Minimum Distance Movement Optimal Control Problem for Robotic Manipulator

Daren Swasey
Utah State University College of Engineering
daren.swasey@usu.edu

***Abstract*— Robotic manipulators are simple to operate from a manual control standpoint, and for typical operations on a manipulator that is not utilized often, there is no need for efficient movement. This project explores the concept, design, and development of an optimal control problem to perform straight-line movement of an end-effector for the RX200 manipulator from Trossen Robotics. Kinematics, dynamics, and the optimal control problem are reviewed and discussed. The issues in implementation and the results of the project are highlighted with potential for further work.**

I. Introduction

The outline of the submission is as follows. This section gives a brief overview of the problem to be solved. This section also addresses the basics of the system used as a reference, the RX200 manipulator from Trossen Robotics. The dynamics of the arm will be covered in part I-B, and the control input will be discussed in part I-C. Lastly, this section presents some results regarding system discretization in part I-D.

Section II defines the solution approach. Two approaches were taken, and both failed to produce the expected results due to implementation difficulties. The first approach taken was to use the OCS2 library (Optimal Control toolbox for Switched Systems), by means of an iterative LQR problem. The second approach taken was a simultaneous direct method in MATLAB, utilizing code used previously in the semester as a starting point. Continuing, the first subsection of section II provides a brief overview about both attempted solutions, but will focus mainly on the development from the OCS2 library. A mathematical description of the solution approach is then shown, again with more emphasis on the OCS2 library approach, which utilized a form of LQR.

Section III addresses the results of the solution attempts. There some time is taken to reflect on the original approach, how developments were made over time, and what the final results were. Although the solution approach started with the OCS2 library, roadblocks there indicated it was better to begin to utilize MATLAB. Both approaches had their benefits and detriments, but ultimately neither could be developed to a state where the problem was actually solved.

In section IV, a conclusion about the project is available, recapping the key learning moments and solution attempts of this project. After the conclusion, references are provided regarding the work that was completed thus far.

A. Problem Description

The problem this project attempts to solve is a minimum distance optimization problem for a robotic arm end-effector. The desired end goal is to be able to tell a robot arm a point within it's workspace, and from the current position of the end-effector, move along a minimum distance path (minimum distance for the end-effector) and reach and stay at the desired point.

The solution development will focus on achieving the capability to simulate the problem, as other considerations such as object collision (including self-intersection) are not within the scope of this project. The results are expected to be that given a feasible point in the robot manipulator workspace, the robot arm will move from initial position to final position with what would generally be as close to a linear motion as possible. That is, if the end-effector were drawing its trajectory in the workspace, it would be linear or nearly linear. A few variations would be expected to appear if the line between the starting and ending poses intersected any positions outside the safe workspace of the manipulator, or if the path would cause self-intersection.

B. Dynamic System Definition

The optimization problem for this system can be developed in quite a simple form. Several things need to be known in order to fully define the optimization problem. This includes joint-angle bounds, forward kinematics, and the linear dynamics.

To start, let's develop the kinematics. The transform matrix from body frame to end-effector frame can be utilized to define components of both the cost and the partials in the optimization problem setup. In order to develop the transformation matrix, first we need to define the coordinate transforms for the system. Figure 2 shows the definition of these frames.

The frame at the bottom left is the shoulder frame. The shoulder frame is obtained by a transform from the body frame to the waist frame and the waist frame to the shoulder frame. The body frame's i axis points to the front of the RX200 assembly. The waist frame is obtained by a rotation about the k axis of the body frame, with its i axis pointing in the same direction as the i_s axis shown in figure 2. The frame directly above the shoulder frame is the elbow frame. The final frame to the right of the elbow frame is the wrist frame. The transformation matrix from shoulder to

