Assignment 2B

Rime-scaling non-holonomic trajectories

EC4.403 - Robotics: Planning and Navigation

Avneesh Mishra

avneesh.mishra@research.iiit.ac.in *

May 1, 2022

Contents

1 Constant time scaling **2**

1.1 Rule-based Constant time-scaling 2

1.2 Collision cone based constant time-scaling 3

2 Linear time scaling **6**

2.1 Rule-based linear time scaling 6

3 Theory **7**

3.1 Smooth trajectory 7

3.2 MPC Parallel 7

3.3 Multi-robot 7

A Rule-based Constant Time Scaling **8**

A.1 Independent variable time-scaling 8

A.2 Non-Holonomic Robot 10

B Collision Cone Constant Time Scaling **13**

C Rule-based Linear Time Scaling **18**

List of Figures

1 Time scaling 2

2 Rule-based constant time-scaling 3

3 Collision Cone 4

4 Collision cone based constant time scaling 5

5 Collision cone - Constant time scaling - Video snaps 5

6 Rule-based Linear time scaling 6

7 Rule-based - Linear time scaling - Video snaps 6

*M.S. by research - CSE, IIIT Hyderabad, Roll No: 2021701032

[1]Videos and results available at https://iiitaphyd-my.sharepoint.com/:f:/g/personal/avneesh_mishra_research_iiit_ac_in/Er_wRqK4hxVLjVdL56rfDxYBKr9PPed1laN48hLgLisf4w

1 Constant time scaling

In general, we time-scale the velocities to reach the same point but at a later time. This causes us to avoid collisions *while* not deviating from the desired path. Constant time scaling is implemented using the equation below

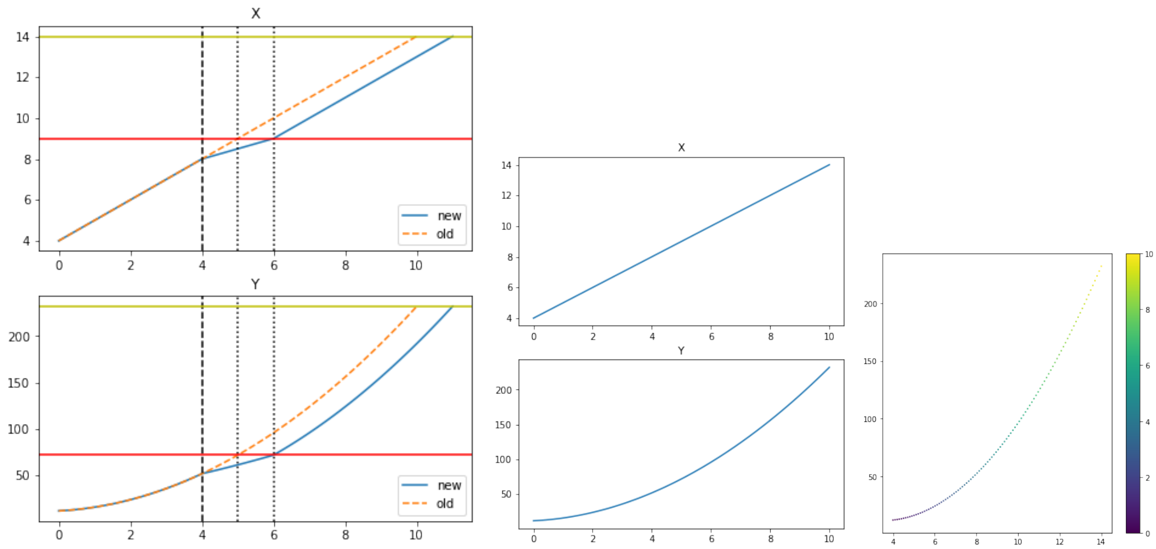
$$\dot{x}(\tau) = k\,\dot{x}(t) \tag{1.1}$$

1.1 Rule-based Constant time-scaling

Say we have a time expression (function) $x(t)$. We first obtain $\dot{x}(t)$ by taking the time derivative $\dot{x}(t) = {}^{dx}/_{dt}$. We then obtain the time range within which this time scaling trick has to be applied. This could be obtained through a sensor. Say this time range is t_{c1} and t_{c2} (in the original time frame t).

We then find the new time values in the scaled time frame τ such that they are increased (farther apart), while the velocity $\dot{x}$ between them reduces. This is both related by the same constant k.

After scaling the velocity down by k and scaling time t_{c2} up by k, we resume the original velocity (as it was after the original t_{c2}). This marks the end of time scaling. However, the entire process will now end later (since the time between t_{c1} and t_{c2} was elongated). This concept is demonstrated in figure 1.



(a) Time scaling (b) Original (c) XY

Figure 1: Time scaling
The variables X and Y are functions of time. The time scaling is applied from 4 to 5 seconds, with $k = 0.5$ (the new end of time scaling will therefore happen at 6 seconds).
In left figure, it is apparent that the velocities have decreased to half in the time scaling period, while the duration has doubled.
The center and right figures show the original trajectory (X and Y as function of time in center, and X vs. Y plot in the right).

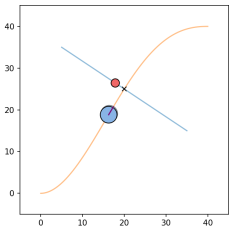
An example code implementing and testing this is available in appendix A.1. However, note that the code there implements time scaling on independent functions of time, and we are dealing with a constrained (non-holonomic) system.

We get around this by modelling the angle as $\theta = \operatorname{atan2}(\dot{y}, \dot{x})$ along the entire new (time scaled) trajectory. The code for this is available in appendix A.2. The results from a simulation run are shown in figure 2.

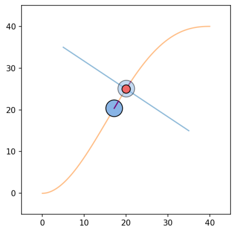
Problems with Rule-based CTS

The problems with *rule-based* constant time scaling are mentioned below

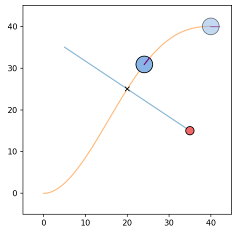
- The velocity profile, though being continuous, is **not** smooth. This leads to *discontinuity in acceleration*, and this occurs at two places (starting and ending of time scaling). This leads to jerks



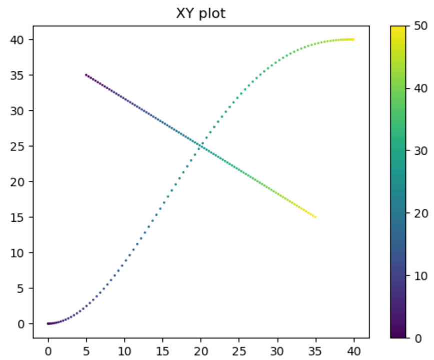
(a) Start



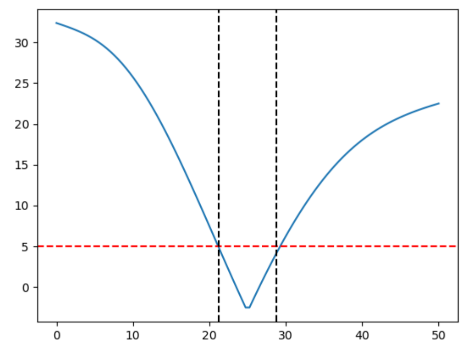
(b) Collision avoidance



(c) End



(d) XY (time colormap)



(e) Distance Rule

Figure 2: Rule-based constant time-scaling
The simulation is available as `rb_cts_exp1.avi` in the shared OneDrive folder. For figure d, we see `XY` plot (colormap has time encoded). For figure e, the distance is the line in blue (as time goes). The horizontal red line is a distance threshold. The two vertical lines indicate the start and end of time scaling (manually configured).

in the motion. We could apply a variable time-scaling which smoothly varies the velocity profile, while keeping us on track.

- We could be *wasting time* even after the collision point has passed. This can be mitigated by using a half-time window (than the actual one) from start to end. However, this assumes that the convergence and divergence patterns of the obstacle and robot are symmetric. If the obstacle takes long to diverge, then this could be risky.

1.2 Collision cone based constant time-scaling

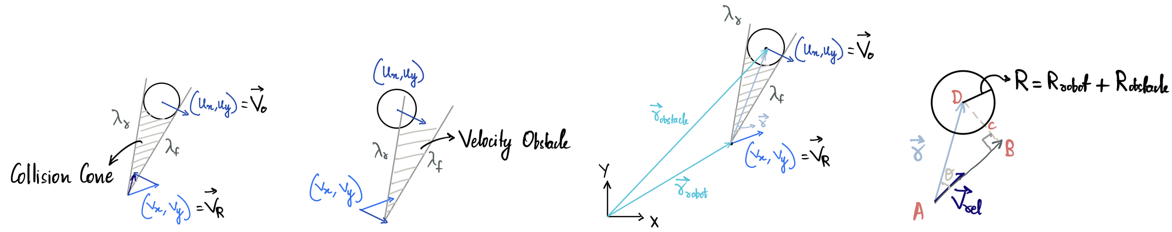
A collision cone is comprised of the relative velocities of the robot (with reference to an obstacle) that will lead to a collision in the future. Usually, we assume a linear velocity trajectory (for both robot and obstacle). However, the method can be generalized to any *robot* profile (say a bernstein profile) using an iterative approach (as we do ahead).

