CS 549: Homework #2

November 5, 2023

E. Batuhan Kaynak & **Berat Biçer**

Problem 1

In this section, we present our pathfinding algorithm for the robot base. As instructed, the algorithm is composed of three parts. We first determine the end-point configuration as an inverse-kinematics (IK) solution where the robot base and the target area overlaps. To do so, we define the following Markov optimization problem (KOMO):

```python
# IK - get the goal position
q_0 = config.getJointState()
komo = ry.KOMO(config, phases=1, slicesPerPhase=1, kOrder=0, enableCollisions=False)
komo.addObjective(times=[1], feature=ry.FS.positionDiff,
            frames=["base", "goal_area"], type=ry.OT.sos, scale=[1], target=0)
ret = ry.NLP_Solver(komo.nlp()).setOptions(stopTolerance=1e-2, verbose=0).solve()
q_target = komo.getPath()[0]
```

This KOMO instance defines an objective function as the positional distance between the robot base and the goal area. Since we are only interested in the endpoint configuration at this stage, the solution to the IK solution is obtained in a single step. This is defined as q_{target}.

Next, we determine the initial solution of the pathfinding problem using a *ry.PathFinder()* instance. The determined trajectory is suboptimal, however, it is one feasible solution to the IK problem defined above:

```python
1  # Determine the initial trajectory
2  while True:
3      rrt = ry.PathFinder()
4      rrt.setProblem(config, [q_0], [q_target])
5      ret = rrt.solve()
6      if ret.feasible == True:
7          break
8      del rrt
9
10 # Display Initial Solution
11 for t in range(ret.x.shape[0]-1):
12     config.setJointState(ret.x[t])
13     config.view(False, f"waypoint {t}")
14     time.sleep(0.01)
15 config.setJointState(q_0)
```

In this snippet, line (3) defines the pathfinder instance and with line (4), we instantiate the problem of finding a feasible path from initial state q_0 to q_{target}. We observe that the pathfinder avoids collisions without explicitly punishing collisions in the KOMO instance. So, related objective function enhancements are removed from this KOMO instance when determining the initial IK solutions for the endpoint position of the robot base. We further highlight that the pathfinder occasionally rarely determines a feasible solution on the first try, thus, we loop until a feasible solution is obtained. This process terminates, in general, after a couple of attempts and returns our initial trajectory.

Lastly, we refine the initial trajectory for the robot base by defining a second KOMO problem. This KOMO instance is similar to the first one, except it is enhanced by restricting the number of collisions the path

Problem 1 continued on next page...

may contain to be zero, forcing the algorithm to find a path around the maze. Further, we add a first-order control objective which punishes the squared distance between consecutive positions of the robot base, which smooths out the trajectory. The corresponding snippet is as follows:

```python
# Trajectory optimization
komo = ry.KOMO(config, phases=1, slicesPerPhase=ret.x.shape[0],
                kOrder=2, enableCollisions=True)
komo.addObjective(times=[], feature=ry.FS.accumulatedCollisions,
                frames=[], type=ry.OT.eq, scale=[1], target=0)
komo.addObjective(times=[1], feature=ry.FS.positionDiff,
                frames=["base", "goal_area"], type=ry.OT.sos, scale=[1], target=0)
komo.addControlObjective(times=[], order=1, scale=1.2e-1)
komo.initWithPath_qOrg(ret.x)
ret2 = ry.NLP_Solver(komo.nlp()).setOptions(stopTolerance=1e-2, verbose=0).solve()

# Display Optimized Solution
ret2_x = ret2.x.reshape(-1,3)
for t in range(len(ret2_x)-1):
    config.setJointState(ret2_x[t])
    config.view(False, f"waypoint {t}")
    time.sleep(0.01)
```

Note that the initial trajectory is used to initialize the KOMO instance by *komo.initWithPath_qOrg()* function.

Here's a link to a video clip showing the effect of trajectory optimization. Notice that the initial path sways in random directions many times, whereas with the enhanced KOMO solution the optimized trajectory follows a smooth path around the maze. Further increasing the penalty (scale) of the control objective forces the base to travel closer to the maze's inner boundaries, but when this value goes above a certain threshold, the solution becomes unfeasible and the robot base stops moving. Therefore, there's a tradeoff between the smoothness of the trajectory and the feasibility of the optimized trajectory. We determined 1.2×10^{-1} to be an adequate value in this context.

