generate_trajectory.py

```python
# Main-Script
# ##############################################################################

# This python script can be used to generate a trajectory with constrains from
# a floorplan or similar geometry.

# Authors:
# Christopher Mahn
# Silas Teske
# Joshua Wolf

# ##############################################################################

# Import of Libraries
# ------------------------------------------------------------------------------

import main as settings
# import string as st
import random as r
import time
# import re
# from turtle import position
# from scipy import interpolate
from concurrent.futures import process
from turtle import position
import numpy as np
import math as m
# import sys
import os
import matplotlib.pyplot as plt
# from scipy.fft import fft, fftfreq
# from scipy import signal
import multiprocessing as mp
import copy
import lib_trajectory as t

# ------------------------------------------------------------------------------
# Debugging-Settings

verbose = True  # Shows more debugging information

# Functions
# ------------------------------------------------------------------------------

def __help_generate_trajectory(trajectory):
    trajectory.generate()
    return(trajectory.get(), trajectory.get_garbage())


def create_multiple_trajectories(trajectory, ammount):
    """
    This function takes one trajectory-object with specified parameters and
    generates multiple trajectories from it.

    Args:
        trajectory (trajectory-object): Trajectory-object with specified parameters
        ammount (int): Ammount of trajectories to be generated
    """
    trajectory_objects = []

    # Making multiple copies of the trajectory-object
    for i in range(ammount):
        if(verbose):
            print(f'[INFO][1/2][{i+1}/{ammount}] Copying trajectory', end="\r")
        trajectory_objects.append(copy.deepcopy(trajectory))

    # Processing the generation and retrieval of trajectories in parallel
    if(verbose):
        print(f'[INFO][PARALLEL][2/2] Generating {ammount} trajectories')
        if(ammount >= 10):
            print(f'[WARN] This might take some time to process...')
        elif(ammount >= 50):
            print(f'[DANGER] This might be extremely long to process...')
    processing = mp.Pool()
    trajectories = processing.map(__help_generate_trajectory, trajectory_objects)
    return(trajectories)


def __help_plot_lines_rgb(input):
```

```
080     t.plot_lines_rgb(lines_red=input["red"], lines_green=input["green"], lines_blue=input["blue"],
          title=input["title"], filename=input["filename"])
081     return(None)
082
083
084 def plot_lines_rgb_multiple(filenames, lines_red=None, lines_green=None, lines_blue=None, titles=None):
085     if(verbose):
086         print(f'[INFO][PARALLEL] Plotting {len(filenames)} graphs')
087     tasks = []
088     for i, filename in enumerate(filenames):
089         red = None
090         green = None
091         blue = None
092         title = "Line-segments"
093         if(lines_red != None):
094             red = lines_red[i]
095         if(lines_green != None):
096             green = lines_green[i]
097         if(lines_blue != None):
098             blue = lines_blue[i]
099         if(titles != None):
100             title = titles[i]
101         tasks.append({"red": red, "green": green, "blue": blue, "title":title, "filename": filename})
102     processing = mp.Pool()
103     processing.map(__help_plot_lines_rgb, tasks)
104     return(None)
105
106
107 # Classes
108 # ----------------------------------------------------------------------------
109
110 class Trajectory():
111     def __init__(self,
112                  position_init={"x": 0.0, "y": 0.0},
113                  direction_init=0,
114                  direction_step_noise=10/180*m.pi,
115                  direction_try_noise_add=5/180*m.pi,
116                  length_total=250,
117                  length_step=0.8,
118                  length_step_noise=0.15,
119                  geometry=[],
120                  trajectory=[],
121                  not_trajectory=[],
122                  tries=5,
123                  check_length_buffer=0.4,
124                  check_width_buffer=0.3):
125         self.position_init = position_init
126         # This is the initial position, where the trajectories are generated
127         # from
128
129         self.direction_init = direction_init
130         # This is the initial direction the trajectory starts from
131
132         self.direction_step_noise = direction_step_noise
133         # This is the amount of bending, that occurs as a base angle-change.
134
135         self.direction_try_noise_add = direction_try_noise_add
136         # This is the amount of bending, that get's applied for areas with
137         # higher generation-difficulty.