The collision cone is shown in figure 3a. The *velocity obstacle* is obtained by adding the obstacle velocity vector to the collision cone (as shown in figure 3b). The robot's velocity should be out of the velocity obstacle.

Consider the environment with the collision cone (as in figure 3c). By looking into a zoomed version of it (as in figure 3d - use this for variables in the equations), we can formulate the following to avoid a collision

$$\mathrm{DB} \geq \mathrm{DC} \Rightarrow \mathrm{DB}^2 \geq R^2 \Rightarrow \mathrm{AD}^2 - \mathrm{AB}^2 \geq R^2 \qquad \vec{r}_{obstacle} = \vec{r}_{robot} + \vec{r} \Rightarrow \vec{r} = \vec{r}_{obstacle} - \vec{r}_{robot}$$

$$\mathrm{AB} = \|\vec{r}\| \cos(\theta) = \frac{\vec{V}_{rel} \cdot \vec{r}}{\left\|\vec{V}_{rel}\right\|} \Rightarrow \mathrm{AB}^2 = \left[\frac{\vec{V}_{rel} \cdot \vec{r}}{\left\|\vec{V}_{rel}\right\|}\right]^2 \qquad \mathrm{AD} = \|\vec{r}\| \Rightarrow \mathrm{AD}^2 = \|\vec{r}\|^2$$



(a) Collision Cone (b) Velocity Obstacle (c) Environment (d) Avoiding collision

Figure 3: Collision Cone

The velocity of obstacle is $\overrightarrow{V}_O = (u_x, u_y)$. The velocity of obstacle is $\overrightarrow{V}_R = (v_x, v_y)$. THe radius of robot is R_{robot} and the radius of obstacle is $R_{obstacle}$. The obstacle is diluted to $R = R_{robot} + R_{obstacle}$ while the robot is reduced to a point.
After time scaling, the velocity of the robot becomes scaled by a factor of s, that is, the time scaled robot velocity is $\vec{V}_R = (s\dot{x}_1, s\dot{y}_1)$.

$$\vec{V}_{rel} = \vec{V}_R - \vec{V}_O = (s\dot{x}_1 - \dot{x}_2, s\dot{y}_1 - \dot{y}_2) \qquad \vec{r} = \vec{r}_{obstacle} - \vec{r}_{robot} = (x_2 - x_1, y_2 - y_1)$$

$$\text{AD}^2 - \text{AB}^2 \geq R^2 \Rightarrow \|\vec{r}\|^2 - \left[\frac{\vec{V}_{rel} \cdot \vec{r}}{\left\|\vec{V}_{rel}\right\|}\right]^2 \geq R^2 \qquad \|\vec{r}\|^2 - \left[\frac{\vec{V}_{rel} \cdot \vec{r}}{\left\|\vec{V}_{rel}\right\|}\right]^2 - R^2 \geq 0$$

$$\vec{V}_{rel} \cdot \vec{r} = (s\dot{x}_1 - \dot{x}_2)(x_2 - x_1) + (s\dot{y}_1 - \dot{y}_2)(y_2 - y_1) \qquad \left\|\vec{V}_{rel}\right\| = \sqrt{(s\dot{x}_1 - \dot{x}_2)^2 + (s\dot{y}_1 - \dot{y}_2)^2}$$

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 - R^2 - \frac{\left((s\dot{x}_1 - \dot{x}_2)(x_2 - x_1) + (s\dot{y}_1 - \dot{y}_2)(y_2 - y_1)\right)^2}{(s\dot{x}_1 - \dot{x}_2)^2 + (s\dot{y}_1 - \dot{y}_2)^2} \geq 0 \tag{1.2}$$

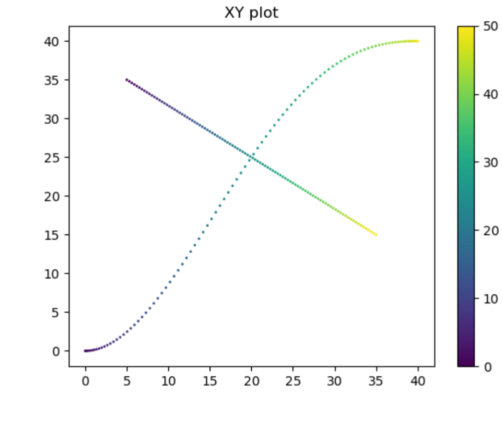
We represent equation 1.2 in the form $as^2 + bs + c \geq 0$. This gives us the following solution space (set) for s

$$S_{sol} = \begin{cases} [s_{min}, \infty) \cap ((-\infty, \gamma_1] \cup [\gamma_2, \infty)) & a > 0,\ d > 0 \\ [s_{min}, \infty) & a > 0,\ d < 0 \\ [s_{min}, \infty) \cap [\gamma_1, \gamma_2] & a < 0,\ d > 0 \\ \phi & a < 0,\ d < 0 \end{cases} \tag{1.3}$$

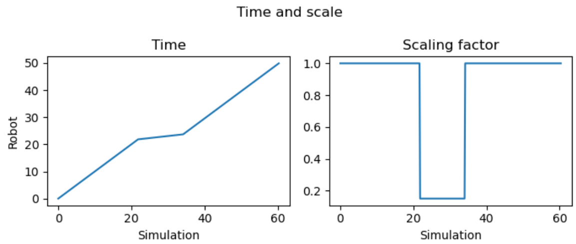
Where

$$d = b^2 - 4ac \qquad \gamma_1 = \frac{-b - \sqrt{d}}{2a} \qquad \gamma_2 = \frac{-b + \sqrt{d}}{2a}$$

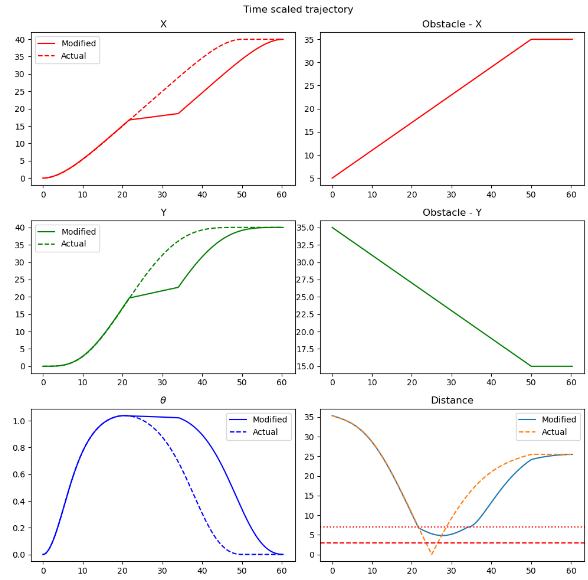
And $s_{min} = 0.1$ (minimum scaling factor). We pick $s = \min\{S_{sol}\}$ (minimum of the solution space).
Usually, equation 1.3 is accompanied with acceleration constraints of the vehicle. The code for this experiment uses `sympy` to find this quadratic equation's coefficients a, b, and c. The code is available in appendix B.
The graphs for the runs are shown in figure 4, and snaps from the simulation are shown in figure 5. You can see that here, extra time is not wasted after avoiding collision (thanks to automatic sensing of distances and choice of the scaling factor).



(a) XY plots

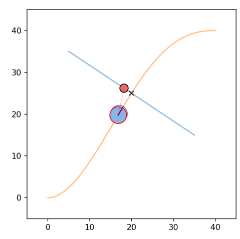


(b) Time scaling

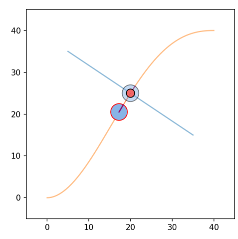


(c) Time scaled results

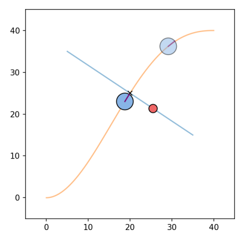
Figure 4: Collision cone based constant time scaling
The distance plot is shown in the bottom right of c. As seen, the time scaling gets activated at the thin red horizontal line (detection distance) and a collision is avoided (the distance plot does not cross the thick horizontal red line). See the slope and the scaling factor change in b. The original (colliding) trajectories are shown in a.



(a) Start



(b) Collision avoidance



(c) End

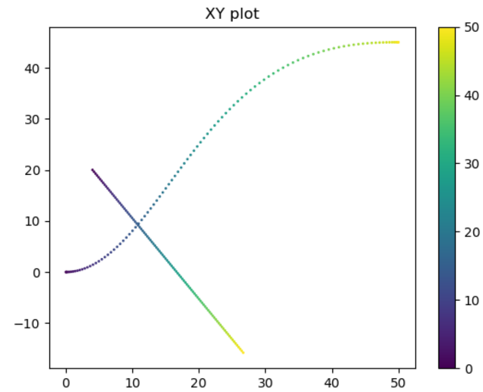
Figure 5: Collision cone - Constant time scaling - Video snaps
The simulation is available as `cc_cts_exp1.avi` in the shared OneDrive folder. The time scaling turns off here when the obstacle passes, unlike in figure 2 (where everything is hardcoded).
Another experiment is present as `cc_cts_exp2.avi` in the same shared folder.

