

# Cart pulling research notes

Marko Guberina

March 13, 2025

## Contents

<b>1</b>	<b>Problem description</b>	<b>1</b>
<b>2</b>	<b>Solution approach</b>	<b>2</b>
2.1	Formal task description . . . . .	2
2.2	System architecture . . . . .	2
<b>3</b>	<b>Control design</b>	<b>2</b>
3.1	Optimal control formulation . . . . .	2
3.1.1	Turning OCPs into MPCs . . . . .	3
3.1.2	Common OCP parts . . . . .	3
3.1.3	Decision variables, dynamics constraint . . . . .	4
3.1.4	State constraints . . . . .	4
3.1.5	Regulation costs (regularization) . . . . .	4
3.2	Task-defining objective functions . . . . .	4
3.2.1	Point-to-point tasks . . . . .	5
3.2.2	Path-following tasks . . . . .	5
3.2.3	Solver choice . . . . .	6
3.3	Reference trajectory construction . . . . .	6
3.3.1	Base reference trajectory . . . . .	6
3.3.2	End-effector reference trajectory . . . . .	7

## 1 Problem description

We have a mobile robot with hands at our disposal. We want the robot to take a cart of things, and pull it from point A to B. This is taking place in a crowded environment, so the robot needs to be able to detect dynamic obstacles (people) and navigate around them. The robot has lidar, a camera, positional encoders on everything, an IMU and TODO: [other things we use].

The key aspects of this problem are:

1. mapping and localization, as for anything mobile
2. detecting obstacles

3. very quick planning - we can't know where the people will go, so we need to plan quickly (can't rely on old plan which assumes obstacle follows a particular trajectory)
4. controller capable of simultaneously following the prescribed path and handling of the cart

We tackle each of these problem as follows:

1. Use off-the-shelf ROS2 Nav2 stack which has an implementation of SLAM. We know the environment we're in, and rely on a map made in advance.
2. Some part of the ROS2 stack on the robot TODO: [know which one] takes the lidar point point cloud, filters out the unmapped detected things as a separate layer on the map. Those can be assumed to be dynamic obstacles.
3. Use Albin Dahlin's [TODO: citation] dynamic system planning in star-world. Furthermore, use his code implementation. During operation, the planner's own map needs to be updated, as it relies on star-shaped obstacles, and not on a pixel-grid. This should use as much caching as possible, but at least the static part of the map.
4. The ideal solution seems to be creating a whole-body MPC. Optimal control strategies can accommodate multiple optimization objectives, and underactuation in the base of the robot. An alternative would be to disjointly control the base and the arms. As will be explained later, we construct an exact path for the end-effector to follow, so as long as it follows it, it doesn't matter whether it's full or disjoint body control (as long as we don't have additional criteria to on top of the base and the end-effector positions).

## 2 Solution approach

### 2.1 Formal task description

### 2.2 System architecture

TODO: finish

The only communication between the planner and the controller is the planner sending the path. However, in the current implementation, it was easier for the controller to communicate the current position to the planner.

## 3 Control design

### 3.1 Optimal control formulation

We propose a direct trajectory optimization approach for our conception of the cart pulling problem. Furthermore, we solve the problem on the dynamics level.

We use the Box-FDDP solver [reference] from the Crocoddyl optimal control suite [reference]. This approach has the following benefits:

- control of underactuated mobile bases like differential drive
- flexibility in objective function design makes the approach extensible, for example allowing to penalization of centrifugal forces when handling tall carts, or extending the approach to legged robots

A generic optimal control problem looks like

$$X^*, U^* = \operatorname{argmin}_{X, U} \sum_{k=1}^{N-1} \int_{t_k}^{t_{k+1}} l(x, u) dt + L(x(N)) \quad (1)$$

$$\text{such that } \dot{x} = f(x, u) \quad (2)$$

$$x \in \mathcal{X}, u \in \mathcal{U} \quad (3)$$

$$(4)$$

A common choice for  $\Delta t = t_{k+1} - t_k$  is  $\Delta t = 0.01$ .

