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

1. 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 1. Dezember 2022

_____

Unterschrift der/des Auszubildenden

Inhaltsverzeichnis

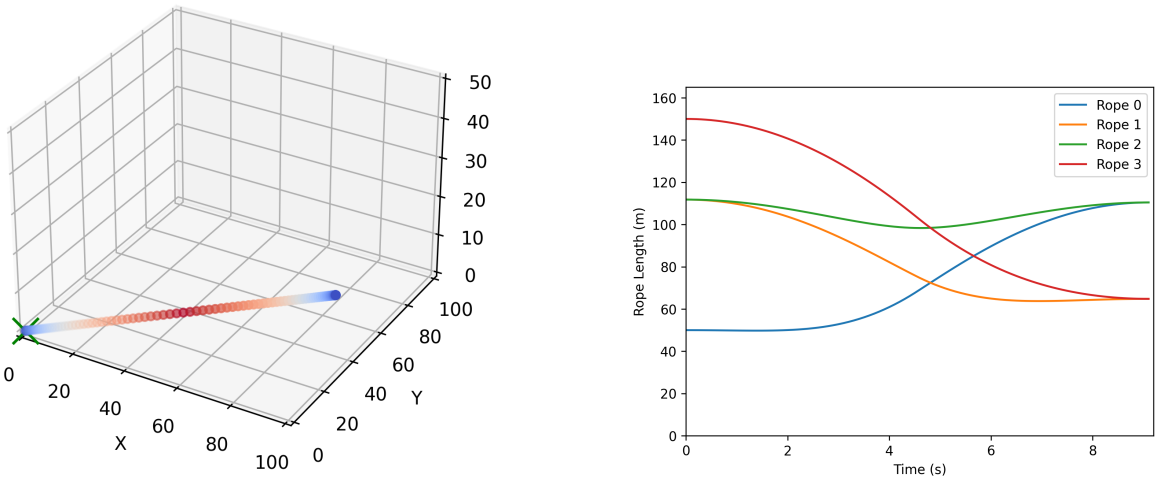
1 Aufgabenanalyse

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 **??**),
- 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.

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

Format

Die Parameter für die Simulation einer Spidercam werden in Form einer Textdatei eingelesen. Ein Beispiel für eine Eingabedatei ist in Abbildung **??** 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`).

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]

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

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 **??**) 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 **??**) 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

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:

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

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

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

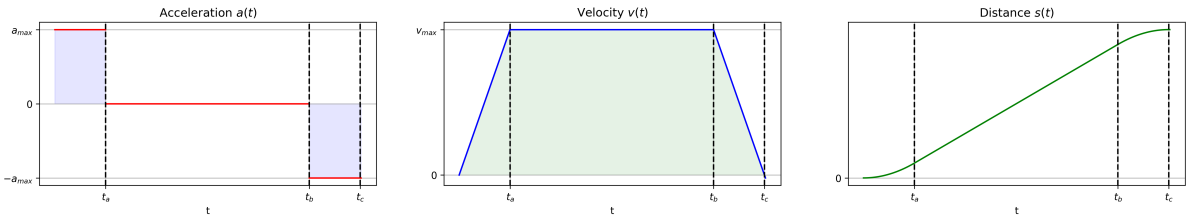


Abbildung 5: Bewegung der Spidercam in drei Phasen. t_a entspricht dem Zeitraum, in dem die Spidercam maximal beschleunigt wird. t_b ist der Zeitpunkt, an dem abgebremst wird und t_c ist der Zeitpunkt, an dem die Spidercam ihren Zielort erreicht hat.

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 **??**):

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

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

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

Eine *zweiphasige Bewegung* (siehe Abbildung **??**) 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 Abbremsphases 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.

