

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

¹Videos and results available at https://iiitaphyd-my.sharepoint.com/:f:/g/personal/avneesh_mishra_research_iiit_ac_in/Er_wRqK4hxVLjVdL56rfDxYBKr9PPed11aN48hLgLisf4w

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

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

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.

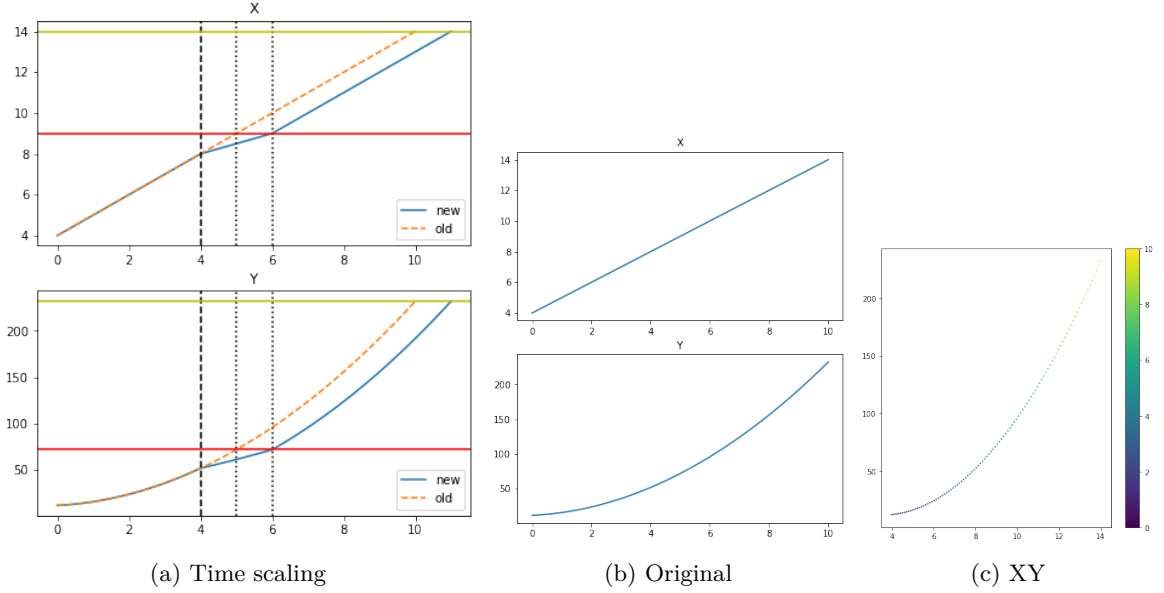


Figure 1: Time scaling

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 $\theta = \text{atan2}(\dot{y}, \dot{x})$ 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

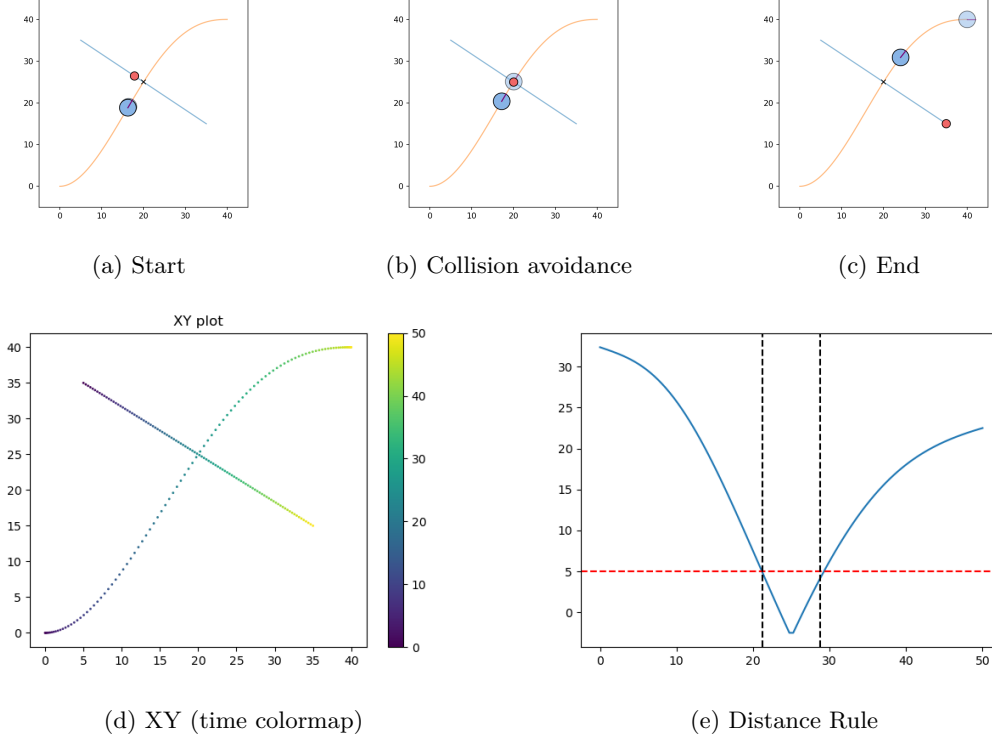


Figure 2: Rule-based constant time-scaling

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

$$\begin{aligned}
 DB \geq DC &\Rightarrow DB^2 \geq R^2 \Rightarrow AD^2 - AB^2 \geq R^2 & \vec{r}_{obstacle} = \vec{r}_{robot} + \vec{r} &\Rightarrow \vec{r} = \vec{r}_{obstacle} - \vec{r}_{robot} \\
 AB = \|\vec{r}\| \cos(\theta) = \frac{\vec{V}_{rel} \cdot \vec{r}}{\|\vec{V}_{rel}\|} &\Rightarrow AB^2 = \left[\frac{\vec{V}_{rel} \cdot \vec{r}}{\|\vec{V}_{rel}\|} \right]^2 & AD = \|\vec{r}\| &\Rightarrow AD^2 = \|\vec{r}\|^2
 \end{aligned}$$

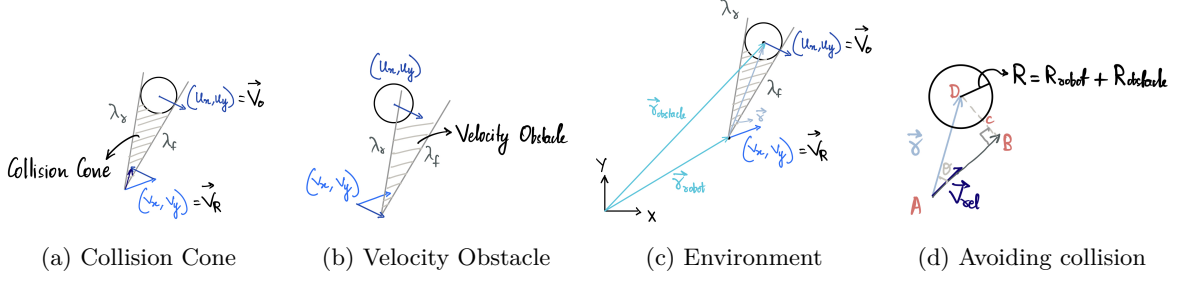


Figure 3: Collision Cone

The velocity of obstacle is $\vec{V}_O = (u_x, u_y)$. The velocity of robot is $\vec{V}_R = (v_x, v_y)$. The radius of robot is R_{robot} and the radius of obstacle is $R_{obstacle}$. The obstacle is dilated 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 $\vec{V}_R = (s\dot{x}_1, s\dot{y}_1)$.

$$\begin{aligned} \vec{V}_{rel} &= \vec{V}_R - \vec{V}_O = (s\dot{x}_1 - \dot{x}_2, s\dot{y}_1 - \dot{y}_2) & \vec{r} &= \vec{r}_{obstacle} - \vec{r}_{robot} = (x_2 - x_1, y_2 - y_1) \\ AD^2 - AB^2 \geq R^2 &\Rightarrow \|\vec{r}\|^2 - \left[\frac{\vec{V}_{rel} \cdot \vec{r}}{\|\vec{V}_{rel}\|} \right]^2 \geq R^2 & \|\vec{r}\|^2 - \left[\frac{\vec{V}_{rel} \cdot \vec{r}}{\|\vec{V}_{rel}\|} \right]^2 - R^2 &\geq 0 \\ \vec{V}_{rel} \cdot \vec{r} &= (s\dot{x}_1 - \dot{x}_2)(x_2 - x_1) + (s\dot{y}_1 - \dot{y}_2)(y_2 - y_1) & \|\vec{V}_{rel}\| &= \sqrt{(s\dot{x}_1 - \dot{x}_2)^2 + (s\dot{y}_1 - \dot{y}_2)^2} \\ (x_1 - x_2)^2 + (y_1 - y_2)^2 - R^2 - \frac{((s\dot{x}_1 - \dot{x}_2)(x_2 - x_1) + (s\dot{y}_1 - \dot{y}_2)(y_2 - y_1))^2}{(s\dot{x}_1 - \dot{x}_2)^2 + (s\dot{y}_1 - \dot{y}_2)^2} &\geq 0 \end{aligned} \quad (1.2)$$

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

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

Where

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

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).