### 3.1.1 Turning OCPs into MPCs

Any OCP described here is turned into an MPC by:

- **Shortening the time horizon  $N$ , capping the maximum number of solver iterations  $n_{\text{solver\_iter}}$ , and reducing control frequency  $f_{\text{ctrl}}$**  until satisfactory performance is reached. From limited qualitative testing, the most common numbers are  $N = 30$ ,  $n_{\text{solver\_iter}} = 10$ , and  $f_{\text{ctrl}} = 200$ .
- **Warmstarting** with the previous solution. The previous solution is offset by the amount of time passed. TODO: make sure it is, make sure the ctrl\_freq amount of time, not  $\Delta t$  time. If the second is the case, you probably want to linearly interpolate to offset correctly - also make sure you're time-testing whether this is actually helpful (if it doesn't shave off an iteration off of the solver, it isn't)

### 3.1.2 Common OCP parts

Depending on the robot used, and the task at hand, there are different OCPs being used. For example, to initialize the robot's position, we use an objective function for point-to-point motions, while we use a different one for path tracking.

Due to the complexity of the final controller, and of the total control system design (including planning), during the development process it was necessary to build smaller controllers to increase understanding of the controller, informing and guiding the final design.

It's easiest to begin by describing the common parts of the all OCPs.

### 3.1.3 Decision variables, dynamics constraint

In our case, the state is a vector of joint positions and velocities:

$$x = \begin{bmatrix} q & v \end{bmatrix}^T. \quad (5)$$

The control inputs can be either acceleration or torques, depending on which we formulate either the inverse or the forward dynamics problem respectively, and solve for the other quantity. We used inverse dynamics. In any case, because the states are also decision variables, a state-input pair must follow the system dynamics into the next that. Thus to put in the discrete version of  $\dot{x} = f(x, u)$ , we can write

$$\text{dynamics}(\delta x_{k+1}, \delta x_k, \delta u_k) = 0 \quad (6)$$

$$\text{dynamics}(\delta \dot{x}, \delta x, \delta u) = \delta \dot{x} - (f_x \delta x + f_u \delta u) \quad (7)$$

TODO: elaborate further . Also explain to self whether efforts are torques. also  
 TODO: probably a good idea to actually formulate with the forward dynamics version too, it's about 10 basic lines of code and an argument.

### 3.1.4 State constraints

Since we use Box-FDDP, we can only have box constraints on torque inputs, i.e.  $\tau_{\min} \leq \tau \leq \tau_{\max}$ . In the case of an underactuated joint, we set the torque limit on it to zero.

To put constraints on the state, namely on arms' joint limits and on all joints' velocities, we put them into a quadratic barrier function which we put into the objective function. In particular (TODO make sure this is correct):

$$r_{lb} = x - x_{\min} \quad (8)$$

$$r_{ub} = x - x_{\max} \quad (9)$$

$$B(x) = \begin{cases} \frac{1}{2}(\|r_{lb}\|^2 + \|r_{ub}\|^2) & \text{if } lb \leq x \leq ub \\ \text{max float} & \text{otherwise} \end{cases} \quad (10)$$

### 3.1.5 Regulation costs (regularization)

Finally, we add regulation costs on both the state  $c_1 x^T x$ , and control inputs  $c_2 u^T u$ .

## 3.2 Task-defining objective functions

Depending on the task, we create a different objective function. Before going into the functions themselves, the reader should be briefed about some specifics of Crocoddyl, the library used for optimal control (more detailed text can be found in the appendix of "Robotics notes").

At every timestep of the OCP, also known as a knot, there is an ActionModel. An ActionModel store all data and functions about the OCP at this point. This

means that an OCP can be constructed as a combination of different dynamics models, constraints, and objective functions, all defined just at one knot. This is used in path-following, where every knot is assigned a each own target (in this case a path point) to reach in the objective function. Another useful feature is setting different cost coefficients, which most useful in setting zero terminal velocities at the final knot. Note that the last point is not a good idea in MPC!

### 3.2.1 Point-to-point tasks

In case there is a fixed end-goal to reach, we use the  $se(3)$  error norm as the task-specific function to put into the objective function.