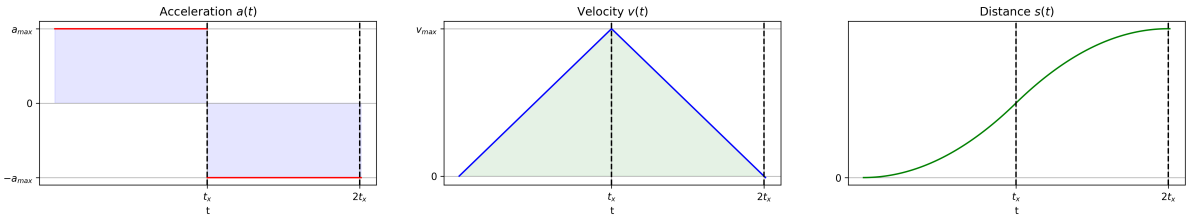


Abbildung 6: Bewegung der Spidercam in zwei Phasen. t_x ist der Zeitpunkt, an dem abgebremst wird und $2t_x$ ist der Zeitpunkt, an dem die Spidercam ihren Zielort erreicht hat.

Abbrechen der Bewegung

Instruktionen zur Bewegung der Spidercam können zu jedem Zeitpunkt 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 $v(t)$. Die Fläche unter der Kurve $v(t)$ ist die Distanz, die die Spidercam zurücklegt (siehe Abbildung **??**).

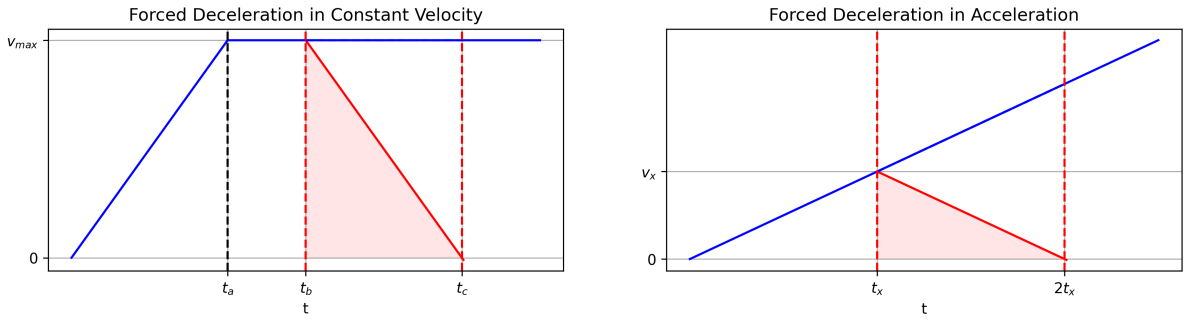


Abbildung 7: Berechnung der Bremsdistanz d_{DEC}

[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 eine Liste von Bewegungen (`movements`) und eine einelementige Warteschlange (`queue`).

2.1 Datenstrukturen

Es werden folgende Datenstrukturen verwendet:

-

2.2 Algorithmen

Die genutzten Algorithmen unterteilen sich in zwei Kategorien:

- Verarbeitung der Eingabedatei
- Erstellung der Ausgabedatei

Verarbeitung der Eingabedatei

Die Eingabe wird wie folgt verarbeitet:

1. ...
2. ...
3. ...

Erstellung der Ausgabedatei

Die Ausgabe wird wie folgt erstellt:

1. ...
2. ...
3. ...

Besonders nennenwert ist hier ....

3 Programmbeschreibung

3.1 Klassen und Schnittstellen

Wie bereits in der Verfahrensbeschreibung beschrieben, werden folgende Datenstrukturen benötigt:

- ...
- ...
- ...

Die konkrete Implementierung der Datenstrukturen erfolgt wie in Abbildung **??** zu sehen.

Besonders nennenswert ist dabei ....

3.2 Algorithmen

Mithilfe der Funktion ... wird die Eingabedatei verarbeitet und die Datenstrukturen gefüllt.

Die Ausgabedatei wird mit Hilfe der Funktion ... erstellt.

3.3 Aufrufhierarchie

Die Aufrufhierarchie ist beispielhaft in Abbildung **??** zu sehen.