Problem 2

In this section, we present our solution to the cargo delivery task. Unlike the previous question, the robot must pass through certain waypoints while transferring a cargo to the target area. In this sense, this task is a joint optimization problem and can be solved using *ry.Skeleton()*. We define the skeleton instance and obtain the endpoints of the solution waypoints as follows:

```python
config.addFrame(name="target", parent="target_final")\
    .setShape(ry.ST.ssBox, size=[0.9, 1.9, 0.01, 0.01])\
    .setRelativePosition([-0.45, 0, 0])\
    .setColor([0, 0, 1])\
    .setQuaternion([1, 0, 0, 0])

skeleton = ry.Skeleton()
```

Problem 2 continued on next page. . .

```
skeleton.addEntry(timeInterval=[1,1], symbol=ry.SY.touch,
                frames=["l_gripper","cargo_handle"])
skeleton.addEntry(timeInterval=[1,4], symbol=ry.SY.stable,
                frames=["l_gripper","cargo_handle"])
skeleton.addEntry(timeInterval=[0.8,1.2], symbol=ry.SY.downUp,
                frames=["l_gripper"])
skeleton.addEntry(timeInterval=[2, 2], symbol=ry.SY.positionEq,
                frames=["cargo","target_3"])
skeleton.addEntry(timeInterval=[3, 3], symbol=ry.SY.positionEq,
                frames=["cargo","target_5"])
skeleton.addEntry(timeInterval=[4,4], symbol=ry.SY.positionEq,
                frames=["cargo","target_final"])
skeleton.addEntry(timeInterval=[3.8,4.2], symbol=ry.SY.downUp,
                frames=["l_gripper"])
skeleton.addEntry(timeInterval=[4,5], symbol=ry.SY.stableOn,
                frames=["target", "cargo"])
skeleton.addEntry(timeInterval=[5,5], symbol=ry.SY.stablePose,
                frames=["base"])
skeleton.enableAccumulatedCollisions(True)

komo = skeleton.getKomo_waypoints(config, lenScale=1e-1,
                homingScale=1e-2, collScale=1e-1)
ret = ry.NLP_Solver(komo.nlp()).setOptions(stopTolerance=1e-2,
                verbose=1, stopEvals=10000, maxStep=100).solve()
waypoints = komo.getPath_qAll()
```

Our skeleton defines a series of tasks as follows: The gripper touches the cargo handle and preserves this pose until the cargo is dropped off. The robot then travels to waypoints 1 and 2 respectively, dropping the cargo on top of the target afterwards. The target is defined as an *ssBox* instance that resides in the goal area, which enables the usage of *ry.SY.stableOn* symbol when determining the final pose of the cargo. Then, the robot is instructed to preserve its pose until the end, fixating it at its final location and position.

Having defined these waypoints, we then run the pathfinding algorithm to solve the waypoint-pairwise pathfinding problem. The corresponding snippet is as follows:

```
rrt_dofs, rrt_paths = [], []
for i in range(len(waypoints)):
    [config_tmp, q_0, q_1] = skeleton.getTwoWaypointProblem(i, komo)
    while True:
        rrt = ry.PathFinder()
        rrt.setProblem(config_tmp, [q_0], [q_1])
        ret = rrt.solve()

        if ret.feasible == True:
            rrt_paths.append(ret.x)
            rrt_dofs.append(config_tmp.getDofIDs())
            break
        del rrt, ret
```

Problem 2 continued on next page. . .

```
    del config_tmp, q_0, q_1
del komo, rrt, ret
```