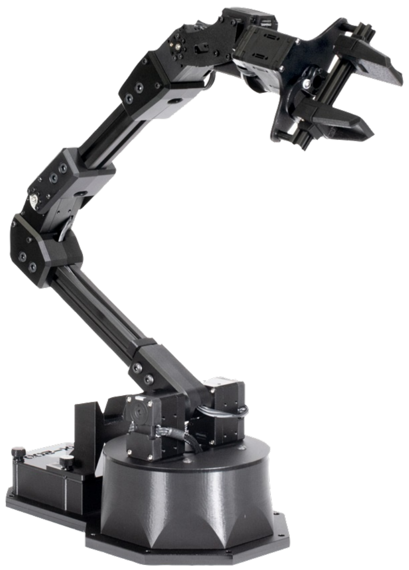
Fig. 1. Reactor X200 Robotic Arm [2]

end-effector position can be formed using a series of general transformations matrices. The general 3D affine transformation matrix is defined in equation (1) from [1].

$$T = \begin{bmatrix} c_\psi c_\theta & -s_\psi c_\phi + c_\psi s_\theta s_\phi & s_\psi s_\phi + c_\psi s_\theta c_\phi & t_x \\ s_\psi c_\theta & c_\psi c_\phi + s_\psi s_\theta s_\phi & -c_\psi s_\phi + s_\psi s_\theta c_\phi & t_y \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Name	Description
c	Shorthand for $\cos(\cdot)$
s	Shorthand for $\sin(\cdot)$
ϕ	Rotation angle about the i axis
θ	Rotation about the j axis
ψ	Rotation about the k axis
t_x	Translation distance in i axis to next frame
t_y	Translation distance in j axis to next frame
t_z	Translation distance in k axis to next frame

Using the general formula in (1), the following parameters are also defined for the transformations between each of the shoulder, elbow, and wrist frames. Subscript s refers to the shoulder, e to the elbow, w to the wrist, b to the body (base of the robot), wt to the waist (with $i_{wt} = i_s$), and ee to the end-effector point.

By using the above variables in their respective transformation matrices, a point $q_{target} \in \mathbb{R}^4$ can be calculated via the following equation, where each variable in table I is plugged in to its respective matrix.

Transformation Variables

Variable	$\mathbf{T}_b^{wt}$	$\mathbf{T}_{wt}^{s}$	$\mathbf{T}_s^{e}$	$\mathbf{T}_e^{w}$	$\mathbf{T}_w^{ee}$
ϕ	0	0	0	0	0
θ	0	θ_s	θ_e	θ_w	0
ψ	ψ_{wt}	0	0	0	0
t_x	0	0	d_1	d_2	d_3
t_y	0	0	0	0	0
t_z	0	0	d_0	0	0

TABLE I
TRANSFORM TABLE

$$q_{target} = \mathbf{T}_b^{wt}\mathbf{T}_{wt}^{s}\mathbf{T}_s^{e}\mathbf{T}_e^{w}\mathbf{T}_w^{ee}\begin{bmatrix}0\\0\\0\\1\end{bmatrix} \tag{2}$$

The dynamic system without any feedback control appears as in equation (3). While there are two more actuators available on the robot, the wrist rotation and gripper actuation, these are not important to the optimization and are therefore left out of the optimization problem development.

$$x = \begin{bmatrix}\psi_{wt}\\ \theta_s\\ \theta_e\\ \theta_w\end{bmatrix}, \dot{x} = \begin{bmatrix}\dot\psi_{wt}\\ \dot\theta_s\\ \dot\theta_e\\ \dot\theta_w\end{bmatrix} = \mathbf{A}x + \mathbf{B}u = u \tag{3}$$

$$\mathbf{A} = 0,\ \mathbf{B} = \mathbf{I},\ \mathbf{I} \in \mathbb{R}^{4x4} \tag{4}$$

C. Representative Control Input

A simple state-feedback control configuration as shown in equation (5) can be used to control the system. Here, we redefine the state of the system in terms of z, where z is the error between the actual state and the desired state. This allows us to develop a set of matrices (A, B, K) that will drive the error to zero.