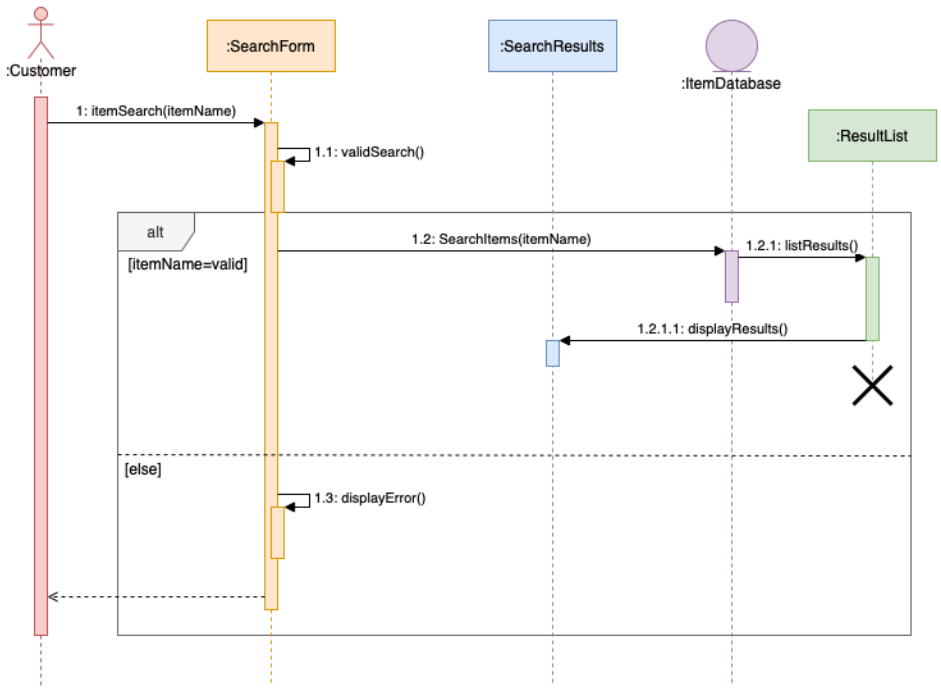


Abbildung 8: Beispiel für eine Aufrufhierarchie

4 Testing

Strategie

Die Testfälle wurden nach der Methode . . . ausgewählt. Die Testfälle sind in Tabelle **??** aufgelistet.[5]

Testfall	Eingabe	Erwartete Ausgabe	Bemerkung
Normalfälle			
Grenzfälle			
Fehlerfälle			

Abbildung 9: Beispiel für eine Tabelle mit Testfällen

Beobachtungen

Es wurden folgende Beobachtungen gemacht:

- . . .
- . . .
- . . .

[5]Die Testfälle sind im Anhang **??** zu finden.

4.1 Laufzeitanalyse

Die Laufzeitanalyse wurde mit ... durchgeführt. Jeder Testfall wurde ... mal ausgeführt und im Anschluss die Laufzeiten gemittelt.

Für die einzelnen Testfälle aus Tabelle **??** ergeben sich die Laufzeiten in Tabelle **??**.

Testfall	Median	Mittelwert	Standardabweichung
Normalfälle			
Grenzfälle			
Fehlerfälle			