Similar to Question 1, we repeat pathfinding until a feasible path is found, and save the necessary data for global path optimization, as performed in the next code snippet:

```
1  komo = skeleton.getKomo_path(config, stepsPerPhase=60, accScale=1e0, lenScale=1e0,
2                               homingScale=1e-2, collScale=1e0)
3  komo.initWithWaypoints(waypoints)
4  for t in range(len(waypoints)):
5      komo.initPhaseWithDofsPath(t_phase=t, dofIDs=rrt_dofs[t], path=rrt_paths[t],
6                                 autoResamplePath=True)
7
8  komo.addControlObjective([], order=1, scale=1.2e-1)
9  ret = ry.NLP_Solver(komo.nlp()).setOptions(stopTolerance=1e-2,
10                              verbose=1, stopEvals=10000, maxStep=100).solve()
11 komo.view(True, "waypoints solution")
12 komo.view_play(True, 0.1)
```

This snippet initializes a KOMO instance from waypoint-pairwise paths found previously on Lines (1-4), enhances the KOMO instance with a control objective that penalizes the squared distance between consecutive positions of the robot (Line 8) to smooth out the trajectory, and then finally solves the optimization problem using *NLP* solver (Lines 9-10). In the end, we obtain an optimized path that looks like this.

Problem 3

In this part, we create an algorithm to push a physics enabled cargo (we also sometimes call it "ball" in text). Our algorithm can push the cargo through all the checkpoints and make it reach the goal_area, in 18.685 seconds in the best case (See Figure 3).

Since the assignment states both the time and algorithm is important in the final decision, we wanted to adhere to some "standards" while devising the algorithm:

1. We did not add any new frames, change physics etc. to either manipulate the world or make ball tracking easier.
2. We run the algorithm multiple times to make sure a solution is reached each time (to the best of our knowledge).
3. We set our hyperparameters after little experimentation, rather than running extensive search.

We believe that robotics is a domain where real-world practicality is extremely important. To this aim, we wanted our algorithm to be as general purpose as possible. Constraints 1 and 3 allow us to create an algorithm where, assuming the cargo and robot objects are the same, a new maze can be solved with little to no changes in the algorithm or the parameters. With constraint 2, we make sure that the algorithm deterministically (again, to the best of our observations) completes the task. This constraint is important since, during our experiments, we encountered cases where the robot could complete the task up to 5 seconds faster, but it also created some cases where the robot could find itself in a state where a solution can never be found.

Problem 3 continued on next page. . .

We now present our progression via ablation studies. In each study, we run the given algorithm 50 times. We then present the mean, standard deviation and minimum time elapsed until successful completion. The results can be found in Table 1. Before we get into the specifics of the algorithm(s), we first present our experimental setup. Please see the last section for execution and result examples.

Table 1: Experiment results. Each model was run 50 times.

Model	Mean	Standard Deviation	Minimum	Fails
M1	36.93	8.09	34.59	3
M2	28.79	6.92	22.94	1
M3 (Submitted)	22.90	2.88	**18.68**	0

Experimental Setup

We use the following loop to conduct our experiments. One can set $EXPERIMENT = False$ to run the simulation only once. The entire algorithm is encapsulated into the *run* function. Next, the code gets current time of the bot object and takes the screenshot of the current bot state (which will be the first timestep where the cargo intersects with the goal_area). We use PIL image library to mirror the image because rai, for some reason, gives us upside-down images (lines until 16). Finally, we reset the state of our environment by resetting our variables and the object positions (lines 18-24). The final part of the code writes the solution times to a file for further processing and persistence.

```
1  all_results = []
2  for i in range(50):
3      solved = run(config, current_cp, cp_index)
4
5      if not EXPERIMENT:
6          break
7
8      solve_time = bot.get_t()
9      if not solved:
10         solve_time = -1
11
12     all_results.append(solve_time)
13     print("Solved in: ", solve_time)
14     unmirrored = np.flipud(config.view_getScreenshot())
15     unmirrored_image = Image.fromarray(unmirrored)
16     unmirrored_image.save(f'{PNG_SAVE_PATH}/solved{i}.png')
17
18     kick_index = 0
19     cp_index = 0
20     current_cp = all_cps[cp_index]
21
22     config.getFrame("base").setPosition([[0, 0, 0.08]])
23     config.getFrame("cargo").setPosition([[0, -1, 0.4]])
24     bot = ry.BotOp(config, False)
25
26 if EXPERIMENT:
```