**End-effector pose task** In the particular case of the goal being defined for the end-effector, whose pose is denoted as  $T_w^e$  reaching a goal denoted as  $T_w^{\text{goal}}$ :

$$l_e(x) = \left\| \log \left( (T_w^e)^{-1} \right) T_w^{\text{goal}} \right\|_2 \quad (11)$$

Here we of course use only joint values  $q$ , not the full state  $x$ , because the end-effector pose is only a function of  $q$ .

We put this cost function at every knot.

**End-effector pose and base position** In case we also want to specify the position of the base, we additionally put in the translational cost for the base. If we denote the pose of the base is

$$T_w^b = \begin{bmatrix} R_w^b & p_b \\ 0 & 1 \end{bmatrix} \quad (12)$$

then the translational cost is given by

$$l_b(x) = \|p_{\text{goal}} - p_b\| \quad (13)$$

Here it should be noted that the  $z$  coordinate is 0 for both the goal and the base pose. The orientation of the base could be accounted for in the same manner.

To get the joined task, we simply sum the two goal costs:

$$l_{\text{joined}}(x) = l_e(x) + l_b(x) \quad (14)$$

### 3.2.2 Path-following tasks

As discussed in 1, the controller is tasked with following a path prescribed by a path planner. One way, and probably the best way, to do this would be to parametrize the path with a parameter  $s \in [0, 1]$ , and task the controller with timing it, i.e. constructing a trajectory on it. In this case the time-scaling factor of the path  $\lambda$  in the equation  $t = \lambda s$  would be a decision variable in the OCP.

However, we found it easier to manually construct a trajectory, i.e. a timing law on the path. This is because it is very easy to construct a trajectory

following OCP. Namely, it is done by modifying the point-to-point formulation by replacing  $T_w^{goal}$  with  $T_w^{traj}[i]$  where  $T_w^{traj}[i]$  is the point on the desired trajectory corresponding to the desired timing law on the path. More on path and trajectory construction can be found in 3.3. The important point here is that the trajectory needs to be interpolated so as to correspond to the timing of the knots. To be more specific, the time difference  $\Delta t$  between two indexes of the array defining the reference trajectory has to be the one in the OCP. For our purposes linear interpolation is sufficient, especially because the time-horizon in the MPC is rather short, leading to a short trajectory in space as well

**End-effector trajectory following cost function** The task-defining running cost for end-effector trajectory is given by (here with explicit indexing, ):

$$l_{path}(x[i]) = \|\log((T_w^e[i])_1) T_w^{ref}[i]\|_2 \quad (15)$$

**Base path following cost function** Works in the same way as the described above for the end-effector case, just with different translations instead.

**Base and end-effector trajectory following cost function** Is again simply the sum of the two costs.

### 3.2.3 Solver choice

Box-FDDP from Crocody seems to be the fastest, and provides control input constraints which is crucial - otherwise one would need to overtune control input regulation, leading to a steep decrease in performance.

CSQP from mim-solvers was also tested, but it was too slow (NOTE: idk if I set it up right, the paper said it should be faster than what I got). The benefit it has is that it can accommodate other linear constraints (it's based on sequential quadratic programming (SQP), made faster for trajectory optimization through it's implementation).

Casadi' IPOPT is a good choice for experimenting with different problem formulations, but being a generic optimization solver, and all the derivatives being obtained through finite differencing, it is of course too slow for practical implementation.

## 3.3 Reference trajectory construction

### 3.3.1 Base reference trajectory

The path planner produces a path  $\gamma_2 \in \mathbb{R}^2$  for the base of the robot. It is given by an array of points  $\Gamma \in \mathbb{R}^{n \times 2}[i]$  where  $i$  is the index denoting  $i$ th element (point) of the array, and indexing starts with 0. We begin by introducing a physically dimensionless parametrization  $s \in [0, 1]$ , resulting in  $\gamma_2(s)$ . Importantly, we assume that  $\Gamma$  is constructed from a uniform sampling of  $\gamma$  (this is true with the chosen path planner).