2 Linear time scaling

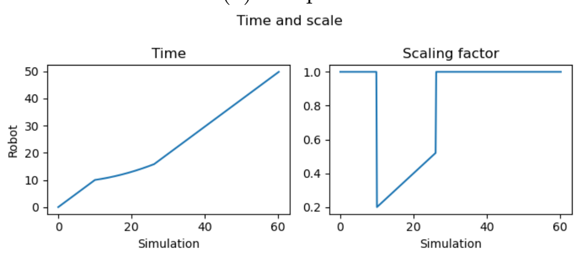
2.1 Rule-based linear time scaling

We use the scaling factor as a function of the simulation time, given by $s(t) = a + bt$. The scaling is done as in constant time scaling case. The results are shown in figure 6. The snippets from simulation are shown in figure 7.

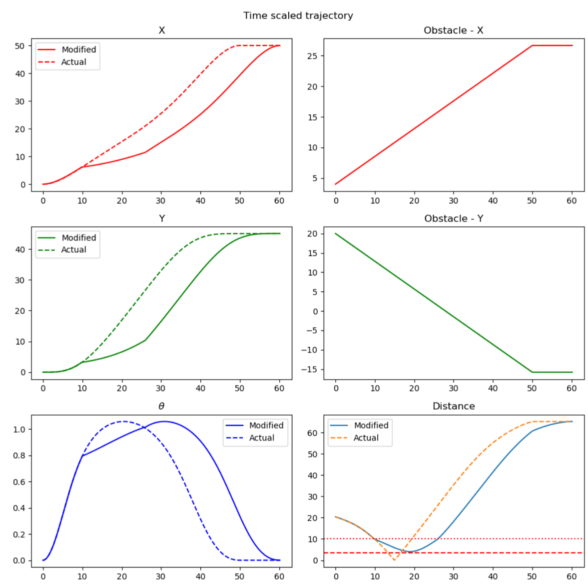
It is interesting to note that the scaling factor plot (figure 6b) shows a straight line in between the scaling times, while the time plot (on its left) shown a parabolic joint (because of accelerating time, as change is linear in time). The code to generate these figures is given in appendix C.



(a) XY plots

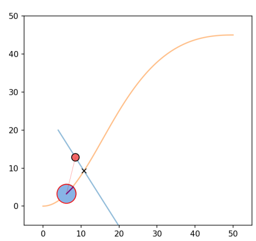


(b) Time scaling

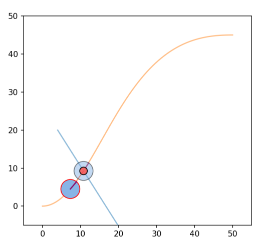


(c) Time scaled results

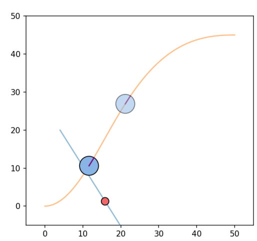
Figure 6: Rule-based Linear time scaling
The distance plot is shown in the bottom right of c. As seen, the time scaling gets activated at the thin red horizontal line (detection distance) and a collision is avoided (the distance plot does not cross the thick horizontal red line). See the slope and the scaling factor change in b (notice the linear slope, with time as parabolic). The original (colliding) trajectories are shown in a.



(a) Start



(b) Collision avoidance



(c) End

Figure 7: Rule-based - Linear time scaling - Video snaps
The simulation is available as `rb_lts_exp1.avi` in the shared OneDrive folder.

3 Theory

3.1 Smooth trajectory

As we have seen, using iterative collision cone (in case of constant) and linear time scaling techniques did not completely fix the discontinuity problem. The following could be tried

- Incorporate the acceleration and velocity constraints on the actuators. This way, even if the controller gives a time scaled velocity, the actuator will clip it towards the limits.

  However, this could be dangerous and could lead to collisions if the frequency of control loop isn't high enough or the constraints are too restrictive.

- Trajectories can be smoothened by incorporating information of higher derivatives in the time scaling problem.

- The entire problem could be converted into a constrained optimization problem, enforcing dynamic constraints.

- Some proposals include interleaving time-scaling with MPC [The+19], or using non-linear time scaling [SK13a] can also be used.

3.2 MPC Parallel

Trajectories given by simple time-scaling (like what's implemented here) may not be very smooth, whereas trajectories from the MPC will be smooth (because it's from an optimizer using many more constraints on motion).

For the MPC to avoid collisions *specifically* using time-scaling, you could add the time-scaling equations (like equation 1.2) as additional solver constraints and incorporate the scaling factor (or scaled velocities) in the system state (unknown variables). Otherwise, the MPC will deviate from the planned trajectory (to avoid the obstacle) whereas time-scaling will stay on the trajectory.

3.3 Multi-robot

Time scaling can be extended to multiple robots using an intersection space of multiple inequalities (as presented in [SK13b]). The solution space can be decomposed into multiple conditions (as done in 1.3, but using different a_i - one for each robot). Linear programming approaches can also solve such problems.

Using velocity obstacle [FS98] is also a feasible solution (single robot and multiple obstacle case). The problem will more or less remain the same.

For any two robot case, the velocity vectors have to have sufficient deviation. If they're parallel or antiparallel, then time-scaling will not give a viable solution (it'll lead to collision or both remaining stationary). In such cases, path will have to be altered.

References

[FS98] Paolo Fiorini and Zvi Shiller. "Motion Planning in Dynamic Environments Using Velocity Obstacles". In: *The International Journal of Robotics Research* 17.7 (1998), pp. 760–772. DOI: 10.1177/027836499801700706. eprint: https://doi.org/10.1177/027836499801700706. URL: https://doi.org/10.1177/027836499801700706.

[SK13a] Arun Kumar Singh and K. Madhava Krishna. "Reactive collision avoidance for multiple robots by non linear time scaling". In: *52nd IEEE Conference on Decision and Control*. 2013, pp. 952–958. DOI: 10.1109/CDC.2013.6760005.

[SK13b] Arun Kumar Singh and K. Madhava Krishna. "Reactive collision avoidance for multiple robots by non linear time scaling". In: *52nd IEEE Conference on Decision and Control*. 2013, pp. 952–958. DOI: 10.1109/CDC.2013.6760005.

[The+19] Raghu Ram Theerthala et al. "Motion Planning Framework for Autonomous Vehicles: A Time Scaled Collision Cone Interleaved Model Predictive Control Approach". In: *2019 IEEE Intelligent Vehicles Symposium (IV)*. 2019, pp. 1075–1080. DOI: 10.1109/IVS.2019.8813823.

A Rule-based Constant Time Scaling

A.1 Independent variable time-scaling

The code below will do rule-based constant time scaling for an independent variable (you could assume the robot to be holonomic here).

```python
# Testing time scaling
"""
    Testing how velocity time scaling can be accomplished
"""

# %%
import sympy as sp
import numpy as np
from matplotlib import pyplot as plt
from lib.ct_scaling import sp_time_scaling_eq

# %%
# Define trajectory
t = sp.symbols('t', positive=True, real=True)
x_t = t + 4
y_t = 2 * t**2 + 2 * x_t + 4
t_lim = [0, 10]

# Show the x, y plot
t_vals = np.linspace(t_lim[0], t_lim[1], 100)
x_vals = np.array([x_t.subs({t: tv}) for tv in t_vals], float)
y_vals = np.array([y_t.subs({t: tv}) for tv in t_vals], float)
# Show the figure
plt.figure(figsize=(6.4, 7.5))
plt.subplot(2,1,1)
plt.title("X")
plt.plot(t_vals, x_vals)
plt.subplot(2,1,2)
plt.title("Y")
plt.plot(t_vals, y_vals)
plt.show()
plt.figure(figsize=(7, 7))
plt.scatter(x_vals, y_vals, c=t_vals, s=1.0)
plt.colorbar()
plt.show()

# %%
k = 1/2
t_c1 = 4
t_c2 = 5
plt.figure(figsize=(6.4, 7.5))
plt.subplot(2,1,1)
nx1_t, t2_new, tend_new = sp_time_scaling_eq(x_t, t_c1, t_c2, t_lim,
    k)
```

```python
nt1_vals = np.linspace(t_lim[0], tend_new, 100)
nx1_vals = np.array([nx1_t.subs({t: tv}) for tv in nt1_vals], float)
plt.title("X")
plt.plot(nt1_vals, nx1_vals, label="new")
plt.plot(t_vals, x_vals, '--', label="old")
plt.axvline(t_c1, color='k', ls='--')
plt.axvline(t_c2, color='k', ls=':')
plt.axvline(t2_new, color='k', ls=':')
plt.axhline(x_t.subs(t, t_c2), color='r', ls='-')
plt.axhline(x_t.subs(t, t_lim[1]), color='y', ls='-')
plt.legend()
plt.subplot(2,1,2)
ny1_t, t2_new, tend_new = sp_time_scaling_eq(y_t, t_c1, t_c2, t_lim,
    k)
nt1_vals = np.linspace(t_lim[0], tend_new, 100)
ny1_vals = np.array([ny1_t.subs({t: tv}) for tv in nt1_vals], float)
plt.title("Y")
plt.plot(nt1_vals, ny1_vals, label="new")
plt.plot(t_vals, y_vals, '--', label="old")
plt.axvline(t_c1, color='k', ls='--')
plt.axvline(t_c2, color='k', ls=':')
plt.axvline(t2_new, color='k', ls=':')
plt.axhline(y_t.subs(t, t_c2), color='r', ls='-')
plt.axhline(y_t.subs(t, t_lim[1]), color='y', ls='-')
plt.legend()
plt.show()

# %% Check implementation with actual
dx_t = x_t.diff(t)
dy_t = y_t.diff(t)
t1_new = t_c1
t2_new = ((t_c2-t_c1)/k) + t_c1
tend_new = t2_new + (t_lim[1] - t_c2)
del_t2 = t2_new - t_c2  # Shift for the end part
# New equations (after time scaling)
new_dx_t = sp.Piecewise((dx_t, t < t1_new),
    (k*dx_t.subs({t: k*(t-t_c1)+t_c1}), t < t2_new),
    (dx_t.subs({t: t-del_t2}), True))
new_dy_t = sp.Piecewise((dy_t, t < t1_new),
    (k*dy_t.subs({t: k*(t-t_c1)+t_c1}), t < t2_new),
    (dy_t.subs({t: t-del_t2}), True))

nx_t = sp.integrate(new_dx_t) + x_t.subs({t:0})
ny_t = sp.integrate(new_dy_t) + y_t.subs({t:0})
nt_vals = np.linspace(t_lim[0], tend_new, 100)
nx_vals = np.array([nx_t.subs({t: tv}) for tv in nt_vals], float)
ny_vals = np.array([ny_t.subs({t: tv}) for tv in nt_vals], float)
plt.figure(figsize=(6.4, 7.5))
plt.subplot(2,1,1)
plt.title("X")
plt.plot(nt_vals, nx_vals, label="new")
plt.plot(t_vals, x_vals, '--', label="old")
plt.axvline(t_c1, color='k', ls='--')
plt.axvline(t_c2, color='k', ls=':')
plt.axvline(t2_new, color='k', ls=':')
plt.axhline(x_t.subs(t, t_c2), color='r', ls='-')
plt.axhline(x_t.subs(t, t_lim[1]), color='y', ls='-')
plt.legend()
plt.subplot(2,1,2)
plt.title("Y")
plt.plot(nt_vals, ny_vals, label="new")
plt.plot(t_vals, y_vals, '--', label="old")
plt.axvline(t_c1, color='k', ls='--')
plt.axvline(t_c2, color='k', ls=':')
plt.axvline(t2_new, color='k', ls=':')
plt.axhline(y_t.subs(t, t_c2), color='r', ls='-')
plt.axhline(y_t.subs(t, t_lim[1]), color='y', ls='-')
plt.legend()
plt.show()

# %%
```