Problem 3 continued on next page. . .

```
    with open('results.txt', 'w') as f:
        for solution in all_results:
            f.write(f"{solution}\n")
```

We use the following function to check if the cargo has intersected with the goal_area, where we basically check if the position of the cargo is contained within the goal_area:

```
def goal_reached():
    ga_frame = config.getFrame("goal_area")
    ga_pos = ga_frame.getPosition()
    ga_size = ga_frame.getSize()
    cargo_pos = cargo_frame.getPosition()
    return (ga_pos[0] - ga_size[0]/2 <= cargo_pos[0] < ga_pos[0] + ga_size[0]/2) and
    (ga_pos[1] - ga_size[1]/2 <= cargo_pos[1] < ga_pos[1] + ga_size[1]/2)
```

One threat to our experimental validity is that sometimes, the time we see in taken screenshot and what we get from *bot.get_t*() is not the same. Actually, the time we see in the screenshots are almost always lower, which means we are over-reporting our mean and std values. The minimum time (and our final result) is checked against the screenshot since we assume that is what most of the participants will do.

Our code also has a *DEBUG* variable, which when set to *True*, shows the output of the model using a new Frame shaped as a little box. That box is a dummy frame denoting the result of our algorithm at that point in time (i.e. where the robot should be). All we do is set its position and we stress that this frame does not interact with the environment in any way. We use this dummy frame to better show what the algorithm is doing at a given time. The *DEBUG* value is always *False* when $EXPERIMENT = True$.

We now explain the base algorithm, and progressively build upon it.

Base Model (M1)

Our base algorithm can be summarized as ”If the ball needs to go somewhere, find the position for the robot that would push the ball there”. While this might sound obvious in a high level (just push the ball to the goal.. duh), doing this in small increments is a good way of finding a solution. To this aim, we use the five checkpoints that the robot should go through as intermediate goals. When the cargo intersects with any of the checkpoints, we start moving towards the next one. We detail the main steps of the algorithm, while referring to the following *run* function, as follows:

1. Setup the initial state. First target is ”cp1”.
2. Until goal is reached, loop:
   (a) Given current config and target, get the optimal path for the robot. Solution fails if no such path can be determined (lines 3-5).
   (b) Make the robot follow the optimal path (lines 6-10).
   (c) Check if the new position of the cargo, which changed due to robot interaction, is intersecting with the target. If so, update the target to the next one (lines 12-19).

```
1 def run(config, current_cp, cp_index):
2     while True:
```

Problem 3 continued on next page. . .

```
        ret = get_robot_step(config, current_cp)
        if ret is None:
            return False
        for t in range(ret.x.shape[0]-1):
            bot.sync(config, SYNC_TIME)
            bot.moveTo(ret.x[t], timeCost=5, overwrite=True)

            config.setJointState(ret.x[t])

        diff = abs(cargo_frame.getPosition()[:2] - config.getFrame(current_cp).getPosition()[:2])
        if check_cp_change(diff, cp_index):
            cp_index += 1
            if cp_index < len(all_cps):
                current_cp = all_cps[cp_index]
                print(current_cp)
            else:
                return True

        if goal_reached():
            return True
```

Finding the optimal path of the robot, and frankly what that path should be, is the crux of the algorithm (Given as *calculate_robot_position* below). To calculate the optimal position for the robot, we use three variables: position of the target, cargo and the robot. The vector $\vec{CT}$ (cargo to target) gives us the optimal direction for the cargo. Since it cannot move by itself, the robot should position itself in the same direction as this vector, and move/push in the direction of $\vec{CT}$. Lines 5-14 show the calculation of this robot position. First, the unit vector $\vec{CT}$ is found. Then, we multiply this unit vector with the kick_distance. kick_distance is the distance of contact of the robot to the cargo. Calculation for the kick distance vector is given in the expertly drawn Figure 1. This distance vector is negated since, when we multiply it with $\vec{CT}$, we want the result to be in the other side of the cargo (i.e. the output will not be between the cargo and its target). This resulting vector, combined with the cargo position, gives the point at which the robot should arrive at to push the cargo in the direction of $\vec{CT}$.

```
1  def calculate_robot_position(config, target_cp):
2      kick_distance = [-.487, -.487, .32]
3      cargo_position = cargo_frame.getPosition()
4
5      def calculate_pos(tcp_position, x_position, kick_distance):
6          diff_vector = tcp_position - x_position
7          magnitude = sum(diff_vector**2)**0.5
8          x_to_tcp_vector = [value / magnitude for value in diff_vector]
9
10         robot_position = cargo_position + [kd * cttcp for kd, cttcp in zip(kick_distance, \
11         x_to_tcp_vector)]
12         robot_position[-1] = 0.08
13
14         return robot_position
15
```