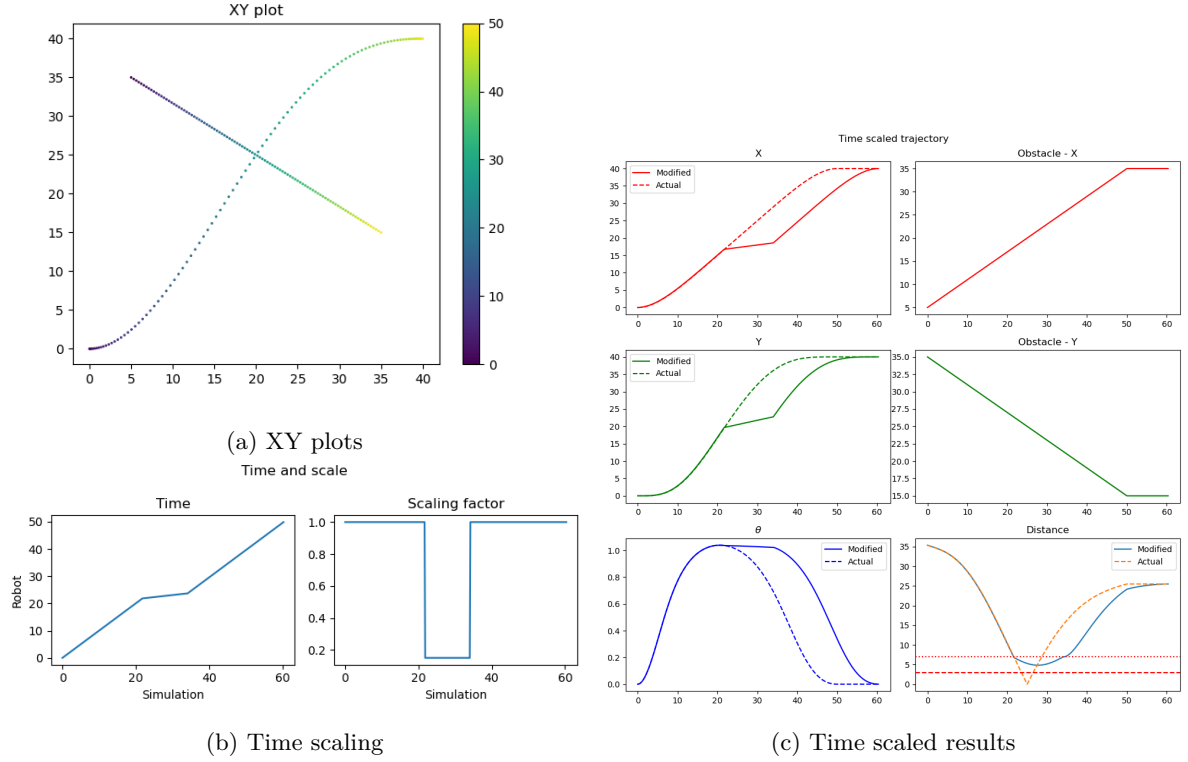


Figure 4: Collision cone based constant time scaling

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.

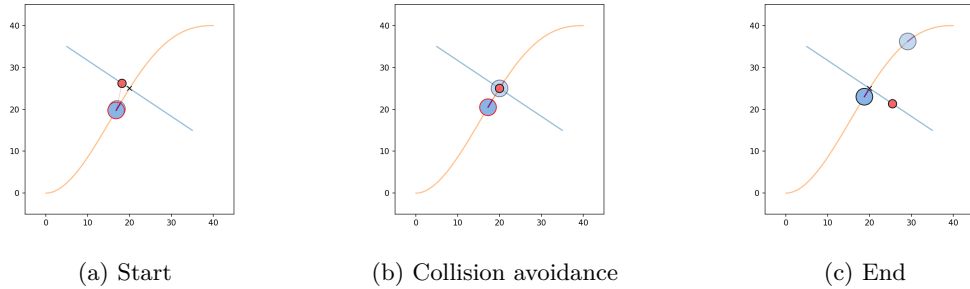


Figure 5: Collision cone - Constant time scaling - Video snaps

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.

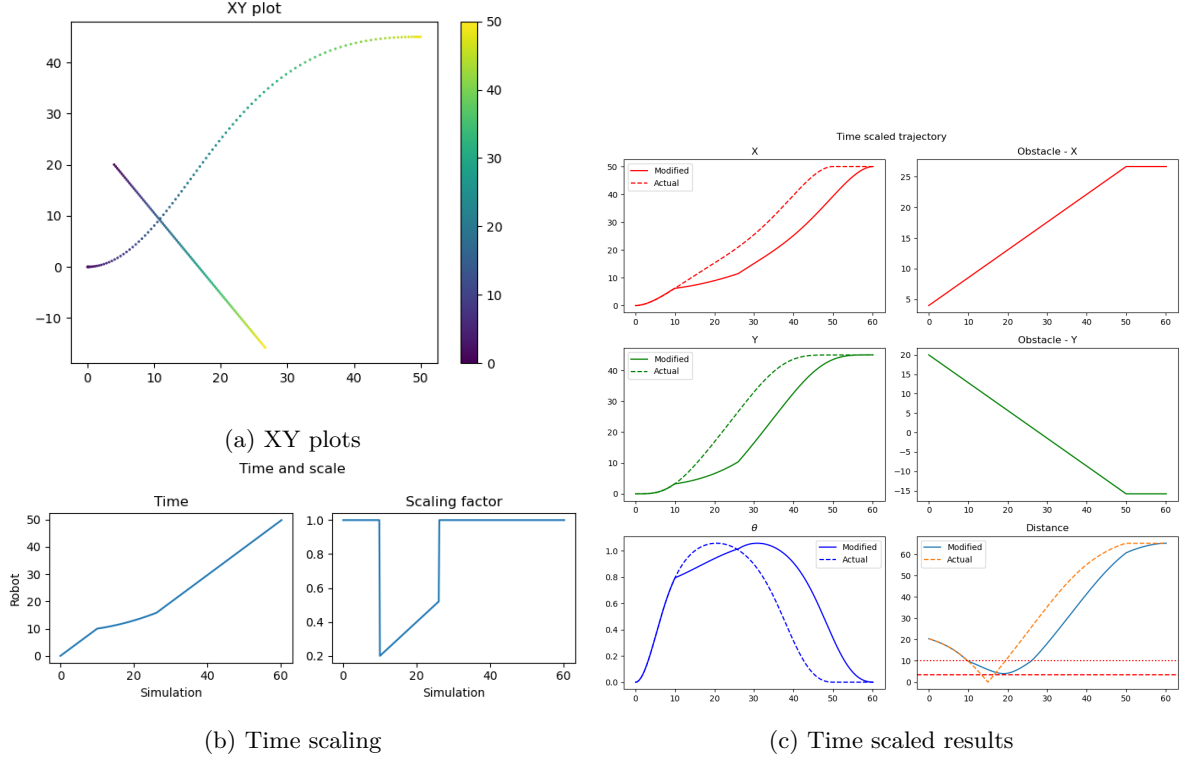


Figure 6: Rule-based Linear time scaling

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.