A.2 Non-Holonomic Robot

The code below will do rule-based constant time scaling for a non-holonomic robot (where θ is modelled using $\dot{x}$ and $\dot{y}$).

```
# Rule-based Constant time-scaling
"""
    Given a robot trajectory generated through bernstein polynomials
    (modified to return the position and velocities), and the
    trajectory of a holonomic obstacle (straight line equation), we
    alter the robot's velocities (when the robot is in the collision
    bounds).
    This script assumes the following:
    - The robot can independently control two variables (using which
        the bernstein model is created): 'x', and 'tan(theta)'.
        Scaling will be applied to these two variables
    - There should only be one collision with the obstacle and robot.
        The paths should not have multiple intersections (only one)

"""

# %% Import everything
# Main imports
import numpy as np
import sympy as sp
from matplotlib import pyplot as plt
from matplotlib import patches as patch
# For trajectory generation
from lib.three_point_traj_planner import NonHoloThreePtBernstein
from lib.ct_scaling import sp_time_scaling_eq

# %%

# %%

# %% Experimental section
# Generate a random robot trajectory
# ==== Begin: User configuration area (robot trajectory) ====
# Points as [x, y]
start_pt = [0, 0]
end_pt = [40, 40]
way_pt = [20, 25]
# Time values
to, tw, tf = [0., 25., 50.]    # Start, waypoint, end
# Other parameters
ko, kw, kf = [0, np.tan(np.pi/4), 0]    # k = np.tan(theta)
dko, dkw, dkf = [0, 0, 0]
dxo, dxw, dxf = [0, 1, 0]
# ==== End: User configuration area (robot trajectory) ====
# Convert to dictionary (for library)
constraint_dict = {
    "to": to, "tw": tw, "tf": tf,
    "xo": start_pt[0], "xw": way_pt[0], "xf": end_pt[0],
    "yo": start_pt[1], "yw": way_pt[1], "yf": end_pt[1],
    "ko": ko, "kw": kw, "kf": kf,
    "dxo": dxo, "dxw": dxw, "dxf": dxf,
    "dko": dko, "dkw": dkw, "dkf": dkf
}
# Initialize solver
path_solver = NonHoloThreePtBernstein()
# Time symbol
t_sp = sp.symbols('t', real=True, positive=True)
t_all = sp.symbols('t')
# Solve for paths
x_vals, y_vals, th_vals, t_vals, x_t, y_t, th_t = \
    path_solver.solve_wpt_constr(constraint_dict)
# Substitute 't' with real and positive 't' (time substitution)
x_t = x_t.subs({t_all: t_sp})
y_t = y_t.subs({t_all: t_sp})
th_t = th_t.subs({t_all: t_sp})
# Plot trajectories
plt.figure()
plt.title("XY plot")
```

```python
plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
plt.colorbar()
plt.show()

# %% Collision with an obstacle
# ==== Begin: User configuration area (obstacle) ====
obs_t_col = 25       # Time of collision (for x, y intermediate)
obs_start = (5, 35)      # (x, y): Starting point of obstacle
# ==== End: User configuration area (obstacle) ====
ox_i = float(x_t.subs({t_sp: obs_t_col}))
oy_i = float(y_t.subs({t_sp: obs_t_col}))
obs_x_t = obs_start[0] + ((ox_i - obs_start[0])/obs_t_col) * t_sp
obs_y_t = obs_start[1] + ((oy_i - obs_start[1])/obs_t_col) * t_sp
# Time, x, y trajectories (array) - visualize
obs_t_vals = t_vals.copy()  # np.linspace(to, tf, 100)
obs_x_vals = np.array([obs_x_t.subs({t_sp: tv}) for tv in obs_t_vals])
obs_y_vals = np.array([obs_y_t.subs({t_sp: tv}) for tv in obs_t_vals])

# %% Show the collision
plt.figure()
plt.title("XY plot")
plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
plt.scatter(obs_x_vals, obs_y_vals, 1.0, c=t_vals)
plt.colorbar()
plt.show()

# %%

# %% Show the evolution as time functions
obs_rad = 1      # Obstacle radius
rob_rad = 2      # Robot radius
# Show the figure
fig = plt.figure(num="Original Trajectory")
ax = fig.add_subplot()
ax.set_aspect('equal')
# v_i = 49
# if True:
for v_i in range(len(t_vals)):  # FIXME: Don't run in VSCode (15s!)
    # Reset animation
    ax.cla()
    # Show the obstacle
    obs_body = patch.Circle((obs_x_vals[v_i], obs_y_vals[v_i]),
        obs_rad, ec='k', fc="#F06767", zorder=3.5)
    # Show the robot
    rob_body = patch.Circle((x_vals[v_i], y_vals[v_i]), rob_rad,
        ec='k', fc="#88B4E6", alpha=0.5, zorder=3.4)
    # Add patches
    ax.add_patch(obs_body)
    ax.add_patch(rob_body)
    # Show the paths
    ax.plot(obs_x_vals, obs_y_vals, alpha=0.5, zorder=3)
    ax.plot(x_vals, y_vals, alpha=0.5, zorder=3)
    ax.plot(ox_i, oy_i, 'kx', zorder=3)
    # Set limits
    ax.set_xlim(start_pt[0]-5, end_pt[0]+5)
    ax.set_ylim(start_pt[1]-5, end_pt[1]+5)
    # Pause simulation
    plt.pause(0.1)

# %% Collision Avoidance
# ==== Begin: User configuration area (collision avoidance) ====
collav_dist = 5     # Sensor activation distance
k_val = 0.25          # Scaling to apply (to the speed)
# ==== End: User configuration area (collision avoidance) ====
# Distance as time goes
dist_t = ((x_t - obs_x_t)**2 + (y_t - obs_y_t)**2)**0.5 - rob_rad - \
    obs_rad
dist_vals = np.array([dist_t.subs({t_sp: tv}) for tv in t_vals],
    float)
# Time of collision
t_si, t_ei = np.where(dist_vals < collav_dist)[0][[0, -1]]
t_cstart = t_vals[t_si]      # Time of start (for collision)
t_cend = t_vals[t_ei]        # Time of end of collision
```

```python
# Plot the trajectory
plt.figure()
plt.plot(t_vals, dist_vals)
plt.axhline(collav_dist, color='r', ls='--')
plt.axvline(t_cstart, color='k', ls='--')
plt.axvline(t_cend, color='k', ls='--')
plt.show()

# %%
# - Main collision avoidance work (const. time scaling, rule based) -
nx_t, t2_new, tend_new = sp_time_scaling_eq(x_t, t_cstart, t_cend,
    [to, tf], k_val)
ny_t, t2_new, tend_new = sp_time_scaling_eq(y_t, t_cstart, t_cend,
    [to, tf], k_val)
# Do this operation over `tan(theta)` instead of `theta`
k_t = sp.tan(th_t)
nk_t, t2_new, tend_new = sp_time_scaling_eq(k_t, t_cstart, t_cend,
    [to, tf], k_val)
nth_t = sp.atan(k_t)    # Retrieve new theta(t) - This WON'T work
# (Precisely because the system is non-holonomic)
# Backup angle (of path) - Reinforce the theta constraint
nth_t = sp.atan2(ny_t.diff(t_sp), nx_t.diff(t_sp))

# %% Change the original and the obstacle trajectory (for viz.)
# New obstacle trajectory (stay at rest in the end)
new_obs_x_t = sp.Piecewise((obs_x_t, t_sp < tf),
    (obs_x_t.subs({t_sp: tf}), True))
new_obs_y_t = sp.Piecewise((obs_y_t, t_sp < tf),
    (obs_y_t.subs({t_sp: tf}), True))
# New original robot trajectories (stay at rest in the end)
new_orig_x_t = sp.Piecewise((x_t, t_sp < tf),
    (x_t.subs({t_sp: tf}), True))
new_orig_y_t = sp.Piecewise((y_t, t_sp < tf),
    (y_t.subs({t_sp: tf}), True))
new_orig_th_t = sp.Piecewise((th_t, t_sp < tf),
    (th_t.subs({t_sp: tf}), True))

# %% Test scaling robot trajectory
new_t_vals = np.linspace(to, tend_new, 300) # New time stamps
# Obstacle positions
obs_x_vals = np.array([new_obs_x_t.subs({t_sp: tv}) \
    for tv in new_t_vals], float)
obs_y_vals = np.array([new_obs_y_t.subs({t_sp: tv}) \
    for tv in new_t_vals], float)
# Original robot pose
orig_x_vals = np.array([new_orig_x_t.subs({t_sp: tv}) \
    for tv in new_t_vals], float)
orig_y_vals = np.array([new_orig_y_t.subs({t_sp: tv}) \
    for tv in new_t_vals], float)
orig_th_vals = np.array([new_orig_th_t.subs({t_sp: tv}) \
    for tv in new_t_vals], float)
# New robot x, y, theta values
x_vals = np.array([nx_t.subs({t_sp: tv}) \
    for tv in new_t_vals], float)
y_vals = np.array([ny_t.subs({t_sp: tv}) \
    for tv in new_t_vals], float)
th_vals = np.array([nth_t.subs({t_sp: tv}) \
    for tv in new_t_vals], float)
th_vals[-1] = 0.0   # Precaution (at the end of simulation)

# %%
# Show the figure
fig = plt.figure(num="Collision Avoidance", dpi=150)
ax = fig.add_subplot()
ax.set_aspect('equal')
# v_i = 100
# if True:
for v_i in range(len(new_t_vals)):  # FIXME: Don't run in VSCode
    # Reset animation
    ax.cla()
    # Show the obstacle
    obs_body = patch.Circle((obs_x_vals[v_i], obs_y_vals[v_i]),
        obs_rad, ec='k', fc="#F06767", zorder=3.6)
```

```
    # Show the robot (original path with collision)
    rob_body_o = patch.Circle((orig_x_vals[v_i], orig_y_vals[v_i]),
        rob_rad, ec='k', fc="#88B4E6", alpha=0.5, zorder=3.4)
    ax.plot(
        [orig_x_vals[v_i], orig_x_vals[v_i] + \
            rob_rad*np.cos(orig_th_vals[v_i])],
        [orig_y_vals[v_i], orig_y_vals[v_i] + \
            rob_rad*np.sin(orig_th_vals[v_i])], c="#7A0C7A",
        zorder=3.45, alpha=0.5)
    # Show the new robot path (hopefully no collision)
    rob_body = patch.Circle((x_vals[v_i], y_vals[v_i]),
        rob_rad, ec='k', fc="#88B4E6", alpha=1, zorder=3.5)
    ax.plot(
        [x_vals[v_i], x_vals[v_i] + rob_rad*np.cos(th_vals[v_i])],
        [y_vals[v_i], y_vals[v_i] + rob_rad*np.sin(th_vals[v_i])],
        c="#7A0C7A", zorder=3.55)
    # Add patches
    ax.add_patch(obs_body)
    ax.add_patch(rob_body_o)
    ax.add_patch(rob_body)
    # Show the paths
    ax.plot(obs_x_vals, obs_y_vals, alpha=0.5, zorder=3)
    ax.plot(x_vals, y_vals, alpha=0.5, zorder=3)
    ax.plot(ox_i, oy_i, 'kx', zorder=3)
    # Set limits
    ax.set_xlim(start_pt[0]-5, end_pt[0]+5)
    ax.set_ylim(start_pt[1]-5, end_pt[1]+5)
    fig.savefig(f"./out/{v_i}.png")
    # Pause simulation
    # plt.pause(0.1)

# %%
```