Abbildung 10: Beispiel für eine Tabelle mit Laufzeiten

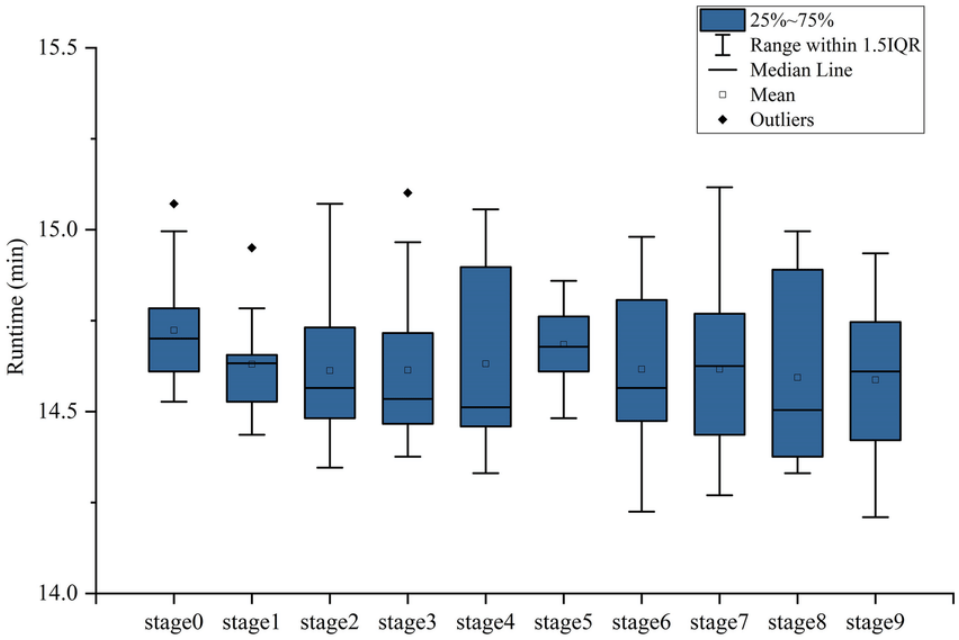


Abbildung 11: Beispiel für eine graphische Darstellung der Laufzeiten

Besonders auffällig ist ..., da der Algorithmus in $\mathcal{O}(n^m)$ läuft.

Für den praktischen Einsatz muss die Laufzeit nochmals verbessert werden. Siehe dazu Abschnitt **??**.

5 Abweichungen

Im Laufe der Entwicklung des Programms sind einige Abweichungen vom ursprünglichen Konzept aufgetreten. Diese sind:

- ...

- ...
- ...

Besonders nennenswert ist dabei ..., da ....

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wurde ein Programm zur ... entwickelt.

Dabei wurde ....

Die ... wurden ....

6.2 Ausblick

In Zukunft könnte man zur Optimierung ....

Anhang

A Grafiken

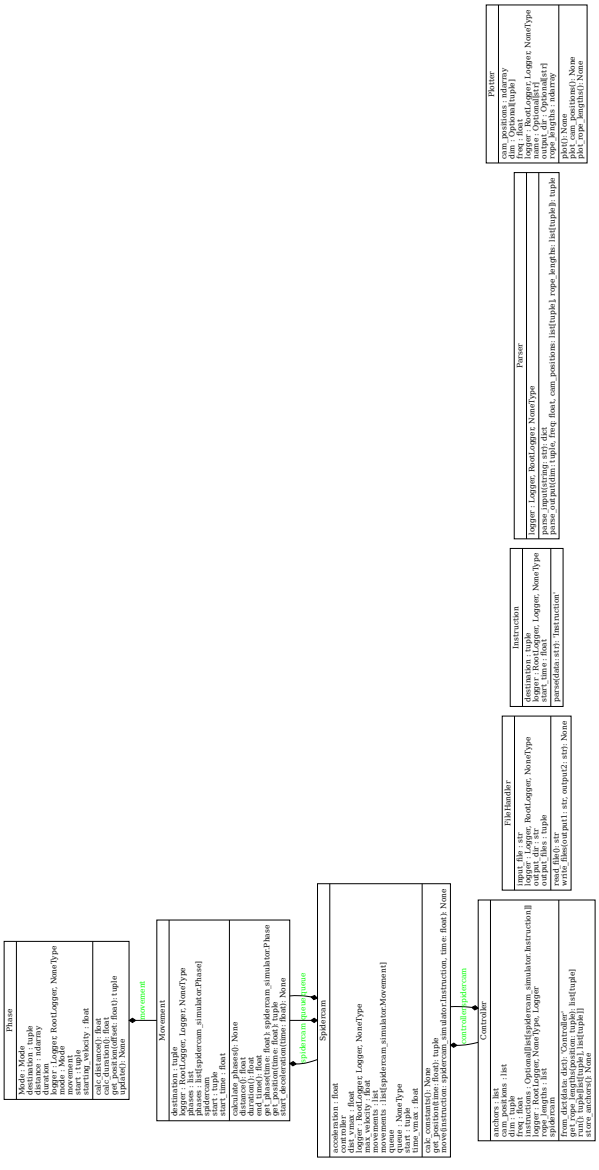


Abbildung 12: Klassendiagramm des Programms

B Benutzeranleitung

B.1 Installation

Voraussetzungen

Vorgehen

B.2 Benutzung

Eingabe

Ausgabe

Kommandozeilenparameter

Beispiele

Fehlermeldungen

C 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 **??** generiert werden.[6]

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

Abbildung 13: Generieren der Dokumentation

D Hilfsmittel

Die genutzten Hilfsmittel sind in **??** aufgeführt.