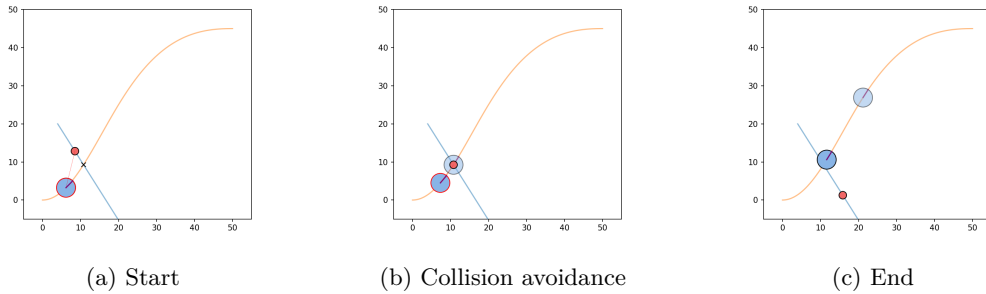


Figure 7: Rule-based - Linear time scaling - Video snaps

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).

```
1 # Testing time scaling
2 """
3     Testing how velocity time scaling can be accomplished
4 """
5
6 # %%
7 import sympy as sp
8 import numpy as np
9 from matplotlib import pyplot as plt
10 from lib.ct_scaling import sp_time_scaling_eq
11
12 # %%
13 # Define trajectory
14 t = sp.symbols('t', positive=True, real=True)
15 x_t = t + 4
16 y_t = 2 * t**2 + 2 * x_t + 4
17 t_lim = [0, 10]
18
19 # Show the x, y plot
20 t_vals = np.linspace(t_lim[0], t_lim[1], 100)
21 x_vals = np.array([x_t.subs({t: tv}) for tv in t_vals], float)
22 y_vals = np.array([y_t.subs({t: tv}) for tv in t_vals], float)
23 # Show the figure
24 plt.figure(figsize=(6.4, 7.5))
25 plt.subplot(2,1,1)
26 plt.title("X")
27 plt.plot(t_vals, x_vals)
28 plt.subplot(2,1,2)
29 plt.title("Y")
30 plt.plot(t_vals, y_vals)
31 plt.show()
32 plt.figure(figsize=(7, 7))
33 plt.scatter(x_vals, y_vals, c=t_vals, s=1.0)
34 plt.colorbar()
35 plt.show()
36
37 # %%
38 k = 1/2
39 t_c1 = 4
40 t_c2 = 5
41 plt.figure(figsize=(6.4, 7.5))
42 plt.subplot(2,1,1)
43 nx1_t, t2_new, tend_new = sp_time_scaling_eq(x_t, t_c1, t_c2, t_lim,
44 k)
```



```

45 nt1_vals = np.linspace(t_lim[0], tend_new, 100)
46 nx1_vals = np.array([nx1_t.subs({t: tv}) for tv in nt1_vals], float)
47 plt.title("X")
48 plt.plot(nt1_vals, nx1_vals, label="new")
49 plt.plot(t_vals, x_vals, '--', label="old")
50 plt.axvline(t_c1, color='k', ls='--')
51 plt.axvline(t_c2, color='k', ls=':')
52 plt.axvline(t2_new, color='k', ls=':')
53 plt.axhline(x_t.subs(t, t_c2), color='r', ls='--')
54 plt.axhline(x_t.subs(t, t_lim[1]), color='y', ls='--')
55 plt.legend()
56 plt.subplot(2,1,2)
57 ny1_t, t2_new, tend_new = sp_time_scaling_eq(y_t, t_c1, t_c2, t_lim,
58 k)
59 nt1_vals = np.linspace(t_lim[0], tend_new, 100)
60 ny1_vals = np.array([ny1_t.subs({t: tv}) for tv in nt1_vals], float)
61 plt.title("Y")
62 plt.plot(nt1_vals, ny1_vals, label="new")
63 plt.plot(t_vals, y_vals, '--', label="old")
64 plt.axvline(t_c1, color='k', ls='--')
65 plt.axvline(t_c2, color='k', ls=':')
66 plt.axvline(t2_new, color='k', ls=':')
67 plt.axhline(y_t.subs(t, t_c2), color='r', ls='--')
68 plt.axhline(y_t.subs(t, t_lim[1]), color='y', ls='--')
69 plt.legend()
70 plt.show()
71
72 # %% Check implementation with actual
73 dx_t = x_t.diff(t)
74 dy_t = y_t.diff(t)
75 t1_new = t_c1
76 t2_new = ((t_c2-t_c1)/k) + t_c1
77 tend_new = t2_new + (t_lim[1] - t_c2)
78 del_t2 = t2_new - t_c2 # Shift for the end part
79 # New equations (after time scaling)
80 new_dx_t = sp.Piecewise((dx_t, t < t1_new),
81 (k*dx_t.subs({t: k*(t-t_c1)+t_c1}), t < t2_new),
82 (dx_t.subs({t: t-del_t2}), True))
83 new_dy_t = sp.Piecewise((dy_t, t < t1_new),
84 (k*dy_t.subs({t: k*(t-t_c1)+t_c1}), t < t2_new),
85 (dy_t.subs({t: t-del_t2}), True))
86
87 nx_t = sp.integrate(new_dx_t) + x_t.subs({t:0})
88 ny_t = sp.integrate(new_dy_t) + y_t.subs({t:0})
89 nt_vals = np.linspace(t_lim[0], tend_new, 100)
90 nx_vals = np.array([nx_t.subs({t: tv}) for tv in nt_vals], float)
91 ny_vals = np.array([ny_t.subs({t: tv}) for tv in nt_vals], float)
92 plt.figure(figsize=(6.4, 7.5))
93 plt.subplot(2,1,1)
94 plt.title("X")
95 plt.plot(nt_vals, nx_vals, label="new")
96 plt.plot(t_vals, x_vals, '--', label="old")
97 plt.axvline(t_c1, color='k', ls='--')
98 plt.axvline(t_c2, color='k', ls=':')
99 plt.axvline(t2_new, color='k', ls=':')
100 plt.axhline(x_t.subs(t, t_c2), color='r', ls='--')
101 plt.axhline(x_t.subs(t, t_lim[1]), color='y', ls='--')
102 plt.legend()
103 plt.subplot(2,1,2)
104 plt.title("Y")
105 plt.plot(nt_vals, ny_vals, label="new")
106 plt.plot(t_vals, y_vals, '--', label="old")
107 plt.axvline(t_c1, color='k', ls='--')
108 plt.axvline(t_c2, color='k', ls=':')
109 plt.axvline(t2_new, color='k', ls=':')
110 plt.axhline(y_t.subs(t, t_c2), color='r', ls='--')
111 plt.axhline(y_t.subs(t, t_lim[1]), color='y', ls='--')
112 plt.legend()
113 plt.show()
114
115 # %%

```

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}).

```

1 # Rule-based Constant time-scaling
2 """
3     Given a robot trajectory generated through bernstein polynomials
4     (modified to return the position and velocities), and the
5     trajectory of a holonomic obstacle (straight line equation), we
6     alter the robot's velocities (when the robot is in the collision
7     bounds).
8     This script assumes the following:
9     - The robot can independently control two variables (using which
10       the bernstein model is created): 'x', and 'tan(theta)'.
11       Scaling will be applied to these two variables
12     - There should only be one collision with the obstacle and robot.
13       The paths should not have multiple intersections (only one)
14 """
15
16
17 # %% Import everything
18 # Main imports
19 import numpy as np
20 import sympy as sp
21 from matplotlib import pyplot as plt
22 from matplotlib import patches as patch
23 # For trajectory generation
24 from lib.three_point_traj_planner import NonHoloThreePtBernstein
25 from lib.ct_scaling import sp_time_scaling_eq
26
27 # %%
28 # %%
29
30
31 # %% Experimental section
32 # Generate a random robot trajectory
33 # ==== Begin: User configuration area (robot trajectory) ====
34 # Points as [x, y]
35 start_pt = [0, 0]
36 end_pt = [40, 40]
37 way_pt = [20, 25]
38 # Time values
39 to, tw, tf = [0., 25., 50.] # Start, waypoint, end
40 # Other parameters
41 ko, kw, kf = [0, np.tan(np.pi/4), 0] # k = np.tan(theta)
42 dko, dkw, dkf = [0, 0, 0]
43 dxo, dxw, dxf = [0, 1, 0]
44 # ==== End: User configuration area (robot trajectory) ====
45 # Convert to dictionary (for library)
46 constraint_dict = {
47     "to": to, "tw": tw, "tf": tf,
48     "xo": start_pt[0], "xw": way_pt[0], "xf": end_pt[0],
49     "yo": start_pt[1], "yw": way_pt[1], "yf": end_pt[1],
50     "ko": ko, "kw": kw, "kf": kf,
51     "dxo": dxo, "dxw": dxw, "dxf": dxf,
52     "dko": dko, "dkw": dkw, "dkf": dkf
53 }
54 # Initialize solver
55 path_solver = NonHoloThreePtBernstein()
56 # Time symbol
57 t_sp = sp.symbols('t', real=True, positive=True)
58 t_all = sp.symbols('t')
59 # Solve for paths
60 x_vals, y_vals, th_vals, t_vals, x_t, y_t, th_t = \
61     path_solver.solve_wpt_constr(constraint_dict)
62 # Substitute 't' with real and positive 't' (time substitution)
63 x_t = x_t.subs({t_all: t_sp})
64 y_t = y_t.subs({t_all: t_sp})
65 th_t = th_t.subs({t_all: t_sp})
66 # Plot trajectories
67 plt.figure()
68 plt.title("XY plot")