B Collision Cone Constant Time Scaling

The below will do collision cone constant time scaling for a non-holonomic robot (where θ is modelled using the robot's velocities).

```
# Collision Cone based constant time scaling
"""
    Given a robot trajectory generated through bernstein polynomials
    (modified to return the position and velocities), and the
    trajectory of a holonomic obstacle (straight line equation), we
    alter the robot's velocities (when the robot is in the collision
    bounds) using collision cones.
    The script assumes the following:
    - The scaling is applied to robot's velocities. The scaling 's'
        is found using collision cone equations (refer PDF submission)
    - There can be multiple collisions, but the settings should allow
        time scaling as a viable solution (obstacle shouldn't stop on
        path)

    Adjust properties in the following sections
    - ==== User configuration area (robot trajectory) ====
    - ==== User configuration area (obstacle) ====

"""

# %% Import everything
# Main imports
import numpy as np
import sympy as sp
from matplotlib import pyplot as plt
from matplotlib import patches as patch
# For trajectory generation
from lib.three_point_traj_planner import NonHoloThreePtBernstein
# Utilities
import time
from tqdm import tqdm

# %%
```

```
# %%

# %% Experimental section
# Generate a random robot trajectory
# ==== Begin: User configuration area (robot trajectory) ====
# Points as [x, y]
start_pt = [0, 0]
end_pt = [40, 40]
way_pt = [20, 25]
# Time values
to, tw, tf = [0., 25., 50.]    # Start, waypoint, end
# Other parameters
ko, kw, kf = [0, np.tan(np.pi/4), 0]    # k = np.tan(theta)
dko, dkw, dkf = [0, 0, 0]
dxo, dxw, dxf = [0, 1, 0]
# ==== End: User configuration area (robot trajectory) ====
# Convert to dictionary (for library)
constraint_dict = {
    "to": to, "tw": tw, "tf": tf,
    "xo": start_pt[0], "xw": way_pt[0], "xf": end_pt[0],
    "yo": start_pt[1], "yw": way_pt[1], "yf": end_pt[1],
    "ko": ko, "kw": kw, "kf": kf,
    "dxo": dxo, "dxw": dxw, "dxf": dxf,
    "dko": dko, "dkw": dkw, "dkf": dkf
}
# Initialize solver
path_solver = NonHoloThreePtBernstein()
# Time symbol
t_sp = sp.symbols('t', real=True, positive=True)
t_all = sp.symbols('t') # Generic time symbol (used by functions)
print("Finding path")
# Solve for paths
x_vals, y_vals, th_vals, t_vals, x_t, y_t, th_t = \
    path_solver.solve_wpt_constr(constraint_dict)
print("Path found")
# Substitute 't' with real and positive 't' (time substitution)
x_t = x_t.subs({t_all: t_sp})
y_t = y_t.subs({t_all: t_sp})
th_t = th_t.subs({t_all: t_sp})
# Plot trajectories
plt.figure()
plt.title("XY plot")
plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
plt.colorbar()
plt.show()

# %% Collision with an obstacle
# ==== Begin: User configuration area (obstacle) ====
obs_t_col = 25      # Time of collision (for x, y intermediate)
obs_start = (5, 35)     # (x, y): Starting point of obstacle
obs_rad = 1     # Obstacle radius
rob_rad = 2     # Robot radius
detection_bound = 7    # Sensor for collision check (else scale = 1)
num_sim_samples = 300   # Number of time steps (not for saving!)
ks_val_min = 0.15       # Minimum scaling factor
# ==== End: User configuration area (obstacle) ====
# Location where collision will take place
ox_i = float(x_t.subs({t_sp: obs_t_col}))
oy_i = float(y_t.subs({t_sp: obs_t_col}))
obs_x_t = obs_start[0] + ((ox_i - obs_start[0])/obs_t_col) * t_sp
obs_y_t = obs_start[1] + ((oy_i - obs_start[1])/obs_t_col) * t_sp
# Time, x, y trajectories (array) - visualize
obs_t_vals = t_vals.copy()  # np.linspace(to, tf, 100)
obs_x_vals = np.array([obs_x_t.subs({t_sp: tv}) \
    for tv in obs_t_vals])
obs_y_vals = np.array([obs_y_t.subs({t_sp: tv}) \
    for tv in obs_t_vals])

# %% Show the collision
plt.figure()
plt.title("XY plot")
plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
```

```
plt.scatter(obs_x_vals, obs_y_vals, 1.0, c=t_vals)
plt.colorbar()
plt.show()

# %% Prepare the system - Collision Cone
x1, x2, y1, y2 = sp.symbols(r"x1, x2, y1, y2", real=True)
dx1, dy1 = sp.symbols(r"\dot{x}_1, \dot{y}_1", real=True)
dx2, dy2 = sp.symbols(r"\dot{x}_2, \dot{y}_2", real=True)
R = sp.symbols(r"R", real=True, positive=True)
s = sp.symbols(r"s", real=True)
# Modify LHS
ineq_lhs_orig = (x1-x2)**2 + (y1-y2)**2 - R**2 - \
    (((s*dx1-dx2)*(x1-x2) + (s*dy1-dy2)*(y1-y2))**2/ \
        ((s*dx1-dx2)**2 + (s*dy1-dy2)**2))
ineq_lhs = ineq_lhs_orig * ((s*dx1-dx2)**2 + (s*dy1-dy2)**2)
# Get coefficients
ineq_lhs_poly = sp.Poly(ineq_lhs.apart(s), s)
all_coeffs = ineq_lhs_poly.all_coeffs()
# a*s**2 + b*s + c
a, b, c = all_coeffs
assert a*s**2 + b*s + c == sp.simplify(ineq_lhs).apart(s)
r1 = (-b - (b**2 - 4*a*c)**0.5)/(2*a)   # Root 1
r2 = (-b + (b**2 - 4*a*c)**0.5)/(2*a)   # Root 2
d = b**2 - 4*a*c    # Discriminant, should be > 0

# %% Main simulation loop (with collision avoidance)
start_ctime = time.time()   # Start computer time
# Declare velocities of robot
vx_t = x_t.diff(t_sp)
vy_t = y_t.diff(t_sp)   # Get theta from velocities
# Declare velocities of obstacle
ovx_t = obs_x_t.diff(t_sp)
ovy_t = obs_y_t.diff(t_sp)
# Time values for simulation
t_sim_start, t_sim_end = to, tf
dt_sim_k1 = (t_sim_end - t_sim_start)/num_sim_samples
t_sim = t_sim_start # Current simulation time
# t_sim = 20 # Random start sim time   # FIXME: Remove this!
t_rob_local = t_sim     # Time for robot's tracking (ONLY IN SIM!)
dt_sim = dt_sim_k1  # Currently, scaling = 1
k_val = 1.0     # Value of scaling constant (for all steps)
# Pose vectors for the robot and obstacle
r_robot = [float(x_t.subs(t_sp, t_sim)),
    float(y_t.subs(t_sp, t_sim))]
th_robot = float(th_t.subs(t_sp, t_sim))
r_obstacle = [float(obs_x_t.subs(t_sp, t_sim)),
    float(obs_y_t.subs(t_sp, t_sim))]
# Logging variables (all time in t_sim)
robot_poses = []    # [time, x, y, theta] of the robot
obstacle_poses = [] # [time, x, y] of the obstacle
k_vals = []         # [time, k_val] - Log time scaling factor
dist_vals = []      # [time, dist_rob_obs] - Robot to obstacle
time_vals = []      # [time, t_robot_local] - Robot time (prop)
# Simulation progress bar (for robot local time)
tq_bar = tqdm(total=t_sim_end, leave=False)
# Start simulation
while t_rob_local < t_sim_end:
    # Distance between robot and obstacle ('r' vector)
    dist_ro = float(((r_robot[0] - r_obstacle[0])**2 + \
        (r_robot[1] - r_obstacle[1])**2)**0.5)
    # Time scaling IFF there is a collision (in detection)
    if dist_ro < detection_bound:
        # Values which can be substituted
        subs_sh = {
            R: rob_rad + obs_rad,   # Dialated obstacle radius
            # Robot pose
            x1: r_robot[0], y1: r_robot[1],
            # Robot velocity - in local time (keep track!)
            dx1: vx_t.subs(t_sp, t_rob_local),
            dy1: vy_t.subs(t_sp, t_rob_local),
            # Obstacle pose
            x2: r_obstacle[0], y2: r_obstacle[1],
            # Obstacle velocity
```

```
            dx2: ovx_t.subs(t_sp, t_sim),
            dy2: ovy_t.subs(t_sp, t_sim)
        }
        # See if 'a' > 0 (for parabola solutions)
        if float(a.subs(subs_sh)) > 0:
            # See if 'discriminator' is < 0
            if float(d.subs(subs_sh)) < 0:
                # Entire range is okay, pick the minimum value
                k_val = ks_val_min
            else:   # 'discriminator' > 0 (roots exist)
                # s = min{ [(-inf, r1) U (r2, inf)] N [s_min, inf] }
                r1_val = float(r1.subs(subs_sh))
                r2_val = float(r2.subs(subs_sh))