138
139         self.length_total = length_total
140         # length of the trajectory in footsteps
141
142         self.length_step = length_step
143         # length of the average footstep in meters
144
145         self.length_step_noise = length_step_noise
146         # standard-deviation of the average footstep
147
148         self.geometry = geometry
149         # This is a list of Line-Objects, that form a complex boundary for the
150         # trajectory
151
152         self.trajectory = trajectory
153         # This is a list, which will contain the generated trajectory
154         # consisting of line segments.
155
156         self.not_trajectory = not_trajectory
157         # This is a list with all Line-segments, that were discarded in the
158         # trajectory-generation.
159
160         self.tries = tries
161         # This variable sets the number of tries a step will be generated,
```

```
162         # before stepping back one step recursively.
163 
164         self.check_length_buffer = check_length_buffer
165         # This variable controls the behaviour for the trajectory to avoid
166         # direct wall-contact. The step will be extended by this amount in
167         # meters before checking intersection with walls.
168 
169         self.check_width_buffer = check_width_buffer
170         # This variable controls the behaviour for the trajectory to avoid
171         # direct wall-contact. The step will be widened by this amount in
172         # meters before checking intersection with walls.
173 
174     def __copy__(self):
175         return(type(self)(self.position_init,
176                           self.direction_init,
177                           self.direction_step_noise,
178                           self.direction_try_noise_add,
179                           self.length_total,
180                           self.length_step,
181                           self.length_step_noise,
182                           self.geometry,
183                           self.trajectory,
184                           self.not_trajectory,
185                           self.tries,
186                           self.check_length_buffer,
187                           self.check_width_buffer))
188 
189     def __deepcopy__(self, memo):  # I don't know what memo is for, but it's
190                                    # something with recursive copy or something
191                                    # similar
192         return(type(self)(
193                copy.deepcopy(self.position_init, memo),
194                copy.deepcopy(self.direction_init, memo),
195                copy.deepcopy(self.direction_step_noise, memo),
196                copy.deepcopy(self.direction_try_noise_add, memo),
197                copy.deepcopy(self.length_total, memo),
198                copy.deepcopy(self.length_step, memo),
199                copy.deepcopy(self.length_step_noise, memo),
200                copy.deepcopy(self.geometry, memo),
201                copy.deepcopy(self.trajectory, memo),
202                copy.deepcopy(self.not_trajectory, memo),
203                copy.deepcopy(self.tries, memo),
204                copy.deepcopy(self.check_length_buffer, memo),
205                copy.deepcopy(self.check_width_buffer, memo)))
206 
207     def export(self):
208         return(self.position_init,
209                self.direction_init,
210                self.direction_step_noise,
211                self.direction_try_noise_add,
212                self.length_total,
213                self.length_step,
214                self.length_step_noise,
215                self.geometry,
216                self.trajectory,
217                self.not_trajectory,
218                self.tries,
219                self.check_length_buffer,
220                self.check_width_buffer)
221 
222     def set_start_coordinate(self, x, y):
223         self.position_init["x"] = float(x)
224         self.position_init["y"] = float(y)
225 
226     def set_start_direction(self, direction):
227         self.direction_init = float(direction)
228 
229     def set_direction_turn_noise(self, mdev):
230         self.direction_step_noise = float(mdev)
231 
232     def set_trajectory_length(self, footsteps):
233         self.length_total = float(footsteps)
234 
235     def set_median_step_size(self, length):
236         self.length_step = float(length)
237 
238     def set_step_size_noise(self, mdev):
239         self.length_step_noise = float(mdev)
240 
241     def set_geometry(self, line_segments):
242         self.geometry = line_segments
243 
244     def __check_intersection(self, line):
```

```
        """
        This method checks if a line is intersecting with some of the internal geometry.

        Args:
            line (Line-Object): This line will be checked against the geometry.