```

```

69 plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
70 plt.colorbar()
71 plt.show()
72
73 # %% Collision with an obstacle
74 # ==== Begin: User configuration area (obstacle) ====
75 obs_t_col = 25      # Time of collision (for x, y intermediate)
76 obs_start = (5, 35) # (x, y): Starting point of obstacle
77 # ==== End: User configuration area (obstacle) ====
78 ox_i = float(x_t.subs({t_sp: obs_t_col}))
79 oy_i = float(y_t.subs({t_sp: obs_t_col}))
80 obs_x_t = obs_start[0] + ((ox_i - obs_start[0])/obs_t_col) * t_sp
81 obs_y_t = obs_start[1] + ((oy_i - obs_start[1])/obs_t_col) * t_sp
82 # Time, x, y trajectories (array) - visualize
83 obs_t_vals = t_vals.copy() # np.linspace(to, tf, 100)
84 obs_x_vals = np.array([obs_x_t.subs({t_sp: tv}) for tv in obs_t_vals])
85 obs_y_vals = np.array([obs_y_t.subs({t_sp: tv}) for tv in obs_t_vals])
86
87 # %% Show the collision
88 plt.figure()
89 plt.title("XY plot")
90 plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
91 plt.scatter(obs_x_vals, obs_y_vals, 1.0, c=t_vals)
92 plt.colorbar()
93 plt.show()
94
95 # %%
96
97 # %% Show the evolution as time functions
98 obs_rad = 1      # Obstacle radius
99 rob_rad = 2      # Robot radius
100 # Show the figure
101 fig = plt.figure(num="Original Trajectory")
102 ax = fig.add_subplot()
103 ax.set_aspect('equal')
104 # v_i = 49
105 # if True:
106 for v_i in range(len(t_vals)): # FIXME: Don't run in VSCode (15s!)
107     # Reset animation
108     ax.cla()
109     # Show the obstacle
110     obs_body = patch.Circle((obs_x_vals[v_i], obs_y_vals[v_i]),
111                             obs_rad, ec='k', fc="#F06767", zorder=3.5)
112     # Show the robot
113     rob_body = patch.Circle((x_vals[v_i], y_vals[v_i]), rob_rad,
114                             ec='k', fc="#88B4E6", alpha=0.5, zorder=3.4)
115     # Add patches
116     ax.add_patch(obs_body)
117     ax.add_patch(rob_body)
118     # Show the paths
119     ax.plot(obs_x_vals, obs_y_vals, alpha=0.5, zorder=3)
120     ax.plot(x_vals, y_vals, alpha=0.5, zorder=3)
121     ax.plot(ox_i, oy_i, 'kx', zorder=3)
122     # Set limits
123     ax.set_xlim(start_pt[0]-5, end_pt[0]+5)
124     ax.set_ylim(start_pt[1]-5, end_pt[1]+5)
125     # Pause simulation
126     plt.pause(0.1)
127
128 # %% Collision Avoidance
129 # ==== Begin: User configuration area (collision avoidance) ====
130 collav_dist = 5      # Sensor activation distance
131 k_val = 0.25         # Scaling to apply (to the speed)
132 # ==== End: User configuration area (collision avoidance) ====
133 # Distance as time goes
134 dist_t = ((x_t - obs_x_t)**2 + (y_t - obs_y_t)**2)**0.5 - rob_rad - \
135         obs_rad
136 dist_vals = np.array([dist_t.subs({t_sp: tv}) for tv in t_vals],
137                       float)
138 # Time of collision
139 t_si, t_ei = np.where(dist_vals < collav_dist)[0][[0, -1]]
140 t_cstart = t_vals[t_si]      # Time of start (for collision)
141 t_cend = t_vals[t_ei]       # Time of end of collision

```

```

142 # Plot the trajectory
143 plt.figure()
144 plt.plot(t_vals, dist_vals)
145 plt.axhline(collav_dist, color='r', ls='--')
146 plt.axvline(t_cstart, color='k', ls='--')
147 plt.axvline(t_cend, color='k', ls='--')
148 plt.show()
149
150 # %%
151 # - Main collision avoidance work (const. time scaling, rule based) -
152 nx_t, t2_new, tend_new = sp_time_scaling_eq(x_t, t_cstart, t_cend,
153 [to, tf], k_val)
154 ny_t, t2_new, tend_new = sp_time_scaling_eq(y_t, t_cstart, t_cend,
155 [to, tf], k_val)
156 # Do this operation over 'tan(theta)' instead of 'theta'
157 k_t = sp.tan(th_t)
158 nk_t, t2_new, tend_new = sp_time_scaling_eq(k_t, t_cstart, t_cend,
159 [to, tf], k_val)
160 nth_t = sp.atan(k_t) # Retrieve new theta(t) - This WON'T work
161 # (Precisely because the system is non-holonomic)
162 # Backup angle (of path) - Reinforce the theta constraint
163 nth_t = sp.atan2(ny_t.diff(t_sp), nx_t.diff(t_sp))
164
165 # %% Change the original and the obstacle trajectory (for viz.)
166 # New obstacle trajectory (stay at rest in the end)
167 new_obs_x_t = sp.Piecewise((obs_x_t, t_sp < tf),
168 (obs_x_t.subs({t_sp: tf}), True))
169 new_obs_y_t = sp.Piecewise((obs_y_t, t_sp < tf),
170 (obs_y_t.subs({t_sp: tf}), True))
171 # New original robot trajectories (stay at rest in the end)
172 new_orig_x_t = sp.Piecewise((x_t, t_sp < tf),
173 (x_t.subs({t_sp: tf}), True))
174 new_orig_y_t = sp.Piecewise((y_t, t_sp < tf),
175 (y_t.subs({t_sp: tf}), True))
176 new_orig_th_t = sp.Piecewise((th_t, t_sp < tf),
177 (th_t.subs({t_sp: tf}), True))
178
179 # %% Test scaling robot trajectory
180 new_t_vals = np.linspace(to, tend_new, 300) # New time stamps
181 # Obstacle positions
182 obs_x_vals = np.array([new_obs_x_t.subs({t_sp: tv}) \
183 for tv in new_t_vals], float)
184 obs_y_vals = np.array([new_obs_y_t.subs({t_sp: tv}) \
185 for tv in new_t_vals], float)
186 # Original robot pose
187 orig_x_vals = np.array([new_orig_x_t.subs({t_sp: tv}) \
188 for tv in new_t_vals], float)
189 orig_y_vals = np.array([new_orig_y_t.subs({t_sp: tv}) \
190 for tv in new_t_vals], float)
191 orig_th_vals = np.array([new_orig_th_t.subs({t_sp: tv}) \
192 for tv in new_t_vals], float)
193 # New robot x, y, theta values
194 x_vals = np.array([nx_t.subs({t_sp: tv}) \
195 for tv in new_t_vals], float)
196 y_vals = np.array([ny_t.subs({t_sp: tv}) \
197 for tv in new_t_vals], float)
198 th_vals = np.array([nth_t.subs({t_sp: tv}) \
199 for tv in new_t_vals], float)
200 th_vals[-1] = 0.0 # Precaution (at the end of simulation)
201
202 # %%
203 # Show the figure
204 fig = plt.figure(num="Collision Avoidance", dpi=150)
205 ax = fig.add_subplot()
206 ax.set_aspect('equal')
207 # v_i = 100
208 # if True:
209 for v_i in range(len(new_t_vals)): # FIXME: Don't run in VSCode
210 # Reset animation
211 ax.cla()
212 # Show the obstacle
213 obs_body = patch.Circle((obs_x_vals[v_i], obs_y_vals[v_i]),
214 obs_rad, ec='k', fc="#F06767", zorder=3.6)

```

```

215 # Show the robot (original path with collision)
216 rob_body_o = patch.Circle((orig_x_vals[v_i], orig_y_vals[v_i]),
217     rob_rad, ec='k', fc="#88B4E6", alpha=0.5, zorder=3.4)
218 ax.plot(
219     [orig_x_vals[v_i], orig_x_vals[v_i] + \
220     rob_rad*np.cos(orig_th_vals[v_i])],
221     [orig_y_vals[v_i], orig_y_vals[v_i] + \
222     rob_rad*np.sin(orig_th_vals[v_i])], c="#7AOC7A",
223     zorder=3.45, alpha=0.5)
224 # Show the new robot path (hopefully no collision)
225 rob_body = patch.Circle((x_vals[v_i], y_vals[v_i]),
226     rob_rad, ec='k', fc="#88B4E6", alpha=1, zorder=3.5)
227 ax.plot(
228     [x_vals[v_i], x_vals[v_i] + rob_rad*np.cos(th_vals[v_i])],
229     [y_vals[v_i], y_vals[v_i] + rob_rad*np.sin(th_vals[v_i])],
230     c="#7AOC7A", zorder=3.55)
231 # Add patches
232 ax.add_patch(obs_body)
233 ax.add_patch(rob_body_o)
234 ax.add_patch(rob_body)
235 # Show the paths
236 ax.plot(obs_x_vals, obs_y_vals, alpha=0.5, zorder=3)
237 ax.plot(x_vals, y_vals, alpha=0.5, zorder=3)
238 ax.plot(ox_i, oy_i, 'kx', zorder=3)
239 # Set limits
240 ax.set_xlim(start_pt[0]-5, end_pt[0]+5)
241 ax.set_ylim(start_pt[1]-5, end_pt[1]+5)
242 fig.savefig(f"./out/{v_i}.png")
243 # Pause simulation
244 # plt.pause(0.1)
245
246 # %%