                if r1_val < ks_val_min: # 'r1' doesn't matter
                    k_val = max(r2_val, ks_val_min)
                else:   # 'r2' doesn't matter
                    # {S_vals} = (s_min, r1); take min
                    k_val = ks_val_min
        else:   # 'a' < 0 here
            # See if 'discriminator' is < 0
            if float(d.subs(subs_sh)) < 0:
                # No solution for poly > 0 possible (all -ve vals)
                raise Exception("Time scaling not possible")
            else:   # 'discriminator' > 0
                # s = min{ [r1, r2] N [s_min, inf] }
                r1_val = float(r1.subs(subs_sh))
                r2_val = float(r2.subs(subs_sh))
                if r2_val < ks_val_min: # {S_vals} = NULL!
                    # raise Exception("Time scaling not possible")
                    # Fingers crossed: Hopefully no collision!
                    k_val = ks_val_min
                else:
                    k_val = max(r1_val, ks_val_min)
    else:   # Not in detection bounds, don't bother scaling
        k_val = 1.0
    # Continue robot simulation with k_val (float) scaling
    # Using velocities, progress the next states
    r_obstacle = [  # Use real time for obstacle updates
        float(r_obstacle[0] + ovx_t.subs(t_sp, t_sim) * dt_sim),
        float(r_obstacle[1] + ovy_t.subs(t_sp, t_sim) * dt_sim),
    ]
    robot_dx = float(k_val * vx_t.subs(t_sp, t_rob_local) * dt_sim)
    robot_dy = float(k_val * vy_t.subs(t_sp, t_rob_local) * dt_sim)
    r_robot = [
        float(r_robot[0] + robot_dx), float(r_robot[1] + robot_dy)
    ]
    th_robot = np.arctan2(robot_dy, robot_dx)
    # Log these values
    robot_poses.append([t_sim, r_robot[0], r_robot[1], th_robot])
    obstacle_poses.append([t_sim, r_obstacle[0], r_obstacle[1]])
    k_vals.append([t_sim, k_val])
    dist_vals.append([t_sim, dist_ro])
    time_vals.append([t_sim, t_rob_local])
    # Change in time
    t_rob_local += k_val * dt_sim   # Time scale the robot
    t_sim += dt_sim # The simulation proceeds
    tq_bar.update(k_val * dt_sim)
tq_bar.close()
# Convert all logs to numpy arrays
robot_poses = np.array(robot_poses, float)  # [time, x, y, theta]
obstacle_poses = np.array(obstacle_poses, float)    # [time, x, y]
k_vals = np.array(k_vals, float)    # [time, k_val]
dist_vals = np.array(dist_vals, float)  # [time, dist_rob_obs]
time_vals = np.array(time_vals, float)  # [time, t_robot_local]
end_ctime = time.time() # End computer time
print(f"Simulation took {end_ctime - start_ctime:.3f} seconds!")

# %% Get all trajectories (with time clipping)
# Time values
res_tvals = time_vals[:, 0]
# Robot avoiding collision
res_robposes = robot_poses[:, 1:4]  # [x, y, theta]
# Obstacle path
```

```python
res_obsposes_x = np.array([obs_x_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_obsposes_y = np.array([obs_y_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_obsposes = np.stack([res_obsposes_x, res_obsposes_y]).T
# Robot (with collision)
res_crobotposes_x = np.array([x_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_crobotposes_y = np.array([y_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_crobotposes_th = np.array([th_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_crobotposes = np.stack([res_crobotposes_x, res_crobotposes_y,
    res_crobotposes_th]).T
# Fix the last angle
res_robposes[-1, 2] = res_crobotposes[-1, 2]    # Theta fix
# Distance between robot and obstacle
res_cdist = np.linalg.norm(res_crobotposes[:, 0:2] - \
    res_obsposes[:, 0:2], axis=1)
res_dist = np.linalg.norm(res_robposes[:, 0:2] - \
    res_obsposes[:, 0:2], axis=1)
print(f"Processed {res_tvals.shape[0]} time samples")

# %%
# Show the time
plt.figure(figsize=(7, 3))
plt.suptitle("Time and scale")
plt.subplot(1,2,1)
plt.title("Time")
plt.xlabel("Simulation")
plt.ylabel("Robot")
plt.plot(time_vals[:, 0], time_vals[:, 1], '-')
plt.subplot(1,2,2)
plt.title("Scaling factor")
plt.xlabel("Simulation")
plt.plot(k_vals[:, 0], k_vals[:, 1], '-')
plt.tight_layout()
plt.show()
# Show the robot trajectory (avoiding collision)
plt.figure(figsize=(10, 10))
plt.suptitle("Time scaled trajectory")
plt.subplot(3,2,1)
plt.title("X")
plt.plot(res_tvals, res_robposes[:, 0], 'r-', label="Modified")
plt.plot(res_tvals, res_crobotposes[:, 0], 'r--', label="Actual")
plt.legend()
plt.subplot(3,2,3)
plt.title("Y")
plt.plot(res_tvals, res_robposes[:, 1], 'g-', label="Modified")
plt.plot(res_tvals, res_crobotposes[:, 1], 'g--', label="Actual")
plt.legend()
plt.subplot(3,2,5)
plt.title(r"$\theta$")
plt.plot(res_tvals, res_robposes[:, 2], 'b-', label="Modified")
plt.plot(res_tvals, res_crobotposes[:, 2], 'b--', label="Actual")
plt.legend()
# Obstacle trajectory
plt.subplot(3,2,2)
plt.title("Obstacle - X")
plt.plot(res_tvals, res_obsposes[:, 0], 'r-')
plt.subplot(3,2,4)
plt.title("Obstacle - Y")
plt.plot(res_tvals, res_obsposes[:, 1], 'g-')
plt.subplot(3,2,6)
plt.title("Distance")
plt.plot(res_tvals, res_dist, '-', label="Modified")
plt.plot(res_tvals, res_cdist, '--', label="Actual")
plt.axhline(obs_rad + rob_rad, ls='--', c='r')
plt.axhline(detection_bound, ls=':', c='r')
plt.legend()
# Show the plot
plt.tight_layout()
plt.show()
```

```

# %% Show as video
# Show the figure
fig = plt.figure(num="Collision Avoidance", dpi=150)
ax = fig.add_subplot()
ax.set_aspect('equal')
# v_i = 140
# if True:
for v_i in tqdm(range(len(res_tvals))): # FIXME: Don't run in VSCode
    # Reset animation
    ax.cla()
    # Show the obstacle
    obs_body = patch.Circle(
        (res_obsposes[v_i, 0], res_obsposes[v_i, 1]),
        obs_rad, ec='k', fc="#F06767", zorder=3.6)
    # Show the robot (original path with collision)
    rob_body_o = patch.Circle(
        (res_crobotposes[v_i, 0], res_crobotposes[v_i, 1]),
        rob_rad, ec='k', fc="#88B4E6", alpha=0.5, zorder=3.4)
    ax.plot(
        [res_crobotposes[v_i, 0], res_crobotposes[v_i, 0] + \
            rob_rad*np.cos(res_crobotposes[v_i, 2])],
        [res_crobotposes[v_i, 1], res_crobotposes[v_i, 1] + \
            rob_rad*np.sin(res_crobotposes[v_i, 2])], c="#7A0C7A",
        zorder=3.45, alpha=0.5)
    # Show the new robot path (hopefully no collision)
    if abs(k_vals[v_i, 1] - 1.0) > 1e-3:    # TS active
        rb_ec = 'r'
        # Line joining robot and obstacle
        ax.plot([res_robposes[v_i, 0], res_obsposes[v_i, 0]],
            [res_robposes[v_i, 1], res_obsposes[v_i, 1]], c='r',
            lw=0.2, zorder=3.65)    # Above robot and obstacle
    else:
        rb_ec = 'k'
    rob_body = patch.Circle(
        (res_robposes[v_i, 0], res_robposes[v_i, 1]),
        rob_rad, ec=rb_ec, fc="#88B4E6", alpha=1, zorder=3.5)
    ax.plot(
        [res_robposes[v_i, 0], res_robposes[v_i, 0] + \
            rob_rad*np.cos(res_robposes[v_i, 2])],
        [res_robposes[v_i, 1], res_robposes[v_i, 1] + \
            rob_rad*np.sin(res_robposes[v_i, 2])],
        c="#7A0C7A", zorder=3.55)
    # Add patches
    ax.add_patch(obs_body)
    ax.add_patch(rob_body_o)
    ax.add_patch(rob_body)
    # Show the paths
    ax.plot(res_obsposes[:, 0], res_obsposes[:, 1], alpha=0.5,
        zorder=3)
    ax.plot(res_robposes[:, 0], res_robposes[:, 1], alpha=0.5,
        zorder=3)
    # Location where the collision will take place
    ax.plot(ox_i, oy_i, 'kx', zorder=3)
    # Set limits
    ax.set_xlim(start_pt[0]-5, end_pt[0]+5)
    ax.set_ylim(start_pt[1]-5, end_pt[1]+5)
    # Show/store result
    fig.savefig(f"./out/{v_i}.png")   # Use for saving everything
    # plt.pause(0.05)    # Use only for python script
    # plt.show()        # Use only for VSCode


# %%
```