Note that this development makes the assumption that the RX200 servos give an accurate measure of position, thus making a simple state-feedback controller possible. The optimal control problem can also be considered to implement a form of this control. The OCS2 implementation of the control problem would seek to provide something of a global optimal feedback policy to reach the goal, where the MATLAB implementation of the simultaneous method would use a discretized form of this, with x_d being defined at each discrete time-step k.

$$u = -Kz,\ z = x - x_d \tag{5}$$

The operational limits of the state are as follows in equation (6), where the limits are defined in degrees. A further constraint to maintain the safety of the robot is the z-axis position of the end-effector, shown in equation (7).

$$\begin{gathered} -180 < \psi_{wt} < 180 \\ -108 < \theta_s < 113 \\ -108 < \theta_e < 93 \\ -100 < \theta_w < 123 \end{gathered} \tag{6}$$

$$q_{target_z} >= 0 \tag{7}$$

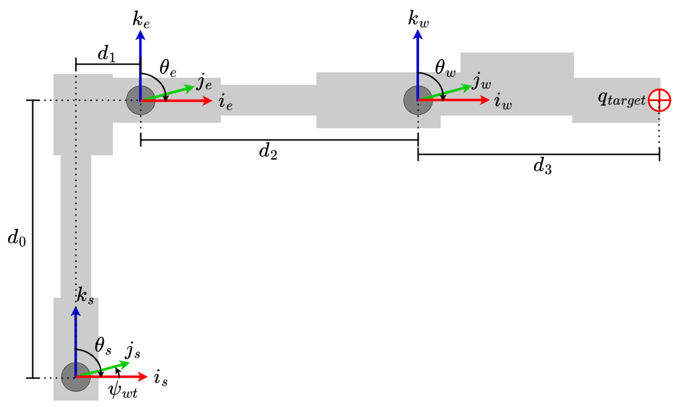

Fig. 2. Home position coordinate frame definitions with transformation matrix variables

The limits of the control input u from equation (5) are defined in parameters given by Trossen Robotics given in [2]. The limits can also be represented as shown in equations (8, 9). The velocity limit is defined in [3], assuming an input voltage of 12V. Were the project to be implemented on a physical robotic arm, the API for the robot would need to be explored so the limits of the optimization problem can be suitably defined within the context of the software/hardware specifications of the RX200.

$$\begin{aligned} |\dot{\psi}_{wt}| &< \dot{\psi}_{wt_{max}} \\ |\dot{\theta}_s| &< \dot{\theta}_{s_{max}} \\ |\dot{\theta}_e| &< \dot{\theta}_{e_{max}} \\ |\dot{\theta}_w| &< \dot{\theta}_{w_{max}} \end{aligned} \tag{8}$$

$$\dot{\psi}_{wt_{max}} = \dot{\theta}_{s_{max}} = \dot{\theta}_{e_{max}} = \dot{\theta}_{w_{max}} = 46\ [rev/min] \tag{9}$$

D. System Discretization

The linear system developed in the previous section was discretized using three methods, Euler, RK4, and exact discretization. Due to the linearity of the system and the value of the A matrix, methods of discretization result in the same answer, which result will be developed here. The implementation of these discretizations can be seen in the $discrete\_dynamics.m$ file in the $euler\_discretize()$, $rk4\_discretize()$ and $exact\_discretize()$ functions. The file can be found in the Gitlab repository (I).