        """
        intersection = False
        point1 = {"x": line.x1(), "y": line.y1()}
        point2 = {"x": line.x2(), "y": line.y2()}
        for i in self.geometry:
            point3 = {"x": i.x1(), "y": i.y1()}
            point4 = {"x": i.x2(), "y": i.y2()}
            try:
                intersected_point = t.geradenschnitt(point1, point2, point3, point4)
                if(point1["x"] <= intersected_point["x"] <= point2["x"] and point3["x"] <= intersected_point["x"] <= point4["x"]):
                    intersection = True
                elif(point1["x"] >= intersected_point["x"] >= point2["x"] and point3["x"] <= intersected_point["x"] <= point4["x"]):
                    intersection = True
                elif(point1["x"] <= intersected_point["x"] <= point2["x"] and point3["x"] >= intersected_point["x"] >= point4["x"]):
                    intersection = True
                elif(point1["x"] >= intersected_point["x"] >= point2["x"] and point3["x"] >= intersected_point["x"] >= point4["x"]):
                    intersection = True
            except(Exception):
                pass
                # if(verbose):
                #     print(f'[WARN] intersection could not be calculated')
            if(intersection):
                break
        return(intersection)

    def generate_legacy(self):
        """
        This method generates the trajectory using a iterative aproach.
        """
        direction = self.direction_init
        position = self.position_init
        self.trajectory = []
        for i in range(self.length_total):
            tries = 0
            direction_try = direction
            while(tries < 100):
                # print(f'{tries}    ', end="\r")
                direction_try += np.random.normal(0, self.direction_step_noise)
                length_try = self.length_step + np.random.normal(0, self.length_step_noise)
                position_try = {"x": None, "y": None}
                position_try["x"] = position["x"] + (m.cos(direction_try) * length_try)
                position_try["y"] = position["y"] + (m.sin(direction_try) * length_try)
                line_try = t.Line(position["x"], position["y"], position_try["x"], position_try["y"])
                if(self.__check_intersection(line_try)):
                    tries += 1
                else:
                    self.trajectory.append(line_try)
                    direction = direction_try
                    position = position_try
                    break

    def generate(self):
        """
        This function generates the trajectory recursively.
        """
        direction = [self.direction_init]
        position = [self.position_init]
        self.trajectory = []
        tries = [0]
        while(len(position) <= self.length_total):
            if(verbose):
                # print(f'[INFO][{len(position)+1}/{self.length_total}] Generating trajectory ', end="\r")
                pass
            while(tries[-1] < self.tries):
                tries[-1] += 1
                direction_try = np.random.normal(direction[-1], (self.direction_step_noise + (tries[-1] * self.direction_try_noise_add)))
                length_try = np.random.normal(self.length_step, self.length_step_noise)

                # Line-segment for the trajectory
                position_try = {"x": None, "y": None}
                position_try["x"] = position[-1]["x"] + (m.cos(direction_try) * length_try)
                position_try["y"] = position[-1]["y"] + (m.sin(direction_try) * length_try)
```

```
322                 line_try = t.Line(position[-1]["x"], position[-1]["y"], position_try["x"],
        position_try["y"])
323
324                 # Line-segments for the Intersection-Check
325                 position_check1_1 = {"x": None, "y": None}
326                 position_check1_2 = {"x": None, "y": None}
327                 position_check2_1 = {"x": None, "y": None}
328                 position_check2_2 = {"x": None, "y": None}
329                 position_check1_1["x"] = position[-1]["x"] + (m.cos(direction_try - (90/180*m.pi)) *
        (self.check_width_buffer))
330                 position_check1_1["y"] = position[-1]["y"] + (m.sin(direction_try - (90/180*m.pi)) *
        (self.check_width_buffer))
331                 position_check1_2["x"] = position_check1_1["x"] + (m.cos(direction_try) * (length_try +
        self.check_length_buffer))
332                 position_check1_2["y"] = position_check1_1["y"] + (m.sin(direction_try) * (length_try +
        self.check_length_buffer))
333                 position_check2_1["x"] = position[-1]["x"] + (m.cos(direction_try + (90/180*m.pi)) *
        (self.check_width_buffer))
334                 position_check2_1["y"] = position[-1]["y"] + (m.sin(direction_try + (90/180*m.pi)) *
        (self.check_width_buffer))