Problem 3 continued on next page. . .

```
    if target_cp == "cargo":
        robot_position = config.getFrame("base").getPosition()
        return calculate_pos(cargo_position, robot_position, kick_distance)
    else:
        tcp_position = config.getFrame(target_cp).getPosition()
        return calculate_pos(tcp_position, cargo_position, kick_distance)
```

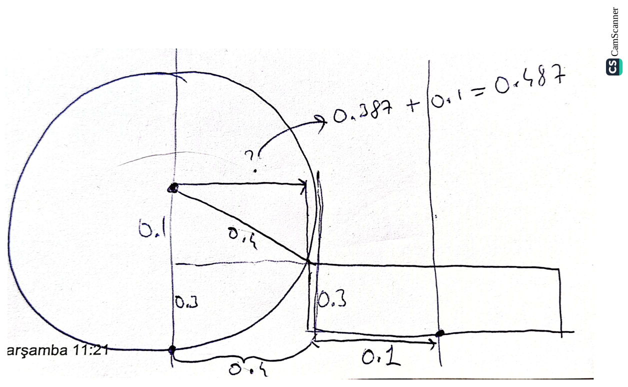

Figure 1: calculation of kick_distance

One case where this approach fails is that, what will happen if this given position is infeasible? i.e. what happens if the calculate robot_position collides with a wall? To handle this case, if the calculated position is infeasible, we temporarily set the robot's vector alignment target to be the cargo itself, rather than cargo's target trajectory (see code for *get_robot_step*). This way, the robot always has a sub-optimal but feasible position at all times.

```
def get_robot_step(config, target_cp):
    robot_position = calculate_robot_position(config, target_cp)
    q_0, q_target = IK(config, robot_position, target_cp)
    ret = find_path(config, q_0, q_target)
    if ret is None:
        robot_position = calculate_robot_position(config, "cargo")
        q_0, q_target = IK(config, robot_position, "cargo")
        ret = find_path(config, q_0, q_target)
    return ret
```

The last parts to explain in the algorithm are the IK and $find\_path$ functions. These functions are very close to the solution in Problem 1, where we use KOMO and PathFinder to find a non-colliding path for the robot (We do not do the smoothing step since the calculated waypoints are usually very close together and smoothing them are not very useful). For optimization constraint, instead of using two frames with the ry.FS.position feature, we use a single frame and ask KOMO to optimize the distance of the base frame with the robot_position (which we calculated prior):

```
def IK(config, robot_position):
    q_0 = config.getJointState()
```

Problem 3 continued on next page. . .

```
    komo = ry.KOMO(config, phases=1, slicesPerPhase=1, kOrder=2, enableCollisions=True)
    komo.addObjective(times=[], feature=ry.FS.accumulatedCollisions, frames=[], \
    type=ry.OT.eq, target=0),
    komo.addControlObjective(times=[], order=1, scale=1.2e-1)

    komo.addObjective(times=[], feature=ry.FS.position, frames=['base'], \
    type=ry.OT.sos, scale=[1], target=robot_position)

    ret = ry.NLP_Solver(komo.nlp()).setOptions(stopTolerance=1e-2, verbose=0).solve()
    q_target = komo.getPath()[0]

    return q_0, q_target

def find_path(config, q_0, q_target):
infeasible_count = 0
while True:
    rrt = ry.PathFinder()
    rrt.setProblem(config, [q_0], [q_target])
    ret = rrt.solve()
    if ret.feasible:
        return ret
    infeasible_count +=1
    if infeasible_count == 5:
        return None
    del rrt
```

This concludes the explanation of the base algorithm. We now do optimizations to this model.