The form of the Euler discretization is as shown in equation (10)

$$x_{k+1} = x_k + \delta \dot{x}_k \tag{10}$$

The exact discretization is as shown below. Observe that because $\mathbf{A}$ is a zero-matrix, the exact discretization and the Euler discretization produce exactly the same results.

$$x_{k+1} = \bar{\mathbf{A}}x_k + \bar{\mathbf{B}}u_k \tag{11}$$

$$\bar{\mathbf{A}} = e^{\mathbf{A}t} = 1,\ \bar{\mathbf{B}} = \int_0^T e^{\mathbf{A}(T-\tau)}d\tau\mathbf{B} = T\mathbf{B} \tag{12}$$

The next set of equations shows the RK4 value. First, in equations (13), the RK4 method is shown. Because $\mathbf{A} = 0$, any update to the x-value gets ignored, and the dynamics produce $\mathbf{B}u$.

$$x_{k+1} = x_k + \Phi \tag{13}$$
$$\Phi = \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{14}$$
$$k_1 = f(t, x) \tag{15}$$
$$k_2 = f(t + \frac{h}{2}, x + \frac{h}{2}k_1) \tag{16}$$
$$k_3 = f(t + \frac{h}{2}, x + \frac{h}{2}k_2) \tag{17}$$
$$k_4 = f(t + h, x + hk_3) \tag{18}$$
$$\tag{19}$$

$$k_1 = f(t, x) \qquad = \mathbf{0}x_k + \mathbf{B}u_k = \mathbf{B}u_k \tag{20}$$

$$k_2 = f(t + \frac{h}{2}, x + \frac{h}{2}k_1) \quad = \mathbf{0}(x + \frac{h}{2}k_1) + \mathbf{B}u_k = \mathbf{B}u_k \tag{21}$$

$$\Phi = \frac{h}{6}(\mathbf{B}u_k + 2\mathbf{B}u_k + 2\mathbf{B}u_k + \mathbf{B}u_k) = h\mathbf{B}u_k \tag{22}$$

And now it can be seen that all these methods provide the same result. Equation (23) is obtained by expanding (10). Equation (24) is developed from (11, 12). Equation (25) is developed from (13, 22)

$$x_k + \delta\dot{x}_k = x_k + \delta(\mathbf{A}x_k + \mathbf{B}u_k) \quad = x_k + \delta\mathbf{B}u_k \tag{23}$$
$$\bar{\mathbf{A}}x_k + \bar{\mathbf{B}}u_k \quad = x_k + T\mathbf{B}u_k \tag{24}$$
$$x_{k+1} \quad = x_k + h\mathbf{B}u_k \tag{25}$$

To see the simulation results of these developments, refer to the MATLAB code in the Gitlab repository found in section (I).

II. Solution Approach

A. Solution Summary

The solution to this minimum distance optimal control problem is defined by the next few subsections. The emphasis here is on the OCS2 solution development. Brief additional notes will be provided in each section regarding how things were or could have been changed for the simultaneous approach in MATLAB.

1) OCS2 Numerical Library: The solver to be used for the project is called OCS2, developed by Farbod Farshidian, a senior scientist at the Robotic System Laboratory of ETH Zurich [6], [7]. OCS2 stands for Optimal Control for Switched Systems. As suggested by the title, it is primarily designed to provide optimal control solutions for systems that have switched dynamics. It efficiently implements several algorithms, listed in the documentation as SLQ, iLQR, SQP, and PISOC (These methods are not thoroughly described in the documentation and aside from the algorithm used here, it is recommended that the reader does an internet search to learn more). The OCS2 library has efficient enough algorithms that some can be reasonably run on robots themselves [8].

The solver and optimization library are combined into the same package with this optimal control library. The method I have elected to use for this solver is called DDP, or Differential Dynamic Programming. More detail on this will be provided in the next section.

2) Solution Method: The solution method here, as mentioned above, is a form of Differential Dynamic Programming. It can be considered a multiple-shooting method, to put it in the scope of in-class terminology. The process involved in using DDP amounts to only a few steps. The first is to pick an arbitrary initial condition on the control input over the entire control trajectory. Using these controls, the system is simulated forward in time to get a value function V which is defined explicitly only at the terminal time. This value function is a cost-to-go from a given step t to the final state of the system at step N. At this point, having simulated the system forward, the value function can be evaluated at every step in time by simulating backwards. In the use of DDP, the partials of this value function V are recursively computed in the backwards simulation, then used for gradient descent [5]. This is similar to some of the LQR/Riccati methods used for homework assignments.

