")

        # Set axis limits
        ax.set_xlim3d(0, self.dim[0])
        ax.set_ylim3d(0, self.dim[1])
```

```
        ax.set_zlim3d(0, self.dim[2])

        # Rotate so that origin is in the bottom left and z is up, angle is normal
        # ax.view_init(azim=0, elev=90)

        # Plot camera positions
        xline = self.cam_positions[:, 0]
        yline = self.cam_positions[:, 1]
        zline = self.cam_positions[:, 2]

        # Define color as distance from one point to the previous
        color = np.zeros(len(xline))
        for i in range(1, len(xline)):
            color[i] = np.linalg.norm(
                np.array([xline[i], yline[i], zline[i]])
                - np.array([xline[i - 1], yline[i - 1], zline[i - 1]])
            )

        # Plot camera positions
        ax.scatter(xline, yline, zline, c=color, cmap="coolwarm")

        # Plot start as big green x
        ax.scatter(xline[0], yline[0], zline[0], c="green", s=200, marker="x")

        # Set title for whole figure
        # fig.suptitle("Camera Positions for " + self.name)

        # Save or show plot
        if self.output_dir is None:
            plt.show()
        else:
            # Save at output + name + cam_pos.png
            # only get base name of file
            plt.savefig(
                f"{self.output_dir}/{os.path.basename(self.name)}_cam_pos.png",
                bbox_inches="tight",
                dpi=300,
            )

    def plot_rope_lengths(self) -> None:
        """ Plots the rope lengths

        Returns:
            None
        """

        self.logger.info("Plotting rope lengths for %s", self.name)

        # If output directory is not specified, plot to screen
        if self.output_dir is None:
            plt.ion()
```

```
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.
    name)
166
167         self.plot_cam_positions()
168         self.plot_rope_lengths()
```

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

```
1 from __future__ import annotations
2
```

```python
import logging

import spidercam_simulator


class Spidercam:
    """ A class for controlling the spidercam"""

    movements = []
    queue = None

    def __init__(
        self,
        controller: spidercam_simulator.Controller,
        max_velocity: float = 1.0,
        acceleration: float = 1.0,
        start: tuple = (0, 0, 0),
    ) -> None:
        """ Initializes the Spidercam class

        Args:
            controller (Controller): The controller instance
            max_velocity (float, optional): The maximum velocity of the spidercam. Defaults to 1.0.
            acceleration (float, optional): The maximum acceleration of the spidercam. Defaults to 1.0.
            start (tuple, optional): The starting position of the spidercam. Defaults to (0, 0, 0).

        Returns:
            None
        """

        self.logger = logging.getLogger(__name__)
        self.logger.debug(
            "Initializing Spidercam with max_velocity=%s, acceleration=%s, start=%s",
            max_velocity,
            acceleration,
            start,
        )

        self.controller = controller
        self.max_velocity = max_velocity
        self.acceleration = acceleration
        self.start = start

        self.movements = []
        self.queue = None

        self.calc_constants()

```

```
    def calc_constants(self) -> None:
        """ Calculates the following constants for the spidercam:

        - time_vmax: The time it takes to reach the maximum velocity
        $$t_{\\max} = \\frac{v_{\\max}}{a_{\\max}}$$

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

        Returns:
            None
        """

        self.logger.debug("Calculating constraints for spidercam")

        self.time_vmax = self.max_velocity / self.acceleration
        self.dist_vmax = self.acceleration * self.time_vmax**2 / 2

    def move(
        self, instruction: spidercam_simulator.Instruction = None, time: float = -1.0
    ) -> None:
        """ Moves the spidercam to a given position or updates the current movement at
         a given time

        Args:
            Instruction (Instruction): The instruction to move the spidercam. Defaults
         to None.
            time (float): The time to update the current movement at. Defaults to
    -1.0.

        Returns:
            None
        """

        # Check if parameters are valid
        if instruction is None and time == -1.0:
            self.logger.debug("No instruction or time given, returning")
            return

        # No instruction given, update the current movement
        if instruction is None and time != -1.0:
            self.logger.debug("Updating movement at time %s", time)

            # REDUNDANT?
            # If there is no movement in the queue, return
            if self.queue is None:
                self.logger.debug("No movement in queue, returning")
                return

            # If the last movement is finished, add the movement in the queue to the
    movements list
```

```
            if self.movements[-1].end_time() <= time:
                self.logger.debug(
                    "Last movement is finished, adding queue to movements list"
                )
                self.movements.append(self.queue)
                self.queue = None
                return

            return

        # Instruction given, apply the instruction
        self.logger.debug("Moving spidercam with instruction %s", instruction)

        # First movement is always possible
        if len(self.movements) == 0:
            self.logger.debug("First movement registered")

            self.movements.append(
                spidercam_simulator.Movement(
                    self, instruction.start_time, self.start, instruction.destination
                )
            )
            return

        self.logger.debug("Checking if last movement is finished")
        self.logger.debug("Last movement: %s", self.movements[-1])

        # If the last movement is not finished, decelerate and update the queue
        if self.movements[-1].end_time() > instruction.start_time:
            self.logger.debug("Last movement is not yet finished")

            self.logger.debug("Queueing instruction %s", instruction)

            # Decelerate
            self.movements[-1].start_deceleration(instruction.start_time)

            # Overwrite queue
            self.queue = spidercam_simulator.Movement(
                self,
                self.movements[-1].end_time(),
                self.movements[-1].destination(),
                instruction.destination,
            )

            return

        # REDUNDANT ?
        # If the last movement is finished and there is a queue, move to the queue
        if self.queue is not None:
            self.logger.debug("Last movement is finished and there is a queue")
            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
    destination
158         self.logger.debug("Last movement is finished and there is no queue")
159
160         self.movements.append(
161             spidercam_simulator.Movement(
162                 self,
163                 max(self.movements[-1].end_time(), instruction.start_time),
164                 self.movements[-1].destination(),
165                 instruction.destination,
166             )
167         )
168
169     def get_position(self, time: float) -> tuple:
170         """ Gets the position of the spidercam at a given time
171
172         Args:
173             time (float): The time to get the position at
174
175         Returns:
176             tuple: The position of the spidercam at the given time
177         """
178
179         self.logger.debug("Getting position at time %s", time)
180
181         self.move(time=time)
182
183         # If there are no movements, return the start position
184         if len(self.movements) == 0:
185             self.logger.debug("No movements registered, returning start position")
186             return self.start
187
188         # If the time is before the first movement, return the start position
189         if time < self.movements[0].start_time:
190             self.logger.debug("Time is before first movement, returning start position
    ")
191             return self.start
192
193         # If the time is after the last movement, return the destination of the last
    movement
194         if time > self.movements[-1].end_time():
195             self.logger.debug(
196                 "Time is after last movement, returning destination of last movement"
```

```
            )
            return self.movements[-1].destination()

        # If the time is during a movement, return the position of that movement
        for movement in self.movements:
            if movement.start_time <= time <= movement.end_time():
                self.logger.debug(
                    "Time is during movement %s, returning position of movement",
                    movement,
                )
                return movement.get_position(time)

        # If the time is between movements, return the destination of the previous movement
        for i in range(len(self.movements) - 1):
            if self.movements[i].end_time() < time < self.movements[i + 1].start_time:
                self.logger.debug(
                    "Time is between movements %s and %s, returning destination of previous movement",
                    self.movements[i],
                    self.movements[i + 1],
                )
                return self.movements[i].destination()

        # If the time is not in any of the above cases, return the start position
        self.logger.debug(
            "Time is not in any of the above cases, returning start position"
        )
```

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