Base Model with Oscillating Kick Distance (M2)

M1 model utilized a kick distance which puts the robot at a point where the cargo is interacted just barely. This approach has two shortcomings: 1) If the ball gets stuck in a corner, it will be stuck there forever and 2) Since the IK calculation tries to maintain this distance, the robot will go between barely touching and not touching at all, reducing the force applied to the cargo by the robot. To alleviate these issues, we calculate an oscillating kick distance schedule, where the kick distance changes to a higher or lower value each time step, oscillating between maximum 0.487 and 0.25. Here, the minimum value of oscillation is the point at which the ball tips over to the top of the robot. We calculate this tipping point to be around 0.21 using a similar method to what we presented in Figure 1. Problem is, since the tipping point also depends on the mass of the ball and the force applied at the tipping edge (which depends on the point acceleration and the mass of the robot), the actual tipping point is significantly harder to predict. We experimentally see that 0.25 is good enough (i.e. the ball doesn't tip during regular push motion) and don't optimize it further.

```
def get_oscillating_kicks():
    osc_range = np.concatenate((
        np.arange(init_kick_distance[0], max_kick_distance[0] + SYNC_TIME, SYNC_TIME),
        np.arange(max_kick_distance[0], init_kick_distance[0] - SYNC_TIME, -SYNC_TIME)
    ))

    oscillating_kick_distances = [(x, y, init_kick_distance[2]) for x, y in zip(osc_range, osc_range)
```

Problem 3 continued on next page. . .

```
    return oscillating_kick_distances

// ... When we calculate robot position:
kick_index += 1
kick_distance = oscillating_kick_distances[kick_index % len(oscillating_kick_distances)]
```

We observe that when the ball is stuck in a corner or near the wall, closer kick distances help nudge it out of there, and a solution can be found. Also, closer kick distances increase the push speed of the ball, giving us a significant solution time reduction.

Base Model with Better Hyperparamters (M3)

We now improve upon the M2 model by changing some of the parameter values:

1. We used to attempt at most 5 times to find a feasible $find\_path$ solution, since that is the optimal path. Now, we just use the unoptimized path if optimal path solution fails for that timestep.
2. We increase the *timeCost* for *bot.moveTo* from default 5 to 15, to punish late movements more (i.e. when the checkpoint changes, the robot used to take more time to adjust to the new trajectory)
3. A checkpoint change would happen when the ball was 0.4 close (radius of sphere) to the checkpoint (i.e. intersected with a checkpoint). This would change the trajectory too suddenly, and the robot had a hard time to adjust. We now use 2 times the radius as the checkpoint change, which creates a smoother path. Of course, intersection logic with the goal area stays the same.

We have also seen a suggestion by the TA in Moodle, regarding the use of different friction coefficient values. We did not include this since 1) Although we could get faster solutions (around 2-3 second), they were not consistent and some solutions would get stuck and 2) We did not want a physical property of the material to be a hyperparameter of the algorithm. We believe that with enough patience, one can hard-code a hit-trajectory that will bounce the ball straight to goal area with minimal pushing.

Attempts at Improvement

We attempted to find an optimized path that the cargo should always follow. Although a bit hacky, we can actually find the optimal path for the cargo. To do this, we first find the optimal path from the robot to the goal, just as we did in Problem 1 of this assignment. Then, if we pick the closest point from this path to the cargo to be our initial point, we have an almost optimal path from the cargo to the goal area! Of course, since the cargo cannot move on its own, we need to find the position for the robot that will move the cargo to the given waypoint (See our main algorithm). This sounds good in theory, but sadly, if the waypoints are too many, the target trajectory vector changes too much and the robot has to constantly make corrections. These corrections in turn cause unintended hits to the cargo, which pushes it further from the optimal spline. If we sample, say, one tenth of the way points, the performance is actually good and the algorithm seems more robust. Problem is, the time we spend calculating the optimal path overcomes the average time we save (which is around 1-2 seconds). We did not build upon this idea further, but with a better implementation this could give a better solution, and help us get rid of the 3rd improvement of M3 (which feels a tiny bit hard-coded). Figure 2 shows a screenshot of this attempted solution. We used dummy frames to show the path (again, they do not affect the execution in any way). Dummy frame in red is the next target the robot is aiming at.