C Rule-based Linear Time Scaling

The code below applies linear time scaling using hardcoded parameters for linear time scaling

```
# Rule-based linear time scaling
"""
```

```
    Given a robot trajectory generated through bernstein polynomials
    (modified to return the position and velocities), and the
    trajectory of a holonomic obstacle (straight line equation), we
    alter the robot's velocities (when the robot is in the collision
    bounds) using a user-defined linear time scaling approach.
    - The scaling is applied to robot's velocities. The scaling 's'
        is given by 's(t) = a + b*t' where 't' is the simulation time
    - For now, the script has been tested only in single collision
        case
"""

# %% Import everything
# Main imports
import numpy as np
import sympy as sp
from matplotlib import pyplot as plt
from matplotlib import patches as patch
# For trajectory generation
from lib.three_point_traj_planner import NonHoloThreePtBernstein
# Utilities
import time
from tqdm import tqdm

# %%

# %%

# %% Experimental section
# Generate a random robot trajectory
# ==== Begin: User configuration area (robot trajectory) ====
# Points as [x, y]
start_pt = [0, 0]
end_pt = [50, 45]
way_pt = [20, 25]
# Time values
to, tw, tf = [0., 25., 50.]    # Start, waypoint, end
# Other parameters
ko, kw, kf = [0, np.tan(np.pi/4), 0]    # k = np.tan(theta)
dko, dkw, dkf = [0, 0, 0]
dxo, dxw, dxf = [0, 1, 0]
# ==== End: User configuration area (robot trajectory) ====
# Convert to dictionary (for library)
constraint_dict = {
    "to": to, "tw": tw, "tf": tf,
    "xo": start_pt[0], "xw": way_pt[0], "xf": end_pt[0],
    "yo": start_pt[1], "yw": way_pt[1], "yf": end_pt[1],
    "ko": ko, "kw": kw, "kf": kf,
    "dxo": dxo, "dxw": dxw, "dxf": dxf,
    "dko": dko, "dkw": dkw, "dkf": dkf
}
# Initialize solver
path_solver = NonHoloThreePtBernstein()
# Time symbol
t_sp = sp.symbols('t', real=True, positive=True)
t_all = sp.symbols('t') # Generic time symbol (used by functions)
print("Finding path")
# Solve for paths
x_vals, y_vals, th_vals, t_vals, x_t, y_t, th_t = \
    path_solver.solve_wpt_constr(constraint_dict)
print("Path found")
# Substitute 't' with real and positive 't' (time substitution)
x_t = x_t.subs({t_all: t_sp})
y_t = y_t.subs({t_all: t_sp})
th_t = th_t.subs({t_all: t_sp})
# Plot trajectories
plt.figure()
plt.title("XY plot")
plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
plt.colorbar()
plt.show()

# %% Collision with an obstacle
# ==== Begin: User configuration area (obstacle) ====
```

```
obs_t_col = 15      # Time of collision (for x, y intermediate)
obs_start = (4, 20)     # (x, y): Starting point of obstacle
obs_rad = 1     # Obstacle radius
rob_rad = 2.5     # Robot radius
detection_bound = 10    # Sensor for collision check (else scale = 1)
# s_func = _a + _b * t -> Functions for 's' (scaling term). t is sim.
sfunc_a = 0.001         # Constant term for s_func
sfunc_b = 0.02          # Time term for s_func
num_sim_samples = 300   # Number of time steps (not for saving!)
# ==== End: User configuration area (obstacle) ====
# Location where collision will take place
ox_i = float(x_t.subs({t_sp: obs_t_col}))
oy_i = float(y_t.subs({t_sp: obs_t_col}))
obs_x_t = obs_start[0] + ((ox_i - obs_start[0])/obs_t_col) * t_sp
obs_y_t = obs_start[1] + ((oy_i - obs_start[1])/obs_t_col) * t_sp
# Time, x, y trajectories (array) - visualize
obs_t_vals = t_vals.copy()  # np.linspace(to, tf, 100)
obs_x_vals = np.array([obs_x_t.subs({t_sp: tv}) \
    for tv in obs_t_vals])
obs_y_vals = np.array([obs_y_t.subs({t_sp: tv}) \
    for tv in obs_t_vals])

# %% Show the collision
plt.figure()
plt.title("XY plot")
plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
plt.scatter(obs_x_vals, obs_y_vals, 1.0, c=t_vals)
plt.colorbar()
plt.show()

# %% Main simulation loop (with collision avoidance)
start_ctime = time.time()   # Start computer time
# Declare velocities of robot
vx_t = x_t.diff(t_sp)
vy_t = y_t.diff(t_sp)   # Get theta from velocities
# Declare velocities of obstacle
ovx_t = obs_x_t.diff(t_sp)
ovy_t = obs_y_t.diff(t_sp)
# Time values for simulation
t_sim_start, t_sim_end = to, tf
dt_sim_k1 = (t_sim_end - t_sim_start)/num_sim_samples
t_sim = t_sim_start # Current simulation time
# t_sim = 20 # Random start sim time   # FIXME: Remove this!
t_rob_local = t_sim     # Time for robot's tracking (ONLY IN SIM!)
dt_sim = dt_sim_k1  # Currently, scaling = 1
k_val = 1.0     # Value of scaling constant (for all steps)
# Pose vectors for the robot and obstacle
r_robot = [float(x_t.subs(t_sp, t_sim)),
    float(y_t.subs(t_sp, t_sim))]
th_robot = float(th_t.subs(t_sp, t_sim))
r_obstacle = [float(obs_x_t.subs(t_sp, t_sim)),
    float(obs_y_t.subs(t_sp, t_sim))]
# Logging variables (all time in t_sim)
robot_poses = []    # [time, x, y, theta] of the robot
obstacle_poses = [] # [time, x, y] of the obstacle
k_vals = []         # [time, k_val] - Log time scaling factor
dist_vals = []      # [time, dist_rob_obs] - Robot to obstacle
time_vals = []      # [time, t_robot_local] - Robot time (prop)
# Simulation progress bar (for robot local time)
tq_bar = tqdm(total=t_sim_end, leave=False)
# Start simulation
while t_rob_local < t_sim_end:
    # Distance between robot and obstacle ('r' vector)
    dist_ro = float(((r_robot[0] - r_obstacle[0])**2 + \
        (r_robot[1] - r_obstacle[1])**2)**0.5)
    if dist_ro < detection_bound:
        # Linear time scaling function for scaling factor
        k_val = sfunc_a + sfunc_b * t_sim
    else:
        k_val = 1.0
    # Continue robot simulation with k_val (float) scaling
    # Using velocities, progress the next states
    r_obstacle = [  # Use real time for obstacle updates
```

```
        float(r_obstacle[0] + ovx_t.subs(t_sp, t_sim) * dt_sim),
        float(r_obstacle[1] + ovy_t.subs(t_sp, t_sim) * dt_sim),
    ]
    robot_dx = float(k_val * vx_t.subs(t_sp, t_rob_local) * dt_sim)
    robot_dy = float(k_val * vy_t.subs(t_sp, t_rob_local) * dt_sim)
    r_robot = [
        float(r_robot[0] + robot_dx), float(r_robot[1] + robot_dy)