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

Typ	Tool	Hinweise
CPU	`AMD Ryzen 5 3600`	`6x 3.60 GHz`
RAM	`32 GB`	`DDR4-3200`
Betriebssystem	`Ubuntu`	20.04.1 LTS
IDE	`Visual Studio Code`	1.49.0
Compiler	`Python`	3.8.5
Debugger	`Visual Studio Code`	1.49.0
Linter	`pylint`	2.5.3
Testframework	`unittest`	3.8.5
Dokumentation	`pdoc`	0.9.2

Abbildung 14: Hilfsmittel

E Quellcode

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

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

Listing 1: Quellcode für `/python/project_name/__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()
    cfg.read("config.ini")

    # 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", "-f", 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)
```

```
71
72      # use defaults if no arguments are passed
73      if ags.input is None:
74          ags.input = cfg["io"]["input_dir"]
75
76      ags.input = spidercam_simulator.find_location(ags.input)
77
78      logging.debug("Input file/directory: %s", ags.input)
79
80      if ags.output is None:
81          ags.output = cfg["io"]["output_dir"]
82
83      ags.output = spidercam_simulator.find_location(ags.output)
84
85      logging.debug("Output directory: %s", ags.output)
86
87      # check if files/directories exist
88      if not os.path.exists(ags.input):
89          logging.error("Input file/directory %s does not exist", ags.input)
90          sys.exit(1)
91
92      if not os.path.exists(ags.output):
93          logging.error(
94              "Output directory %s does not exist. Please create it first.", ags.output
95          )
96          sys.exit(1)
97
98      # check if input and output are the same
99      if os.path.samefile(ags.input, ags.output):
100         logging.error("Input and output directories are the same")
101         sys.exit(1)
102
103     logging.info("Setup complete")
104     return cfg, ags
105
106
107 config, args = setup()
108
109 logging.info("Starting spidercam_simulator")
110
111 input_queue = [args.input] if os.path.isfile(args.input) else os.listdir(args.input)
112
113 logging.info("Found %s files/directories to process", len(input_queue))
114
115 for file in input_queue:
116     if file == ".gitkeep":
117         continue
118
119     logging.info("Processing %s", file)
120
121     file_handler = spidercam_simulator.FileHandler(
```

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

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

    input_dict = spidercam_simulator.Parser.parse_input(contents)

    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 2: Quellcode für `/python/project_name/__main__.py`

```
from __future__ import annotations

import logging
import numpy as np

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

        self.cam_positions = []
        self.rope_lengths = []

        self.store_anchors()

    def store_anchors(self) -> None:
        """     Stores the anchor positions in regard to the dimensions

        Returns:
            None
        """

        # Anchors are at the top corners of the field
        self.anchors = [
            (0, 0, self.dim[2]),
            (self.dim[0], 0, self.dim[2]),
            (0, self.dim[1], self.dim[2]),
            (self.dim[0], self.dim[1], self.dim[2]),
        ]

    @classmethod
    def from_dict(cls, data: dict) -> "Controller":
        """     Creates a Controller object from a dictionary

        Args:
            data (dict): The dictionary to create the Controller object from
                - 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

        Returns:
            Controller: The Controller object
        """

        cls.logger = logging.getLogger(__name__)
        cls.logger.debug("Creating Controller from dict %s", data)
        return cls(**data)

    def __repr__(self) -> str:
```

```
        """     Returns a string representation of the Controller object

        Returns:
            str: The string representation of the Controller object
        """
        return f"Controller({self.dim=}, {self.spidercam.start=}, {self.spidercam.max_velocity=}, {self.spidercam.acceleration=}, {self.freq=}, {self.instructions=})"

    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)
            else:
                self.spidercam.move()

            # 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 3: Quellcode für `/python/project_name/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 = [
        "",  # check absolute
        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:
        if os.path.exists(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))
        else:
            if not instructions:
                raise ValueError("No instructions found")