We believe an optimal solution can be found if one is willing to model the contact of the "roomba" with the sphere as a constant force exerting on the sphere, add in the inertia of the sphere, the friction of the sphere

Problem 3 continued on next page. . .

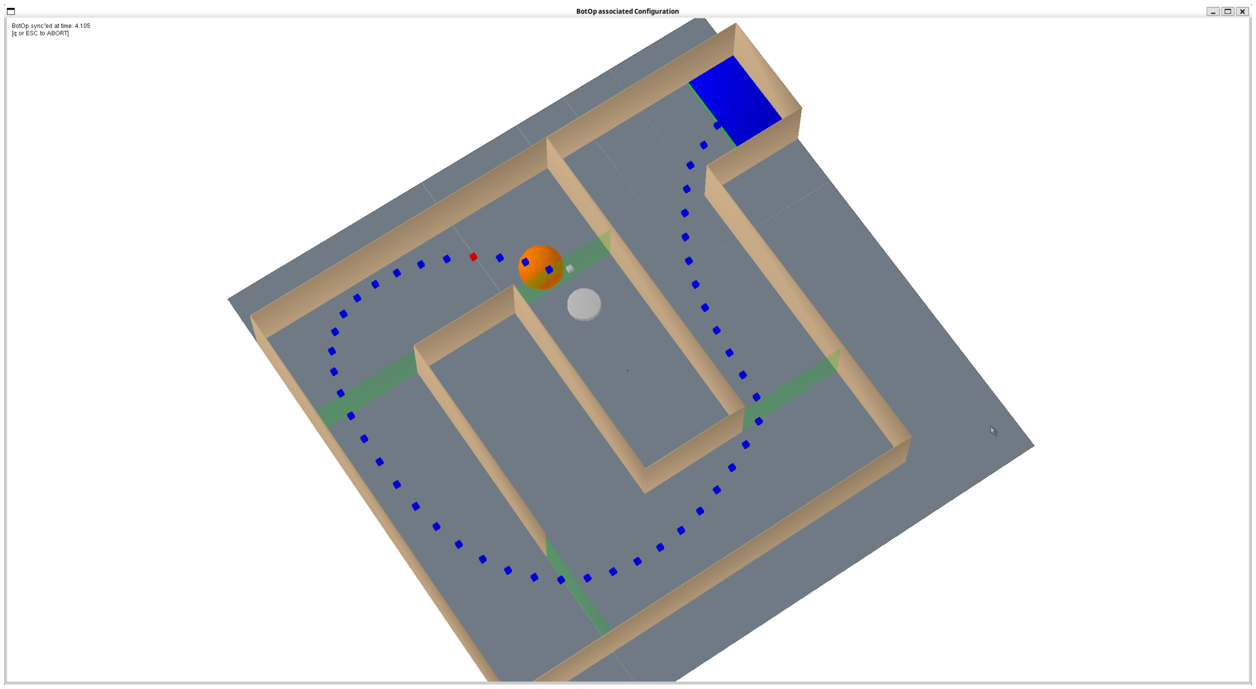

Figure 2: Attempted solution using a precalculated optimized path. Red box shows the next trajectory, white box shows the location the robot should be in at the given time step

while rotating, solve this entire function at each time point to estimate the velocity and the acceleration, to finally predict the next position of the ball, so that we can devise a KOMO path from the robot to this predicted path. We did not attempt to do this because... yeah.

The very best solution, which we did not attempt, happens in 0 seconds of push time with ARAS Kargo algorithm, where the cargo is simply not delivered and the recipient is asked to come to the delivery center to pick up the cargo.

Execution Examples

Figure 3 shows the screenshot of our best run. We are aware that the camera angle is a bit awkward, but we believe even in this angle, the cargo can be clearly seen intersecting the finish line based on the nearby walls and tile. You can find an example run of our our model in the following link. This run takes around 21 seconds and it is not our best run, it is just a sample run. Creating a video is more cumbersome and we hope that the screenshot is proof enough.