```

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).

```

1 # Collision Cone based constant time scaling
2 """
3     Given a robot trajectory generated through bernstein polynomials
4     (modified to return the position and velocities), and the
5     trajectory of a holonomic obstacle (straight line equation), we
6     alter the robot's velocities (when the robot is in the collision
7     bounds) using collision cones.
8     The script assumes the following:
9     - The scaling is applied to robot's velocities. The scaling 's'
10       is found using collision cone equations (refer PDF submission)
11     - There can be multiple collisions, but the settings should allow
12       time scaling as a viable solution (obstacle shouldn't stop on
13       path)
14
15     Adjust properties in the following sections
16     - ==== User configuration area (robot trajectory) ====
17     - ==== User configuration area (obstacle) ====
18
19 """
20
21 # %% Import everything
22 # Main imports
23 import numpy as np
24 import sympy as sp
25 from matplotlib import pyplot as plt
26 from matplotlib import patches as patch
27 # For trajectory generation
28 from lib.three_point_traj_planner import NonHoloThreePtBernstein
29 # Utilities
30 import time
31 from tqdm import tqdm
32
33 # %%

```

```

34
35 # %%
36
37 # %% Experimental section
38 # Generate a random robot trajectory
39 # ==== Begin: User configuration area (robot trajectory) ====
40 # Points as [x, y]
41 start_pt = [0, 0]
42 end_pt = [40, 40]
43 way_pt = [20, 25]
44 # Time values
45 to, tw, tf = [0., 25., 50.] # Start, waypoint, end
46 # Other parameters
47 ko, kw, kf = [0, np.tan(np.pi/4), 0] # k = np.tan(theta)
48 dko, dkw, dkf = [0, 0, 0]
49 dxo, dxw, dxf = [0, 1, 0]
50 # ==== End: User configuration area (robot trajectory) ====
51 # Convert to dictionary (for library)
52 constraint_dict = {
53     "to": to, "tw": tw, "tf": tf,
54     "xo": start_pt[0], "xw": way_pt[0], "xf": end_pt[0],
55     "yo": start_pt[1], "yw": way_pt[1], "yf": end_pt[1],
56     "ko": ko, "kw": kw, "kf": kf,
57     "dxo": dxo, "dxw": dxw, "dxf": dxf,
58     "dko": dko, "dkw": dkw, "dkf": dkf
59 }
60 # Initialize solver
61 path_solver = NonHoloThreePtBernstein()
62 # Time symbol
63 t_sp = sp.symbols('t', real=True, positive=True)
64 t_all = sp.symbols('t') # Generic time symbol (used by functions)
65 print("Finding path")
66 # Solve for paths
67 x_vals, y_vals, th_vals, t_vals, x_t, y_t, th_t = \
68     path_solver.solve_wpt_constr(constraint_dict)
69 print("Path found")
70 # Substitute 't' with real and positive 't' (time substitution)
71 x_t = x_t.subs({t_all: t_sp})
72 y_t = y_t.subs({t_all: t_sp})
73 th_t = th_t.subs({t_all: t_sp})
74 # Plot trajectories
75 plt.figure()
76 plt.title("XY plot")
77 plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
78 plt.colorbar()
79 plt.show()
80
81 # %% Collision with an obstacle
82 # ==== Begin: User configuration area (obstacle) ====
83 obs_t_col = 25 # Time of collision (for x, y intermediate)
84 obs_start = (5, 35) # (x, y): Starting point of obstacle
85 obs_rad = 1 # Obstacle radius
86 rob_rad = 2 # Robot radius
87 detection_bound = 7 # Sensor for collision check (else scale = 1)
88 num_sim_samples = 300 # Number of time steps (not for saving!)
89 ks_val_min = 0.15 # Minimum scaling factor
90 # ==== End: User configuration area (obstacle) ====
91 # Location where collision will take place
92 ox_i = float(x_t.subs({t_sp: obs_t_col}))
93 oy_i = float(y_t.subs({t_sp: obs_t_col}))
94 obs_x_t = obs_start[0] + ((ox_i - obs_start[0])/obs_t_col) * t_sp
95 obs_y_t = obs_start[1] + ((oy_i - obs_start[1])/obs_t_col) * t_sp
96 # Time, x, y trajectories (array) - visualize
97 obs_t_vals = t_vals.copy() # np.linspace(to, tf, 100)
98 obs_x_vals = np.array([obs_x_t.subs({t_sp: tv}) \
99     for tv in obs_t_vals])
100 obs_y_vals = np.array([obs_y_t.subs({t_sp: tv}) \
101     for tv in obs_t_vals])
102
103 # %% Show the collision
104 plt.figure()
105 plt.title("XY plot")
106 plt.scatter(x_vals, y_vals, 1.0, c=t_vals)

```

```

107 plt.scatter(obs_x_vals, obs_y_vals, 1.0, c=t_vals)
108 plt.colorbar()
109 plt.show()
110
111 # %% Prepare the system - Collision Cone
112 x1, x2, y1, y2 = sp.symbols(r"x1, x2, y1, y2", real=True)
113 dx1, dy1 = sp.symbols(r"\dot{x}_1, \dot{y}_1", real=True)
114 dx2, dy2 = sp.symbols(r"\dot{x}_2, \dot{y}_2", real=True)
115 R = sp.symbols(r"R", real=True, positive=True)
116 s = sp.symbols(r"s", real=True)
117 # Modify LHS
118 ineq_lhs_orig = (x1-x2)**2 + (y1-y2)**2 - R**2 - \
119     (((s*dx1-dx2)*(x1-x2) + (s*dy1-dy2)*(y1-y2))**2 / \
120     ((s*dx1-dx2)**2 + (s*dy1-dy2)**2))
121 ineq_lhs = ineq_lhs_orig * ((s*dx1-dx2)**2 + (s*dy1-dy2)**2)
122 # Get coefficients
123 ineq_lhs_poly = sp.Poly(ineq_lhs.apart(s), s)
124 all_coeffs = ineq_lhs_poly.all_coeffs()
125 # a*s**2 + b*s + c
126 a, b, c = all_coeffs
127 assert a*s**2 + b*s + c == sp.simplify(ineq_lhs).apart(s)
128 r1 = (-b - (b**2 - 4*a*c)**0.5)/(2*a) # Root 1
129 r2 = (-b + (b**2 - 4*a*c)**0.5)/(2*a) # Root 2
130 d = b**2 - 4*a*c # Discriminant, should be > 0
131
132 # %% Main simulation loop (with collision avoidance)
133 start_ctime = time.time() # Start computer time
134 # Declare velocities of robot
135 vx_t = x_t.diff(t_sp)
136 vy_t = y_t.diff(t_sp) # Get theta from velocities
137 # Declare velocities of obstacle
138 ovx_t = obs_x_t.diff(t_sp)
139 ovy_t = obs_y_t.diff(t_sp)
140 # Time values for simulation
141 t_sim_start, t_sim_end = to, tf
142 dt_sim_k1 = (t_sim_end - t_sim_start)/num_sim_samples
143 t_sim = t_sim_start # Current simulation time
144 # t_sim = 20 # Random start sim time # FIXME: Remove this!
145 t_rob_local = t_sim # Time for robot's tracking (ONLY IN SIM!)
146 dt_sim = dt_sim_k1 # Currently, scaling = 1
147 k_val = 1.0 # Value of scaling constant (for all steps)
148 # Pose vectors for the robot and obstacle
149 r_robot = [float(x_t.subs(t_sp, t_sim)),
150            float(y_t.subs(t_sp, t_sim))]
151 th_robot = float(th_t.subs(t_sp, t_sim))
152 r_obstacle = [float(obs_x_t.subs(t_sp, t_sim)),
153               float(obs_y_t.subs(t_sp, t_sim))]
154 # Logging variables (all time in t_sim)
155 robot_poses = [] # [time, x, y, theta] of the robot
156 obstacle_poses = [] # [time, x, y] of the obstacle
157 k_vals = [] # [time, k_val] - Log time scaling factor
158 dist_vals = [] # [time, dist_rob_obs] - Robot to obstacle
159 time_vals = [] # [time, t_robot_local] - Robot time (prop)
160 # Simulation progress bar (for robot local time)
161 tq_bar = tqdm(total=t_sim_end, leave=False)
162 # Start simulation
163 while t_rob_local < t_sim_end:
164     # Distance between robot and obstacle ('r' vector)
165     dist_ro = float(((r_robot[0] - r_obstacle[0])**2 + \
166                     (r_robot[1] - r_obstacle[1])**2)**0.5)
167     # Time scaling IFF there is a collision (in detection)
168     if dist_ro < detection_bound:
169         # Values which can be substituted
170         subs_sh = {
171             R: rob_rad + obs_rad, # Dialated obstacle radius
172             # Robot pose
173             x1: r_robot[0], y1: r_robot[1],
174             # Robot velocity - in local time (keep track!)
175             dx1: vx_t.subs(t_sp, t_rob_local),
176             dy1: vy_t.subs(t_sp, t_rob_local),
177             # Obstacle pose
178             x2: r_obstacle[0], y2: r_obstacle[1],
179             # Obstacle velocity