The specific application of DDP used here is the iLQR method, or iterative LQR [4]. Normal LQR solution methods only apply to linear systems. Since the iLQR method is a form of DDP, non-linearities can be handled as well due to the optimization over a whole trajectory. The approach using the MATLAB tools is formed as a simultaneous, direct-method approach. This approach is similar to iLQR in that the optimization considers the whole trajectory as a solution is being calculated. The MATLAB approach to this uses the $fmincon()$ function, which provides simple interfaces for defining constraints at each time point along the trajectory to optimize.

3) Discretization Method: Because this method is a form of DDP, there is no need for a discretization to be applied to the dynamics, making the resulting problem solvable in a continuous time form. When using the simultaneous approach however, the trajectory is not considered a continuous space. This requires constraints to be defined for each time-point, and partials are considered for each time-point as well.

B. Mathematical Description

1) Cost: The cost for the iLQR method in OCS2 is shown below in (26). It is formulated in a Bolsa form. The instantaneous cost is chosen specifically as the difference between the current point and the final point so as to encourage the formation of a straight line, which will always be the minimum distance solution. The cost on the control input is included here but is not strictly necessary. Although the control affects the cost on the position of the end-effector by way of the dynamics, it is indirect and thus the control isn't explicitly necessary here.

Including the control in the cost will help reduce speed as the end-effector approaches its target point, and to prevent large control inputs that would require far too much power on a physical robot arm. Another benefit of including the cost is that the optimization can first be aided by running it without constraints. If the cost on the input is included here, a close-to-optimal solution can be found before constraints are applied. Without doing this, it is possible for the solver to get stuck in local minima since it prioritizes constraints over cost.

The last few terms in the cost are linear costs on the state and the input at the final time. The terminal costs are quite simple and will encourage a zero-distance difference between the current point and the target point of the end-effector. The terminal cost on the control will encourage dropping the control to zero as the end-effector approaches its target. For the simultaneous method, the continuous-time integral would become a summation, with terms in the cost for terminal time on the state and for initial time on the control.

$$\begin{aligned}\min_{u(\cdot)} J(u(\cdot)) &= \int_0^{t_f} L(x(t),u(t))dt + \phi(x(t_f)) \\ &= \int_0^{t_f} \Big((q_{cur}(x(t)) - q_{target})^T (q_{cur}(x(t)) - q_{target}) \\ &+ u(t)^T R u(t) dt \Big) + (q_{cur}(x(t_f)) - q_{target}(x(t_f)))\end{aligned} \tag{26}$$

such that

$$x(t_0) = x_0 \tag{27}$$
$$\dot{x}(t) = f(x(t),u(t),t) \tag{28}$$
$$g_1(x(t),u(t),t) = 0 \tag{29}$$
$$g_2(x(t),t) = 0 \tag{30}$$
$$h(x(t),u(t),t) \geq 0 \tag{31}$$
$$R \succ 0 \tag{32}$$

where

$$x(t) = \begin{bmatrix}\psi_{wt}(t) & \theta_s(t) & \theta_e(t) & \theta_w(t)\end{bmatrix}^T \quad (33)$$
$$u(t) = \begin{bmatrix}\dot\psi_{wt}(t) & \dot\theta_s(t) & \dot\theta_e(t) & \dot\theta_w(t)\end{bmatrix}^T \quad (34)$$
$$q_{cur} = \begin{bmatrix}q_x(t) & q_y(t) & q_z(t) & 1\end{bmatrix}^T \quad (35)$$
$$q_{target} = \begin{bmatrix}q_{x_d} & q_{y_d} & q_{z_d} & 1\end{bmatrix}^T \quad (36)$$
$$g_1(x(t),u(t),t) \Rightarrow \text{State-input equality constraints} \quad (37)$$
$$g_2(x(t),t) \Rightarrow \text{State-only equality constraints} \quad (38)$$
$$h(x(t),u(t),t) \Rightarrow \text{Inequality constraints} \quad (39)$$