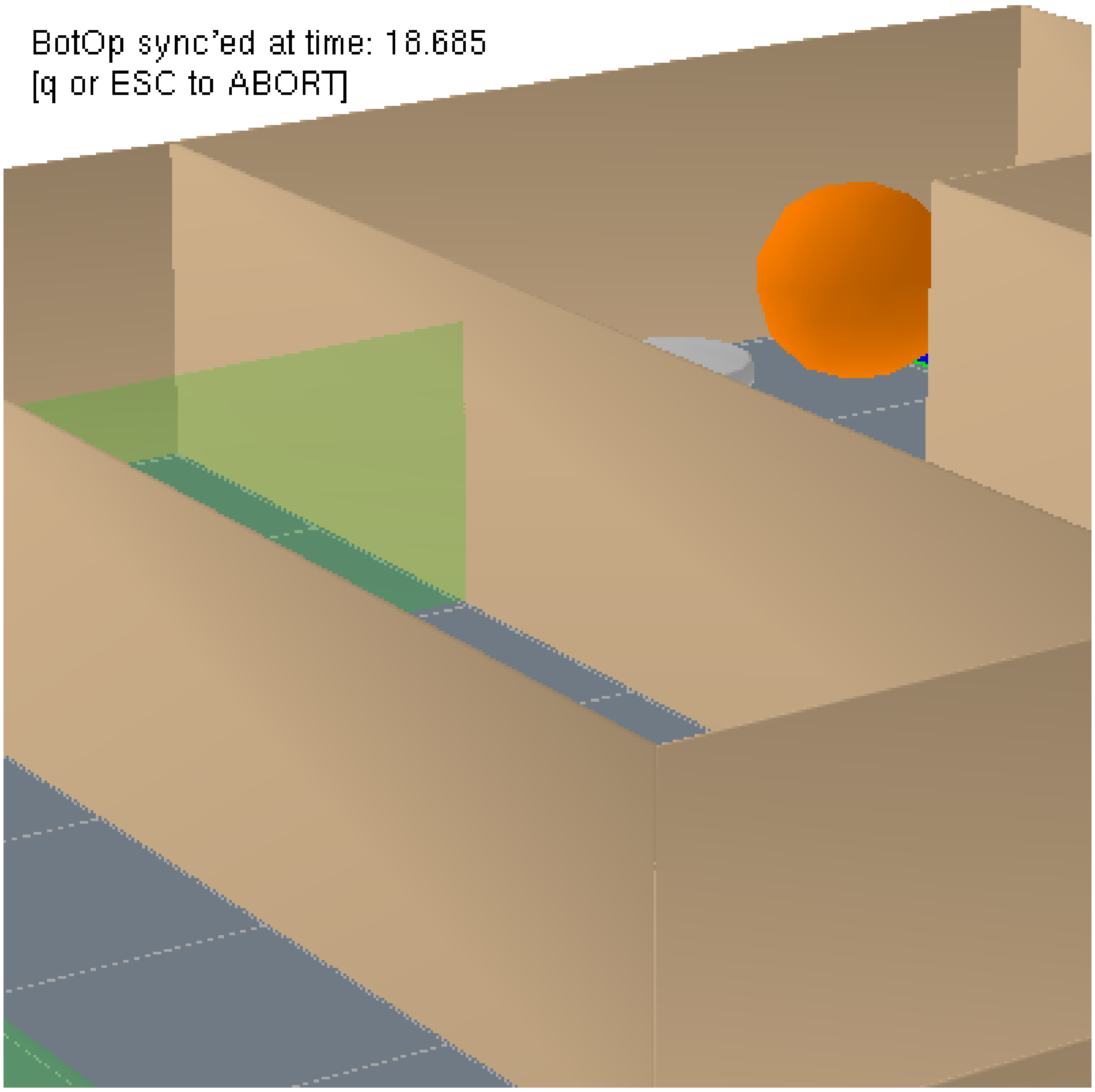

Figure 3: Our best solution