We want to impose a timing law so that the path is traversed at a fixed velocity,  $v_{\text{ref}} c_3 \max v_{\text{base}}$  where  $c_3 \in [0, 1]$ . A different timing law ensuring the same velocity will be constructed for the end-effector reference.

We construct the timing law by calculating the time required to traverse the prescribed path at this velocity. To do so we first calculate the arclength of the path via  $L(\gamma) = \int_s \|\dot{\gamma}(s)\| ds$ . In discrete space perform the following computation  $L(\Gamma) = \sum_{i=1}^n \|\Gamma[i+1] - \Gamma[i]\|$ . From this we get  $t_{\text{final}} = \frac{L(\gamma)}{v_{\text{ref}}}$ , leading to the timing law  $t = t_{\text{final}} s$ , and the reference trajectory  $\gamma(t(s))$ .

### 3.3.2 End-effector reference trajectory

As discussed in 2, we want the cart to follow the past path of the base. We denote the continuous past path with  $\phi(s) \in \mathbb{R}^2, s \in [0, 1]$ , and  $\Phi[i] \in \mathbb{R}^{n \times 2}$  as the discrete collection of points. In the software implementation,  $\Phi$  is a double ended queue where at every iteration of the control loop a new point  $p_{\text{base}}$  is inserted, and the oldest entry is removed. We refer to this double ended queue as the rolling past window. The size of the rolling past window  $n$  is set by the user, and it should be long enough to construct the preferred trajectory. Note that  $\Phi$  is initialized as a straight line from the end-effector to the base of the robot.

Assuming a stable control frequency (it empirically holds well enough),  $\Phi$  is a uniform sampling of  $\phi$ . We want to maintain a fixed distance between the base and the handlebar of the cart. Since the handlebar is a rigid body, it also uniquely determines the references for the end-effector(s). Thus the first task is to find the point on the past path which is the preferred distance from the robot on the arclength of the past path. As in the case of calculating the length of the path given by the path planner for the base, we use  $L(\Phi) = \sum_{i=1}^n \|\Phi[i+1] - \Phi[i]\|$ . By computing this sum in a loop, we find the index  $h$  for which  $L(\Phi) = \sum_{i=1}^h \|\Phi[i+1] - \Phi[i]\| = d_{\text{preferred}}$ , where  $d_{\text{preferred}}$  denotes the preferred distance of the handlebar to the base of the robot. Since we do not care about the past path beyond the index  $h$ , we cut them off of  $\Phi$ . The resulting path from the preferred handlebar distance to the base of the robot can then be parametrized with  $s \in [0, 1]$ .

The next step is to construct an  $SE(3)$  reference from the 2-dimensional past path in  $(x, y)$  coordinates of the world frame. In the dual arm case this will be the absolute frame of the dual arm manipulation task. Since the robot is operating on flat ground, and the cart is rigid, the height, i.e. the  $z$  coordinate is known in advance and fixed. Since the handlebar is rigidly grasped, and we want the cart to follow the base's path, the rotational axes are fixed as well. We set them by constructing a Frene frame on the past path. Without loss of generality, let the  $z$  axis of the end-effector frame be perpendicular to the ground, and pointing down. Also let the  $x$  axis of the end-effector frame be parallel to the tangent of the path (either direction is fine, we choose toward the mobile base). This ensures the same orientation of the cart as the base of the robot on the path. Finally, the  $y$  axis simply completes the right-handed

frame.

To make this construction, we need to compute the tangent of the past path  $\dot{\phi}(s)$ . Alternatively base velocities could also be logged, but this approach is less prone to noise. TODO: this is not a 100% obvious statement due to potential localization-induced teleporting. For simplicity, we compute the angle of the secant of the path in the world frame, and formulate the rotation matrix from it. In particular

$$\theta[i] = \text{atan2} \left( \frac{\phi_y[i+1] - \phi_y[i]}{\phi_x[i+1] - \phi_x[i]} \right) \quad (16)$$

which leads to

$$R_w^{\text{ref}}[i] = \begin{bmatrix} \cos(\theta[i]) & \sin(\theta[i]) & 0 \\ \sin(\theta[i]) & -\cos(\theta[i]) & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad (17)$$