        # Check if the dimensions are valid
```
```

```
        if len(dim) != 3:
            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
        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"
```

```
285
286         # output2 = "\n".join(
287         #     ",".join(map(str, line)) for line in [dim, time_stamps, *cam_positions]
288         # )
289
290         Parser.logger.debug("Parsed output")
291
292         return output1, output2
293
294
295 class Instruction:
296     """     A class for defining an instruction"""
297
298     def __init__(self, start_time: float = 0.0, destination: tuple = (0, 0, 0)) ->
    None:
299         """Initializes the Instruction class
300
301         Args:
302             start_time (float, optional): The time to start the instruction. Defaults
    to 0.0.
303             destination (tuple, optional): The destination of the instruction.
    Defaults to (0, 0, 0).
304
305         Returns:
306             None
307         """
308         self.logger = logging.getLogger(__name__)
309         self.logger.debug(
310             "Initializing Instruction with start_time=%f and destination=%s",
311             start_time,
312             destination,
313         )
314         self.start_time = start_time
315         self.destination = destination
316
317     @classmethod
318     def parse(cls, data: str) -> "Instruction":
319         """Parses an instruction from a string
320
321         Args:
322             data (str): The string to parse the instruction from
323                 - format: start_time x y z
324
325         Returns:
326             Instruction: The instruction
327         """
328         cls.logger = logging.getLogger(__name__)
329         cls.logger.debug("Parsing instruction from %s", data)
330
331         try:
332             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)))

    def __repr__(self) -> str:
        """Returns the representation of the instruction

        Args:
            None

        Returns:
            str: The representation of the instruction
        """
        return (
            f"Instruction(start_time={self.start_time}, destination={self.destination
    })"
        )
```

Listing 4: Quellcode für `/python/project_name/io.py`

```
from __future__ import annotations

import logging

import numpy as np
import spidercam_simulator


class Movement:
    """       A class for defining a movement"""

    phases: list[spidercam_simulator.Phase] = []

    def __init__(
        self,
        spidercam: spidercam_simulator.Spidercam,
        start_time: float = 0,
        start: tuple = (0, 0, 0),
        destination: tuple = (0, 0, 0),
    ) -> None:
        """    Initializes the Movement class

        Args:
            spidercam (Spidercam): The spidercam instance
            start_time (float, optional): The time to start the movement. Defaults to
    0.
            start (tuple, optional): The starting position. Defaults to (0, 0, 0).
```

```
            destination (tuple, optional): The destination position. Defaults to (0, 0, 0).
        Returns:
            None
        """
        self.logger = logging.getLogger(__name__)
        self.logger.info(
            "Initializing Movement at %s with start %s and destination %s",
            start_time,
            start,
            destination,
        )

        self.spidercam = spidercam
        self.start_time = start_time
        self.start = start
        self.destination = destination

        self.calculate_phases()

    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) -> None:
        """     Calculates the phases of the movement

        Returns:
            None
        """
        # HERE 3
```

```
        self.logger.info("Calculating phases")

        distance = np.linalg.norm(np.array(self.destination) - np.array(self.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 = (
                self.start + (np.array(self.destination) - np.array(self.start)) / 2
            )
            self.phases = [
                spidercam_simulator.Phase(
                    self,
                    spidercam_simulator.Phase.Mode.ACCELERATION,
                    self.start,
                    middle_point,
                    0.0,
                ),
                spidercam_simulator.Phase(
                    self,
                    spidercam_simulator.Phase.Mode.DECELERATION,
                    middle_point,
                    self.destination,
                    # starting velocity is the velocity at the end of the acceleration phase
                    # v = sqrt(ad)
                    np.sqrt(self.spidercam.acceleration * distance),
                ),
            ]

        else:
            # Acceleration, constant velocity and deceleration
            # HERE 4
            self.logger.debug("Acceleration, constant velocity and deceleration needed")

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

            point_b = (
                np.array(self.destination)
                - (np.array(self.destination) - np.array(self.start))
                * self.spidercam.dist_vmax
```

```
                / distance
            )

            self.phases = [
                spidercam_simulator.Phase(
                    self,
                    spidercam_simulator.Phase.Mode.ACCELERATION,
                    self.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,
                    self.destination,
                    self.spidercam.max_velocity,
                ),
            ]

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

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

        Returns:
            None
        """

        # HERE 2
        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
```

```
177
178             # Nothing to do if already decelerating
179             if phase.mode == spidercam_simulator.Phase.Mode.DECELERATION:
180                 self.logger.debug("Already decelerating, nothing to do")
181                 return
182
183             offset = time - time_sum
184
185             # Found phase should end at offset
186             phase.destination = phase.get_position(offset)
187             phase.update()
188
189             # Update next phases
190             # There always is a next phase because the final phase is deceleration and
     the previous phase was not
191             next_phase = self.phases[self.phases.index(phase) + 1]
192
193             next_phase.start = phase.destination
194             next_phase.starting_velocity = (
195                 self.spidercam.max_velocity
196                 if phase.mode is spidercam_simulator.Phase.Mode.CONSTANT_VELOCITY
197                 else self.spidercam.acceleration * offset
198             )
199             next_phase.mode = spidercam_simulator.Phase.Mode.DECELERATION
200             next_phase.destination = next_phase.get_position(
201                 next_phase.starting_velocity / self.spidercam.acceleration
202             )
203             next_phase.update()
204
205             # If there is a phase after the next phase, pop it
206             if len(self.phases) > self.phases.index(next_phase) + 1:
207                 self.phases.pop(self.phases.index(next_phase) + 1)
208
209             break
210
211         # # If there are two deceleration phases at the end, delete the second one
212         # if (
213         #     self.phases[-1].mode == spidercam_simulator.Phase.Mode.DECELERATION
214         #     and self.phases[-2].mode == spidercam_simulator.Phase.Mode.DECELERATION
215         # ):
216         #     self.phases.pop()
217
218         # Update the destination of the movement
219         self.destination = self.phases[-1].destination
220
221         # Debug: Check if destination of movement is the same as the destination of
    the last phase
222         if not np.array_equal(self.destination, self.phases[-1].destination):
223             self.logger.error(
224                 "Destination of movement is not the same as the destination of the
    last phase (%s != %s)",
```

```
                self.destination,
                self.phases[-1].destination,
            )

    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

        raise Exception("No phase found")
```