```

```

180         dx2: ovx_t.subs(t_sp, t_sim),
181         dy2: ovy_t.subs(t_sp, t_sim)
182     }
183     # See if 'a' > 0 (for parabola solutions)
184     if float(a.subs(subs_sh)) > 0:
185         # See if 'discriminator' is < 0
186         if float(d.subs(subs_sh)) < 0:
187             # Entire range is okay, pick the minimum value
188             k_val = ks_val_min
189         else: # 'discriminator' > 0 (roots exist)
190             # s = min{ [(-inf, r1) U (r2, inf)] N [s_min, inf] }
191             r1_val = float(r1.subs(subs_sh))
192             r2_val = float(r2.subs(subs_sh))
193             if r1_val < ks_val_min: # 'r1' doesn't matter
194                 k_val = max(r2_val, ks_val_min)
195             else: # 'r2' doesn't matter
196                 # {S_vals} = (s_min, r1); take min
197                 k_val = ks_val_min
198     else: # 'a' < 0 here
199         # See if 'discriminator' is < 0
200         if float(d.subs(subs_sh)) < 0:
201             # No solution for poly > 0 possible (all -ve vals)
202             raise Exception("Time scaling not possible")
203         else: # 'discriminator' > 0
204             # s = min{ [r1, r2] N [s_min, inf] }
205             r1_val = float(r1.subs(subs_sh))
206             r2_val = float(r2.subs(subs_sh))
207             if r2_val < ks_val_min: # {S_vals} = NULL!
208                 # raise Exception("Time scaling not possible")
209                 # Fingers crossed: Hopefully no collision!
210                 k_val = ks_val_min
211             else:
212                 k_val = max(r1_val, ks_val_min)
213     else: # Not in detection bounds, don't bother scaling
214         k_val = 1.0
215     # Continue robot simulation with k_val (float) scaling
216     # Using velocities, progress the next states
217     r_obstacle = [ # Use real time for obstacle updates
218         float(r_obstacle[0] + ovx_t.subs(t_sp, t_sim) * dt_sim),
219         float(r_obstacle[1] + ovy_t.subs(t_sp, t_sim) * dt_sim),
220     ]
221     robot_dx = float(k_val * vx_t.subs(t_sp, t_rob_local) * dt_sim)
222     robot_dy = float(k_val * vy_t.subs(t_sp, t_rob_local) * dt_sim)
223     r_robot = [
224         float(r_robot[0] + robot_dx), float(r_robot[1] + robot_dy)
225     ]
226     th_robot = np.arctan2(robot_dy, robot_dx)
227     # Log these values
228     robot_poses.append([t_sim, r_robot[0], r_robot[1], th_robot])
229     obstacle_poses.append([t_sim, r_obstacle[0], r_obstacle[1]])
230     k_vals.append([t_sim, k_val])
231     dist_vals.append([t_sim, dist_ro])
232     time_vals.append([t_sim, t_rob_local])
233     # Change in time
234     t_rob_local += k_val * dt_sim # Time scale the robot
235     t_sim += dt_sim # The simulation proceeds
236     tq_bar.update(k_val * dt_sim)
237 tq_bar.close()
238 # Convert all logs to numpy arrays
239 robot_poses = np.array(robot_poses, float) # [time, x, y, theta]
240 obstacle_poses = np.array(obstacle_poses, float) # [time, x, y]
241 k_vals = np.array(k_vals, float) # [time, k_val]
242 dist_vals = np.array(dist_vals, float) # [time, dist_rob_obs]
243 time_vals = np.array(time_vals, float) # [time, t_robot_local]
244 end_ctime = time.time() # End computer time
245 print(f"Simulation took {end_ctime - start_ctime:.3f} seconds!")
246
247 # %% Get all trajectories (with time clipping)
248 # Time values
249 res_tvals = time_vals[:, 0]
250 # Robot avoiding collision
251 res_robot_poses = robot_poses[:, 1:4] # [x, y, theta]
252 # Obstacle path

```



```

253 res_obsposes_x = np.array([obs_x_t.subs(t_sp, min(tv, tf)) \
254     for tv in res_tvals], float)
255 res_obsposes_y = np.array([obs_y_t.subs(t_sp, min(tv, tf)) \
256     for tv in res_tvals], float)
257 res_obsposes = np.stack([res_obsposes_x, res_obsposes_y]).T
258 # Robot (with collision)
259 res_crobotposes_x = np.array([x_t.subs(t_sp, min(tv, tf)) \
260     for tv in res_tvals], float)
261 res_crobotposes_y = np.array([y_t.subs(t_sp, min(tv, tf)) \
262     for tv in res_tvals], float)
263 res_crobotposes_th = np.array([th_t.subs(t_sp, min(tv, tf)) \
264     for tv in res_tvals], float)
265 res_crobotposes = np.stack([res_crobotposes_x, res_crobotposes_y,
266     res_crobotposes_th]).T
267 # Fix the last angle
268 res_robposes[-1, 2] = res_crobotposes[-1, 2] # Theta fix
269 # Distance between robot and obstacle
270 res_cdist = np.linalg.norm(res_crobotposes[:, 0:2] - \
271     res_obsposes[:, 0:2], axis=1)
272 res_dist = np.linalg.norm(res_robposes[:, 0:2] - \
273     res_obsposes[:, 0:2], axis=1)
274 print(f"Processed {res_tvals.shape[0]} time samples")
275
276 # %%
277 # Show the time
278 plt.figure(figsize=(7, 3))
279 plt.suptitle("Time and scale")
280 plt.subplot(1,2,1)
281 plt.title("Time")
282 plt.xlabel("Simulation")
283 plt.ylabel("Robot")
284 plt.plot(time_vals[:, 0], time_vals[:, 1], '-.')
285 plt.subplot(1,2,2)
286 plt.title("Scaling factor")
287 plt.xlabel("Simulation")
288 plt.plot(k_vals[:, 0], k_vals[:, 1], '-.')
289 plt.tight_layout()
290 plt.show()
291 # Show the robot trajectory (avoiding collision)
292 plt.figure(figsize=(10, 10))
293 plt.suptitle("Time scaled trajectory")
294 plt.subplot(3,2,1)
295 plt.title("X")
296 plt.plot(res_tvals, res_robposes[:, 0], 'r-', label="Modified")
297 plt.plot(res_tvals, res_crobotposes[:, 0], 'r--', label="Actual")
298 plt.legend()
299 plt.subplot(3,2,3)
300 plt.title("Y")
301 plt.plot(res_tvals, res_robposes[:, 1], 'g-', label="Modified")
302 plt.plot(res_tvals, res_crobotposes[:, 1], 'g--', label="Actual")
303 plt.legend()
304 plt.subplot(3,2,5)
305 plt.title(r"$\theta$")
306 plt.plot(res_tvals, res_robposes[:, 2], 'b-', label="Modified")
307 plt.plot(res_tvals, res_crobotposes[:, 2], 'b--', label="Actual")
308 plt.legend()
309 # Obstacle trajectory
310 plt.subplot(3,2,2)
311 plt.title("Obstacle - X")
312 plt.plot(res_tvals, res_obsposes[:, 0], 'r-')
313 plt.subplot(3,2,4)
314 plt.title("Obstacle - Y")
315 plt.plot(res_tvals, res_obsposes[:, 1], 'g-')
316 plt.subplot(3,2,6)
317 plt.title("Distance")
318 plt.plot(res_tvals, res_dist, '-', label="Modified")
319 plt.plot(res_tvals, res_cdist, '--', label="Actual")
320 plt.axhline(obs_rad + rob_rad, ls='--', c='r')
321 plt.axhline(detection_bound, ls=':', c='r')
322 plt.legend()
323 # Show the plot
324 plt.tight_layout()
325 plt.show()

```

```

326
327 # %% Show as video
328 # Show the figure
329 fig = plt.figure(num="Collision Avoidance", dpi=150)
330 ax = fig.add_subplot()
331 ax.set_aspect('equal')
332 # v_i = 140
333 # if True:
334 for v_i in tqdm(range(len(res_tvalls))): # FIXME: Don't run in VSCode
335     # Reset animation
336     ax.cla()
337     # Show the obstacle
338     obs_body = patch.Circle(
339         (res_obsposes[v_i, 0], res_obsposes[v_i, 1]),
340         obs_rad, ec='k', fc="#F06767", zorder=3.6)
341     # Show the robot (original path with collision)
342     rob_body_o = patch.Circle(
343         (res_crobotposes[v_i, 0], res_crobotposes[v_i, 1]),
344         rob_rad, ec='k', fc="#88B4E6", alpha=0.5, zorder=3.4)
345     ax.plot(
346         [res_crobotposes[v_i, 0], res_crobotposes[v_i, 0] + \
347          rob_rad*np.cos(res_crobotposes[v_i, 2])],
348         [res_crobotposes[v_i, 1], res_crobotposes[v_i, 1] + \
349          rob_rad*np.sin(res_crobotposes[v_i, 2])], c="#7A0C7A",
350         zorder=3.45, alpha=0.5)
351     # Show the new robot path (hopefully no collision)
352     if abs(k_vals[v_i, 1] - 1.0) > 1e-3: # TS active
353         rb_ec = 'r'
354         # Line joining robot and obstacle
355         ax.plot([res_robposes[v_i, 0], res_obsposes[v_i, 0]],
356                [res_robposes[v_i, 1], res_obsposes[v_i, 1]], c='r',
357                lw=0.2, zorder=3.65) # Above robot and obstacle
358     else:
359         rb_ec = 'k'
360     rob_body = patch.Circle(
361         (res_robposes[v_i, 0], res_robposes[v_i, 1]),
362         rob_rad, ec=rb_ec, fc="#88B4E6", alpha=1, zorder=3.5)
363     ax.plot(
364         [res_robposes[v_i, 0], res_robposes[v_i, 0] + \
365          rob_rad*np.cos(res_robposes[v_i, 2])],
366         [res_robposes[v_i, 1], res_robposes[v_i, 1] + \
367          rob_rad*np.sin(res_robposes[v_i, 2])],
368         c="#7A0C7A", zorder=3.55)
369     # Add patches
370     ax.add_patch(obs_body)
371     ax.add_patch(rob_body_o)
372     ax.add_patch(rob_body)
373     # Show the paths
374     ax.plot(res_obsposes[:, 0], res_obsposes[:, 1], alpha=0.5,
375            zorder=3)
376     ax.plot(res_robposes[:, 0], res_robposes[:, 1], alpha=0.5,
377            zorder=3)
378     # Location where the collision will take place
379     ax.plot(ox_i, oy_i, 'kx', zorder=3)
380     # Set limits
381     ax.set_xlim(start_pt[0]-5, end_pt[0]+5)
382     ax.set_ylim(start_pt[1]-5, end_pt[1]+5)
383     # Show/store result
384     fig.savefig(f"./out/{v_i}.png") # Use for saving everything
385     # plt.pause(0.05) # Use only for python script
386     # plt.show() # Use only for VSCode
387
388 # %%
389