    ]
    th_robot = np.arctan2(robot_dy, robot_dx)
    # Log these values
    robot_poses.append([t_sim, r_robot[0], r_robot[1], th_robot])
    obstacle_poses.append([t_sim, r_obstacle[0], r_obstacle[1]])
    k_vals.append([t_sim, k_val])
    dist_vals.append([t_sim, dist_ro])
    time_vals.append([t_sim, t_rob_local])
    # Change in time
    t_rob_local += k_val * dt_sim   # Time scale the robot
    t_sim += dt_sim # The simulation proceeds
    tq_bar.update(k_val * dt_sim)
tq_bar.close()
# Convert all logs to numpy arrays
robot_poses = np.array(robot_poses, float)  # [time, x, y, theta]
obstacle_poses = np.array(obstacle_poses, float)    # [time, x, y]
k_vals = np.array(k_vals, float)    # [time, k_val]
dist_vals = np.array(dist_vals, float)  # [time, dist_rob_obs]
time_vals = np.array(time_vals, float)  # [time, t_robot_local]
end_ctime = time.time() # End computer time
print(f"Simulation took {end_ctime - start_ctime:.3f} seconds!")

# %% Get all trajectories (with time clipping)
# Time values
res_tvals = time_vals[:, 0]
# Robot avoiding collision
res_robposes = robot_poses[:, 1:4]  # [x, y, theta]
# Obstacle path
res_obsposes_x = np.array([obs_x_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_obsposes_y = np.array([obs_y_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_obsposes = np.stack([res_obsposes_x, res_obsposes_y]).T
# Robot (with collision)
res_crobotposes_x = np.array([x_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_crobotposes_y = np.array([y_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_crobotposes_th = np.array([th_t.subs(t_sp, min(tv, tf)) \
    for tv in res_tvals], float)
res_crobotposes = np.stack([res_crobotposes_x, res_crobotposes_y,
    res_crobotposes_th]).T
# Fix the last angle
res_robposes[-1, 2] = res_crobotposes[-1, 2]    # Theta fix
# Distance between robot and obstacle
res_cdist = np.linalg.norm(res_crobotposes[:, 0:2] - \
    res_obsposes[:, 0:2], axis=1)
res_dist = np.linalg.norm(res_robposes[:, 0:2] - \
    res_obsposes[:, 0:2], axis=1)
print(f"Processed {res_tvals.shape[0]} time samples")

# %%
# Show the time
plt.figure(figsize=(7, 3))
plt.suptitle("Time and scale")
plt.subplot(1,2,1)
plt.title("Time")
plt.xlabel("Simulation")
plt.ylabel("Robot")
plt.plot(time_vals[:, 0], time_vals[:, 1], '-')
plt.subplot(1,2,2)
plt.title("Scaling factor")
plt.xlabel("Simulation")
plt.plot(k_vals[:, 0], k_vals[:, 1], '-')
plt.tight_layout()
plt.show()
```

```python
# Show the robot trajectory (avoiding collision)
plt.figure(figsize=(10, 10))
plt.suptitle("Time scaled trajectory")
plt.subplot(3,2,1)
plt.title("X")
plt.plot(res_tvals, res_robposes[:, 0], 'r-', label="Modified")
plt.plot(res_tvals, res_crobotposes[:, 0], 'r--', label="Actual")
plt.legend()
plt.subplot(3,2,3)
plt.title("Y")
plt.plot(res_tvals, res_robposes[:, 1], 'g-', label="Modified")
plt.plot(res_tvals, res_crobotposes[:, 1], 'g--', label="Actual")
plt.legend()
plt.subplot(3,2,5)
plt.title(r"$\theta$")
plt.plot(res_tvals, res_robposes[:, 2], 'b-', label="Modified")
plt.plot(res_tvals, res_crobotposes[:, 2], 'b--', label="Actual")
plt.legend()
# Obstacle trajectory
plt.subplot(3,2,2)
plt.title("Obstacle - X")
plt.plot(res_tvals, res_obsposes[:, 0], 'r-')
plt.subplot(3,2,4)
plt.title("Obstacle - Y")
plt.plot(res_tvals, res_obsposes[:, 1], 'g-')
plt.subplot(3,2,6)
plt.title("Distance")
plt.plot(res_tvals, res_dist, '-', label="Modified")
plt.plot(res_tvals, res_cdist, '--', label="Actual")
plt.axhline(obs_rad + rob_rad, ls='--', c='r')
plt.axhline(detection_bound, ls=':', c='r')
plt.legend()
# Show the plot
plt.tight_layout()
plt.show()

# %% Show as video
# Show the figure
fig = plt.figure(num="Collision Avoidance", dpi=150)
ax = fig.add_subplot()
ax.set_aspect('equal')
# v_i = 140
# if True:
for v_i in tqdm(range(len(res_tvals))): # FIXME: Don't run in VSCode
    # Reset animation
    ax.cla()
    # Show the obstacle
    obs_body = patch.Circle(
        (res_obsposes[v_i, 0], res_obsposes[v_i, 1]),
        obs_rad, ec='k', fc="#F06767", zorder=3.6)
    # Show the robot (original path with collision)
    rob_body_o = patch.Circle(
        (res_crobotposes[v_i, 0], res_crobotposes[v_i, 1]),
        rob_rad, ec='k', fc="#88B4E6", alpha=0.5, zorder=3.4)
    ax.plot(
        [res_crobotposes[v_i, 0], res_crobotposes[v_i, 0] + \
            rob_rad*np.cos(res_crobotposes[v_i, 2])],
        [res_crobotposes[v_i, 1], res_crobotposes[v_i, 1] + \
            rob_rad*np.sin(res_crobotposes[v_i, 2])], c="#7A0C7A",
        zorder=3.45, alpha=0.5)
    # Show the new robot path (hopefully no collision)
    if abs(k_vals[v_i, 1] - 1.0) > 1e-3:    # TS active
        rb_ec = 'r'
        # Line joining robot and obstacle
        ax.plot([res_robposes[v_i, 0], res_obsposes[v_i, 0]],
            [res_robposes[v_i, 1], res_obsposes[v_i, 1]], c='r',
            lw=0.2, zorder=3.65)    # Above robot and obstacle
    else:
        rb_ec = 'k'
    rob_body = patch.Circle(
        (res_robposes[v_i, 0], res_robposes[v_i, 1]),
        rob_rad, ec=rb_ec, fc="#88B4E6", alpha=1, zorder=3.5)
    ax.plot(
```

```
        [res_robposes[v_i, 0], res_robposes[v_i, 0] + \
            rob_rad*np.cos(res_robposes[v_i, 2])],
        [res_robposes[v_i, 1], res_robposes[v_i, 1] + \
            rob_rad*np.sin(res_robposes[v_i, 2])],
        c="#7A0C7A", zorder=3.55)
    # Add patches
    ax.add_patch(obs_body)
    ax.add_patch(rob_body_o)
    ax.add_patch(rob_body)
    # Show the paths
    ax.plot(res_obsposes[:, 0], res_obsposes[:, 1], alpha=0.5,
        zorder=3)
    ax.plot(res_robposes[:, 0], res_robposes[:, 1], alpha=0.5,
        zorder=3)
    # Location where the collision will take place
    ax.plot(ox_i, oy_i, 'kx', zorder=3)
    # Set limits
    ax.set_xlim(start_pt[0]-5, end_pt[0]+5)
    ax.set_ylim(start_pt[1]-5, end_pt[1]+5)
    # Show/store result
    fig.savefig(f"./out/{v_i}.png")   # Use for saving everything
    # plt.pause(0.05)    # Use only for python script
    # plt.show()        # Use only for VSCode


# %%
```