Listing 5: Quellcode für `/python/project_name/movement.py`

```python
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
```

```
49
50          self.update()
51
52      def update(self) -> None:
53          """     Updates the phase
54
55          Args:
56              None
57
58          Returns:
59              None
60          """
61          self.logger.debug("Updating phase %s", self.mode)
62
63          self.distance = self.calc_distance()
64          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
```

```
        Args:
            offset (float): The offset to get the position after

        Returns:
            tuple: The position of the phase after the given offset
        """

        self.logger.debug(
            "Getting position after %s seconds, mode %s", offset, self.mode
        )

        # if offset > self.duration:
        #     self.logger.debug(
        #         f"Offset {offset} is greater than duration {self.duration},
    returning destination {repr(self.destination)}"
        #     )
        #     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 6: Quellcode für `/python/project_name/phase.py`

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

```python


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
            plt.savefig(
                f"{self.output_dir}/{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()

        # Plot rope lengths as a function of time (discrete)
        fig = plt.figure()

        # Set axis labels
        plt.xlabel("Time (s)")
        plt.ylabel("Rope Length (m)")

        # Set axis limits
        plt.xlim(0, len(self.rope_lengths) / self.freq)
        plt.ylim(0, np.max(self.rope_lengths) * 1.1)

        # Plot rope lengths
        plt.plot(
            np.arange(0, len(self.rope_lengths) / self.freq, 1 / self.freq),
            self.rope_lengths,
        )

        # Legend of Rope i
        plt.legend([f"Rope {i}" for i in range(len(self.rope_lengths[0]))])

        # Set title for whole figure
        # fig.suptitle("Rope Lengths for " + self.name)

        # Save or show plot
        if self.output_dir is None:
            plt.show()
        else:
            # Save at output + name + rope_lengths.png
            plt.savefig(
                f"{self.output_dir}/{self.name}_rope_lengths.png",
                bbox_inches="tight",
                dpi=300,
            )

    def plot(self) -> None:
        """     Plots the camera positions and rope lengths
```

```
158
159         Returns:
160             None
161         """
162
163         self.logger.info("Plotting camera positions and rope lengths for %s", self.
    name)
164
165         self.plot_cam_positions()
166         self.plot_rope_lengths()
```

Listing 7: Quellcode für `/python/project_name/plotter.py`