and

$$q_{cur} = T_b^{ee}(x(t))p_0 \quad (40)$$
$$p_0 = \begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}^T \quad (41)$$
$$L(x(t),u(t)) = \begin{matrix}(q_{cur}(x(t)) - q_{target})^T \\ \cdot(q_{cur}(x(t)) - q_{target}) \\ +u(t)^T R u(t)\end{matrix} \quad (42)$$
$$\phi(x(t_f)) = \begin{matrix}(q_{cur}(x(t_f)) - q_{target}(x(t_f)))^T \\ \cdot(q_{cur}(x(t_f)) - q_{target}(x(t_f)))\end{matrix} \quad (43)$$

Variable definitions include x_0, which is the initial state of the joint angles for the active problem. The matrix R weights the cost on the control input based on Bryson's method, where $R = diag(r,r,r,r)$ with $r = \frac{1}{(\frac{23\pi}{15})^2}$, the reciprocal of the max motor speed recommended for the Dynamixel servos, expressed in radians per second.

For q_{cur}, the (q_x, q_y, q_z) components represent the current position of the end-effector. For q_{target}, the $(q_{x_d}, q_{y_d}, q_{z_d})$ components represent the desired position of the end-effector. The time t_f is a fixed final time, which can be set as a parameter in the final solution attempt. Using a fixed final time is part of the OCS2 DDP implementation.

$$g_1(x(t),u(t),t) = u(t_f) = 0 \quad (44)$$

$$g_2(x(t),t) = T_b^{ee}(x(t_f))p_0 - p_d = 0 \quad (45)$$

$$h(x(t),u(t),t) = \begin{cases} -|\psi_{wt}| + 180 & \geq 0 \\ \theta_s + 108 & \geq 0 \\ -\theta_s + 113 & \geq 0 \\ \theta_e + 108 & \geq 0 \\ -\theta_e + 93 & \geq 0 \\ \theta_w + 100 & \geq 0 \\ -\theta_w + 123 & \geq 0 \\ -|\dot\psi_{wt}| + \frac{23\pi}{15} & \geq 0 \\ -|\dot\theta_s| + \frac{23\pi}{15} & \geq 0 \\ -|\dot\theta_e| + \frac{23\pi}{15} & \geq 0 \\ -|\dot\theta_w| + \frac{23\pi}{15} & \geq 0 \end{cases} \quad (46)$$

In order to implement the problem, several interfaces will be used from OCS2. Much of the problem can be set up through the OCS2 `OptimalControlProblem` struct. It defines pointers to all the key components of an optimal control problem, including

- Intermediate cost (instantaneous cost)
- Terminal cost
- Soft constraints (cost penalties)
  - Intermediate
  - Terminal
- Constraints (equality constraints)
  - Intermediate constraints
  - Terminal constraints
  - State-only or state-input options
- Inequality Constraints
  - Intermediate constraints
  - Terminal constraints
  - State-only or state-input options
- Dynamics
- Lagrangians that implement constraints

Part of the problem development can utilize what is called "auto-differentiation". Utilizing those tools could allow for the automatic calculation of first- and second-order partials that can be utilized in the problem. Due to the complexity of the code required to even use a basic form of this auto-differentiation feature, the partials were simply derived in MATLAB then converted into a C++ form for the OCS2 problem implementation.