```

C Rule-based Linear Time Scaling

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

```

1 # Rule-based linear time scaling
2 """

```

```

3      Given a robot trajectory generated through bernstein polynomials
4      (modified to return the position and velocities), and the
5      trajectory of a holonomic obstacle (straight line equation), we
6      alter the robot's velocities (when the robot is in the collision
7      bounds) using a user-defined linear time scaling approach.
8      - The scaling is applied to robot's velocities. The scaling 's'
9        is given by 's(t) = a + b*t' where 't' is the simulation time
10     - For now, the script has been tested only in single collision
11         case
12     """
13
14     # %% Import everything
15     # Main imports
16     import numpy as np
17     import sympy as sp
18     from matplotlib import pyplot as plt
19     from matplotlib import patches as patch
20     # For trajectory generation
21     from lib.three_point_traj_planner import NonHoloThreePtBernstein
22     # Utilities
23     import time
24     from tqdm import tqdm
25
26     # %%
27
28     # %%
29
30     # %% Experimental section
31     # Generate a random robot trajectory
32     # ==== Begin: User configuration area (robot trajectory) ====
33     # Points as [x, y]
34     start_pt = [0, 0]
35     end_pt = [50, 45]
36     way_pt = [20, 25]
37     # Time values
38     to, tw, tf = [0., 25., 50.]      # Start, waypoint, end
39     # Other parameters
40     ko, kw, kf = [0, np.tan(np.pi/4), 0]      # k = np.tan(theta)
41     dko, dkw, dkf = [0, 0, 0]
42     dxo, dxw, dxf = [0, 1, 0]
43     # ==== End: User configuration area (robot trajectory) ====
44     # Convert to dictionary (for library)
45     constraint_dict = {
46         "to": to, "tw": tw, "tf": tf,
47         "xo": start_pt[0], "xw": way_pt[0], "xf": end_pt[0],
48         "yo": start_pt[1], "yw": way_pt[1], "yf": end_pt[1],
49         "ko": ko, "kw": kw, "kf": kf,
50         "dxo": dxo, "dxw": dxw, "dxf": dxf,
51         "dko": dko, "dkw": dkw, "dkf": dkf
52     }
53     # Initialize solver
54     path_solver = NonHoloThreePtBernstein()
55     # Time symbol
56     t_sp = sp.symbols('t', real=True, positive=True)
57     t_all = sp.symbols('t') # Generic time symbol (used by functions)
58     print("Finding path")
59     # Solve for paths
60     x_vals, y_vals, th_vals, t_vals, x_t, y_t, th_t = \
61         path_solver.solve_wpt_constr(constraint_dict)
62     print("Path found")
63     # Substitute 't' with real and positive 't' (time substitution)
64     x_t = x_t.subs({t_all: t_sp})
65     y_t = y_t.subs({t_all: t_sp})
66     th_t = th_t.subs({t_all: t_sp})
67     # Plot trajectories
68     plt.figure()
69     plt.title("XY plot")
70     plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
71     plt.colorbar()
72     plt.show()
73
74     # %% Collision with an obstacle
75     # ==== Begin: User configuration area (obstacle) ====

```

```

76 obs_t_col = 15      # Time of collision (for x, y intermediate)
77 obs_start = (4, 20) # (x, y): Starting point of obstacle
78 obs_rad = 1        # Obstacle radius
79 rob_rad = 2.5       # Robot radius
80 detection_bound = 10 # Sensor for collision check (else scale = 1)
81 # s_func = _a + _b * t -> Functions for 's' (scaling term). t is sim.
82 sfunc_a = 0.001     # Constant term for s_func
83 sfunc_b = 0.02       # Time term for s_func
84 num_sim_samples = 300 # Number of time steps (not for saving!)
85 # ==== End: User configuration area (obstacle) ====
86 # Location where collision will take place
87 ox_i = float(x_t.subs({t_sp: obs_t_col}))
88 oy_i = float(y_t.subs({t_sp: obs_t_col}))
89 obs_x_t = obs_start[0] + ((ox_i - obs_start[0])/obs_t_col) * t_sp
90 obs_y_t = obs_start[1] + ((oy_i - obs_start[1])/obs_t_col) * t_sp
91 # Time, x, y trajectories (array) - visualize
92 obs_t_vals = t_vals.copy() # np.linspace(to, tf, 100)
93 obs_x_vals = np.array([obs_x_t.subs({t_sp: tv}) \
94     for tv in obs_t_vals])
95 obs_y_vals = np.array([obs_y_t.subs({t_sp: tv}) \
96     for tv in obs_t_vals])
97
98 # %% Show the collision
99 plt.figure()
100 plt.title("XY plot")
101 plt.scatter(x_vals, y_vals, 1.0, c=t_vals)
102 plt.scatter(obs_x_vals, obs_y_vals, 1.0, c=t_vals)
103 plt.colorbar()
104 plt.show()
105
106 # %% Main simulation loop (with collision avoidance)
107 start_ctime = time.time() # Start computer time
108 # Declare velocities of robot
109 vx_t = x_t.diff(t_sp)
110 vy_t = y_t.diff(t_sp) # Get theta from velocities
111 # Declare velocities of obstacle
112 ovx_t = obs_x_t.diff(t_sp)
113 ovy_t = obs_y_t.diff(t_sp)
114 # Time values for simulation
115 t_sim_start, t_sim_end = to, tf
116 dt_sim_k1 = (t_sim_end - t_sim_start)/num_sim_samples
117 t_sim = t_sim_start # Current simulation time
118 # t_sim = 20 # Random start sim time # FIXME: Remove this!
119 t_rob_local = t_sim # Time for robot's tracking (ONLY IN SIM!)
120 dt_sim = dt_sim_k1 # Currently, scaling = 1
121 k_val = 1.0 # Value of scaling constant (for all steps)
122 # Pose vectors for the robot and obstacle
123 r_robot = [float(x_t.subs(t_sp, t_sim)),
124     float(y_t.subs(t_sp, t_sim))]
125 th_robot = float(th_t.subs(t_sp, t_sim))
126 r_obstacle = [float(obs_x_t.subs(t_sp, t_sim)),
127     float(obs_y_t.subs(t_sp, t_sim))]
128 # Logging variables (all time in t_sim)
129 robot_poses = [] # [time, x, y, theta] of the robot
130 obstacle_poses = [] # [time, x, y] of the obstacle
131 k_vals = [] # [time, k_val] - Log time scaling factor
132 dist_vals = [] # [time, dist_rob_obs] - Robot to obstacle
133 time_vals = [] # [time, t_robot_local] - Robot time (prop)
134 # Simulation progress bar (for robot local time)
135 tq_bar = tqdm(total=t_sim_end, leave=False)
136 # Start simulation
137 while t_rob_local < t_sim_end:
138     # Distance between robot and obstacle ('r' vector)
139     dist_ro = float(((r_robot[0] - r_obstacle[0])**2 + \
140         (r_robot[1] - r_obstacle[1])**2)**0.5)
141     if dist_ro < detection_bound:
142         # Linear time scaling function for scaling factor
143         k_val = sfunc_a + sfunc_b * t_sim
144     else:
145         k_val = 1.0
146     # Continue robot simulation with k_val (float) scaling
147     # Using velocities, progress the next states
148     r_obstacle = [ # Use real time for obstacle updates