```
1  from __future__ import annotations
2
3  import logging
4
5  import spidercam_simulator
6
7
8  class Spidercam:
9      """        A class for controlling the spidercam"""
10
11     movements: list[spidercam_simulator.Movement] = []
12     queue: spidercam_simulator.Movement = None
13
14     def __init__(
15         self,
16         controller: spidercam_simulator.Controller,
17         max_velocity: float = 1.0,
18         acceleration: float = 1.0,
19         start: tuple = (0, 0, 0),
20     ) -> None:
21         """    Initializes the Spidercam class
22
23         Args:
24             controller (Controller): The controller instance
25             max_velocity (float, optional): The maximum velocity of the spidercam.
    Defaults to 1.0.
26             acceleration (float, optional): The maximum acceleration of the spidercam.
     Defaults to 1.0.
27             start (tuple, optional): The starting position of the spidercam. Defaults
    to (0, 0, 0).
28
29         Returns:
30             None
31         """
32
33         self.logger = logging.getLogger(__name__)
34         self.logger.debug(
35             "Initializing Spidercam with max_velocity=%s, acceleration=%s, start=%s",
36             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 constants for the spidercam

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

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

        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")
```

```
85              return
86
87          # No instruction given, update the current movement
88          if instruction is None and time != -1.0:
89              self.logger.debug("Updating movement at time %s", time)
90
91              # If there is no movement in the queue, return
92              if self.queue is None:
93                  self.logger.debug("No movement in queue, returning")
94                  return
95
96              # If the last movement is finished, add the movement in the queue to the
    movements list
97              if self.movements[-1].end_time() <= time:
98                  self.logger.debug(
99                      "Last movement is finished, adding queue to movements list"
100                 )
101                 self.movements.append(self.queue)
102                 self.queue = None
103                 return
104
105             return
106
107         # Instruction given, apply the instruction
108         self.logger.debug("Moving spidercam with instruction %s", instruction)
109
110         # First movement is always possible
111         if len(self.movements) == 0:
112             self.logger.debug("First movement registered")
113
114             self.movements.append(
115                 spidercam_simulator.Movement(
116                     self, instruction.start_time, self.start, instruction.destination
117                 )
118             )
119             return
120
121         self.logger.debug("Checking if last movement is finished")
122         self.logger.debug("Last movement: %s", self.movements[-1])
123
124         # HERE 1
125         # If the last movement is not finished, decelerate and update the queue
126         if self.movements[-1].end_time() > instruction.start_time:
127             self.logger.debug("Last movement is not yet finished")
128
129             self.logger.debug("Queueing instruction %s", instruction)
130
131             # Decelerate
132             self.movements[-1].start_deceleration(instruction.start_time)
133
134             # Overwrite queue
```

```
            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")

            self.movements.append(self.queue)
            self.queue = None

            # Move to the destination after the queue
            self.move(instruction)
            return

        # If the last movement is finished and there is no queue, move to the
    destination
        self.logger.debug("Last movement is finished and there is no queue")

        self.movements.append(
            spidercam_simulator.Movement(
                self,
                max(self.movements[-1].end_time(), instruction.start_time),
                self.movements[-1].destination,
                instruction.destination,
            )
        )

    def get_position(self, time: float) -> tuple:
        """     Gets the position of the spidercam at a given time

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

        Returns:
            tuple: The position of the spidercam at the given time
        """

        self.logger.debug("Getting position at time %s", time)

        self.move(time=time)

        # If there are no movements, return the start position
        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"
197             )
198             return self.movements[-1].destination
199 
200         # If the time is during a movement, return the position of that movement
201         for movement in self.movements:
202             if movement.start_time <= time <= movement.end_time():
203                 self.logger.debug(
204                     "Time is during movement %s, returning position of movement",
205                     movement,
206                 )
207                 return movement.get_position(time)
208 
209         # If the time is between movements, return the destination of the previous
    movement
210         for i in range(len(self.movements) - 1):
211             if self.movements[i].end_time() < time < self.movements[i + 1].start_time:
212                 self.logger.debug(
213                     "Time is between movements %s and %s, returning destination of
    previous movement",
214                     self.movements[i],
215                     self.movements[i + 1],
216                 )
217                 return self.movements[i].destination
218 
219         # If the time is not in any of the above cases, return the start position
220         self.logger.debug(
221             "Time is not in any of the above cases, returning start position"
222         )
```

Listing 8: Quellcode für `/python/project_name/spidercam.py`