335                 position_check2_2["x"] = position_check2_1["x"] + (m.cos(direction_try) * (length_try +
        self.check_length_buffer))
336                 position_check2_2["y"] = position_check2_1["y"] + (m.sin(direction_try) * (length_try +
        self.check_length_buffer))
337                 line_check_1 = t.Line(position_check1_1["x"], position_check1_1["y"],
        position_check1_2["x"], position_check1_2["y"])
338                 line_check_2 = t.Line(position_check2_1["x"], position_check2_1["y"],
        position_check2_2["x"], position_check2_2["y"])
339                 line_check_3 = t.Line(position_check1_1["x"], position_check1_1["y"],
        position_check2_1["x"], position_check2_1["y"])
340                 line_check_4 = t.Line(position_check1_1["x"], position_check1_1["y"],
        position_check2_2["x"], position_check2_2["y"])
341
342                 # Intersection-Check and Saving if Line-Segments
343                 if(self.__check_intersection(line_check_1) or
344                    self.__check_intersection(line_check_2) or
345                    self.__check_intersection(line_check_3) or
346                    self.__check_intersection(line_check_4)):
347                     pass
348                 else:
349                     direction.append(direction_try)
350                     position.append(position_try)
351                     tries.append(0)
352                     self.trajectory.append(line_try)
353                     if(len(self.trajectory) > self.length_total):
354                         break
355             direction.pop(-1)
356             position.pop(-1)
357             tries.pop(-1)
358             self.not_trajectory.append(self.trajectory.pop(-1))
359
360     def get(self):
361         # This method returns the trajectory
362         return(self.trajectory)
363
364     def get_garbage(self):
365         # This method returns the line-segments, that failed to generate
366         return(self.not_trajectory)
367
368 # Beginning of the Programm
369 # ----------------------------------------------------------------------------
370
371 if __name__ == '__main__':
372
373     # Setting starting values
374     start_positions = settings.project_start_positions
375     start_directions = settings.project_start_directions
376
377     # Collecting all plot-data for later parallel processing
378     plots = {"filenames": [], "lines_red": [], "lines_green": [], "lines_blue": [], "titles": []}
379
380     # Iterating over all projects, do ...
381     for index, projectname in enumerate(settings.project_filenames):
382
383         # Import floorplan
384         floorplan = t.lines_import(f'{projectname}_lines.csv')
385
386         # Set up the trajectory
387         trajectory = Trajectory()
388         trajectory.set_geometry(floorplan)
389         trajectory.set_start_coordinate(start_positions[index]["x"], start_positions[index]["y"])
390         trajectory.set_start_direction(start_directions[index])
391
```

```
392         # Creating the trajectories
393         data = create_multiple_trajectories(trajectory, settings.trajectories_per_project)
394 
395         # Saving the data
396         trajectories = []
397         not_trajectories = []
398         for i in data:
399             trajectories.append(i[0])
400             not_trajectories.append(i[1])
401         del i
402 
403         # Iterating over the individual trajectories, do ...
404         for trajectory_index, current_trajectory in enumerate(trajectories):
405 
406             # Export the trajectory as .csv-file
407             t.lines_export(current_trajectory,
    f'trajectory_{projectname}_{trajectory_index+1:05d}_ground-truth.csv')
408 
409             # Create Plot 1
410             plots["filenames"].append(f'trajectory_{projectname}_{trajectory_index+1:05d}')
411             plots["titles"].append(f'Trajectory with floorplan "{projectname}"')
412             plots["lines_red"].append(None)
413             plots["lines_green"].append(current_trajectory)
414             plots["lines_blue"].append(floorplan)
415 
416             # Create Plot 2
417 
    plots["filenames"].append(f'trajectory_{projectname}_{trajectory_index+1:05d}_with_disregarded_li
    nes')
418             plots["titles"].append(f'Trajectory with floorplan "{projectname}" and discarded Line-
    segments')
419             plots["lines_red"].append(not_trajectories[trajectory_index])
420             plots["lines_green"].append(current_trajectory)
421             plots["lines_blue"].append(floorplan)
422 
423     # Plotting everything in parallel
424     plot_lines_rgb_multiple(filenames=plots["filenames"],
425                             lines_red=plots["lines_red"],
426                             lines_green=plots["lines_green"],
427                             lines_blue=plots["lines_blue"],
428                             titles=plots["titles"])
```