```

```

149         float(r_obstacle[0] + ovx_t.subs(t_sp, t_sim) * dt_sim),
150         float(r_obstacle[1] + ovy_t.subs(t_sp, t_sim) * dt_sim),
151     ]
152     robot_dx = float(k_val * vx_t.subs(t_sp, t_rob_local) * dt_sim)
153     robot_dy = float(k_val * vy_t.subs(t_sp, t_rob_local) * dt_sim)
154     r_robot = [
155         float(r_robot[0] + robot_dx), float(r_robot[1] + robot_dy)
156     ]
157     th_robot = np.arctan2(robot_dy, robot_dx)
158     # Log these values
159     robot_poses.append([t_sim, r_robot[0], r_robot[1], th_robot])
160     obstacle_poses.append([t_sim, r_obstacle[0], r_obstacle[1]])
161     k_vals.append([t_sim, k_val])
162     dist_vals.append([t_sim, dist_ro])
163     time_vals.append([t_sim, t_rob_local])
164     # Change in time
165     t_rob_local += k_val * dt_sim # Time scale the robot
166     t_sim += dt_sim # The simulation proceeds
167     tq_bar.update(k_val * dt_sim)
168 tq_bar.close()
169 # Convert all logs to numpy arrays
170 robot_poses = np.array(robot_poses, float) # [time, x, y, theta]
171 obstacle_poses = np.array(obstacle_poses, float) # [time, x, y]
172 k_vals = np.array(k_vals, float) # [time, k_val]
173 dist_vals = np.array(dist_vals, float) # [time, dist_rob_obs]
174 time_vals = np.array(time_vals, float) # [time, t_robot_local]
175 end_ctime = time.time() # End computer time
176 print(f"Simulation took {end_ctime - start_ctime:.3f} seconds!")
177
178 # %% Get all trajectories (with time clipping)
179 # Time values
180 res_tvals = time_vals[:, 0]
181 # Robot avoiding collision
182 res_robotposes = robot_poses[:, 1:4] # [x, y, theta]
183 # Obstacle path
184 res_obsposes_x = np.array([obs_x_t.subs(t_sp, min(tv, tf)) \
185     for tv in res_tvals], float)
186 res_obsposes_y = np.array([obs_y_t.subs(t_sp, min(tv, tf)) \
187     for tv in res_tvals], float)
188 res_obsposes = np.stack([res_obsposes_x, res_obsposes_y]).T
189 # Robot (with collision)
190 res_crobotposes_x = np.array([x_t.subs(t_sp, min(tv, tf)) \
191     for tv in res_tvals], float)
192 res_crobotposes_y = np.array([y_t.subs(t_sp, min(tv, tf)) \
193     for tv in res_tvals], float)
194 res_crobotposes_th = np.array([th_t.subs(t_sp, min(tv, tf)) \
195     for tv in res_tvals], float)
196 res_crobotposes = np.stack([res_crobotposes_x, res_crobotposes_y,
197     res_crobotposes_th]).T
198 # Fix the last angle
199 res_robotposes[-1, 2] = res_crobotposes[-1, 2] # Theta fix
200 # Distance between robot and obstacle
201 res_cdist = np.linalg.norm(res_crobotposes[:, 0:2] - \
202     res_obsposes[:, 0:2], axis=1)
203 res_dist = np.linalg.norm(res_robotposes[:, 0:2] - \
204     res_obsposes[:, 0:2], axis=1)
205 print(f"Processed {res_tvals.shape[0]} time samples")
206
207 # %%
208 # Show the time
209 plt.figure(figsize=(7, 3))
210 plt.suptitle("Time and scale")
211 plt.subplot(1,2,1)
212 plt.title("Time")
213 plt.xlabel("Simulation")
214 plt.ylabel("Robot")
215 plt.plot(time_vals[:, 0], time_vals[:, 1], '-.')
216 plt.subplot(1,2,2)
217 plt.title("Scaling factor")
218 plt.xlabel("Simulation")
219 plt.plot(k_vals[:, 0], k_vals[:, 1], '-.')
220 plt.tight_layout()
221 plt.show()

```

```

222 # Show the robot trajectory (avoiding collision)
223 plt.figure(figsize=(10, 10))
224 plt.suptitle("Time scaled trajectory")
225 plt.subplot(3,2,1)
226 plt.title("X")
227 plt.plot(res_tvals, res_robposes[:, 0], 'r-', label="Modified")
228 plt.plot(res_tvals, res_crobotposes[:, 0], 'r--', label="Actual")
229 plt.legend()
230 plt.subplot(3,2,3)
231 plt.title("Y")
232 plt.plot(res_tvals, res_robposes[:, 1], 'g-', label="Modified")
233 plt.plot(res_tvals, res_crobotposes[:, 1], 'g--', label="Actual")
234 plt.legend()
235 plt.subplot(3,2,5)
236 plt.title(r"$\theta$")
237 plt.plot(res_tvals, res_robposes[:, 2], 'b-', label="Modified")
238 plt.plot(res_tvals, res_crobotposes[:, 2], 'b--', label="Actual")
239 plt.legend()
240 # Obstacle trajectory
241 plt.subplot(3,2,2)
242 plt.title("Obstacle - X")
243 plt.plot(res_tvals, res_obsposes[:, 0], 'r-')
244 plt.subplot(3,2,4)
245 plt.title("Obstacle - Y")
246 plt.plot(res_tvals, res_obsposes[:, 1], 'g-')
247 plt.subplot(3,2,6)
248 plt.title("Distance")
249 plt.plot(res_tvals, res_dist, '-', label="Modified")
250 plt.plot(res_tvals, res_cdist, '--', label="Actual")
251 plt.axhline(obs_rad + rob_rad, ls='--', c='r')
252 plt.axhline(detection_bound, ls=':', c='r')
253 plt.legend()
254 # Show the plot
255 plt.tight_layout()
256 plt.show()
257
258 # %% Show as video
259 # Show the figure
260 fig = plt.figure(num="Collision Avoidance", dpi=150)
261 ax = fig.add_subplot()
262 ax.set_aspect('equal')
263 # v_i = 140
264 # if True:
265 for v_i in tqdm(range(len(res_tvals))): # FIXME: Don't run in VSCode
266     # Reset animation
267     ax.cla()
268     # Show the obstacle
269     obs_body = patch.Circle(
270         (res_obsposes[v_i, 0], res_obsposes[v_i, 1]),
271         obs_rad, ec='k', fc="#F06767", zorder=3.6)
272     # Show the robot (original path with collision)
273     rob_body_o = patch.Circle(
274         (res_crobotposes[v_i, 0], res_crobotposes[v_i, 1]),
275         rob_rad, ec='k', fc="#88B4E6", alpha=0.5, zorder=3.4)
276     ax.plot(
277         [res_crobotposes[v_i, 0], res_crobotposes[v_i, 0] + \
278          rob_rad*np.cos(res_crobotposes[v_i, 2])],
279         [res_crobotposes[v_i, 1], res_crobotposes[v_i, 1] + \
280          rob_rad*np.sin(res_crobotposes[v_i, 2])], c="#7A0C7A",
281         zorder=3.45, alpha=0.5)
282     # Show the new robot path (hopefully no collision)
283     if abs(k_vals[v_i, 1] - 1.0) > 1e-3: # TS active
284         rb_ec = 'r'
285         # Line joining robot and obstacle
286         ax.plot([res_robposes[v_i, 0], res_obsposes[v_i, 0]],
287                [res_robposes[v_i, 1], res_obsposes[v_i, 1]], c='r',
288                lw=0.2, zorder=3.65) # Above robot and obstacle
289     else:
290         rb_ec = 'k'
291     rob_body = patch.Circle(
292         (res_robposes[v_i, 0], res_robposes[v_i, 1]),
293         rob_rad, ec=rb_ec, fc="#88B4E6", alpha=1, zorder=3.5)
294     ax.plot(

```

```

295         [res_robposes[v_i, 0], res_robposes[v_i, 0] + \
296           rob_rad*np.cos(res_robposes[v_i, 2])],
297         [res_robposes[v_i, 1], res_robposes[v_i, 1] + \
298           rob_rad*np.sin(res_robposes[v_i, 2])],
299         c="#7A0C7A", zorder=3.55)
300     # Add patches
301     ax.add_patch(obs_body)
302     ax.add_patch(rob_body_o)
303     ax.add_patch(rob_body)
304     # Show the paths
305     ax.plot(res_obsposes[:, 0], res_obsposes[:, 1], alpha=0.5,
306             zorder=3)
307     ax.plot(res_robposes[:, 0], res_robposes[:, 1], alpha=0.5,
308             zorder=3)
309     # Location where the collision will take place
310     ax.plot(ox_i, oy_i, 'kx', zorder=3)
311     # Set limits
312     ax.set_xlim(start_pt[0]-5, end_pt[0]+5)
313     ax.set_ylim(start_pt[1]-5, end_pt[1]+5)
314     # Show/store result
315     fig.savefig(f"./out/{v_i}.png")    # Use for saving everything
316     # plt.pause(0.05)    # Use only for python script
317     # plt.show()        # Use only for VSCode
318
319
320 # %%

```