After all the necessary objects (only the dynamics and a cost are required) are defined in the `OptimalControlProblem` struct, then the problem can be run using `DDP` after defining some settings for how that particular problem should be computed. Some example code from an OCS2 `iLQR` test file is shown below. The most significant part of the code will come in defining each of the parts that lead to this part. After the problem is run, the optimized control policy can be extracted using an OCS2-provided $PrimalSolution$ tool.

```
const auto algorithm = ocs2::ddp::Algorithm::ILQR;

// ddp settings
const auto ddpSettings =
  getSettings(algorithm,
              getNumThreads(),
              getSearchStrategy());

// dynamics and rollout
const ocs2::CircularKinematicsSystem systemDynamics;
const ocs2::TimeTriggeredRollout
  rollout(systemDynamics, rolloutSettings(algorithm));

// instantiate
ocs2::ILQR
  ddp(ddpSettings, rollout, problem, *initializerPtr);

...

// run ddp
ddp.run(startTime, initState, finalTime);
```

2) Dynamics: The dynamics are formed as show before as a direct control input. $\dot\psi_{wt}$ is the angular velocity for the waist joint, $\dot\theta_s$ is the angular velocity for the shoulder joint, $\dot\theta_e$ is the angular velocity for the elbow joint, and $\dot\theta_w$ is the angular velocity for the wrist joint. This is formed in the same way for OCS2 and MATLAB problems.

$$\dot x(t) = u(t) = \begin{bmatrix}\dot\psi_{wt}(t) & \dot\theta_s(t) & \dot\theta_e(t) & \dot\theta_w(t)\end{bmatrix}^T \quad (47)$$

3) Partials: Partials for the cost function were evaluated in MATLAB and verified using the $jacobianest$ function [9]. Originally, a linear term was given for the terminal cost, but the dimension of the cost does not come out to be scalar. Because the goal is still to minimize distance to the desired position, the terminal cost will also be quadratic so it is continuously differentiable. Forming the terminal cost this way favors simplicity in code and in the problem itself.

As mentioned before, for the OCS2 problem implementation, the MATLAB partials were ported into a C++ form. The Jacobian of the cost with respect to the state was verified using a simple test function for the C++ version of the partials. Part of the development of the OCS2 problem also required the use of the Hessian of the cost with respect to state and input. Those matrices are not shown here for brevity. To view the code implementations of the Hessians, use either the $matlab/cost\_partials.m$ script or take a look at the $src/costs/MinDistRunningCost.cpp$ file in the GitLab repository.

III. Results and Discussion

This section covers the results of the attempted solutions using OCS2 and MATLAB. Neither was entirely successful at generating the expected trajectories. Perceived advantages and disadvantages of each will be discussed, as well as the process of implementing solutions up to the point where the development was halted. After those sections, the simulation results will be shown.

A. OCS2: iLQR Implementation

The OCS2 solution attempt was definitely more mentally taxing than the MATLAB approach given that OCS2 is relatively new as a control toolbox, undeveloped in some cases, and at the same time, deeply complex at first glance. When starting to write the code for the OCS2, the first need was to install Ubuntu 20.04, ROS, and to learn how to use some tools for building the OCS2 library in the first place. After that, a custom package for this project was added.

From there, there digging through code and examples was essential to getting an optimal control problem to run at all. There is very little documentation for OCS2, both on the documentation website and within the code itself. Comments were not a very common thing to see for the setup of examples in a variety of packages. Relying heavily on the example setup from the $ocs2/ocs2\_ddp/test/CircularKinematicsTest.cpp$ file, a somewhat working example of the control problem was developed for OCS2. The minimum distance costs and some constraints were developed by inheriting from pre-existing classes in OCS2. Some simple utilties headers were also created for implementing the forward kinematics and defining the properties of the RX200.

Once the basics were set up, costs and partials were verified. At this point, initial testing was possible. A basic test case was set up such that the RX200 would start in its home position (which is as shown in figure (2)) and move to the position $\begin{bmatrix} 0 & 300 & 100 \end{bmatrix}^T$. It was quickly apparent that something was wrong, when on the first evaluations of the first iteration of the control problem had increased the cost to a value far beyond what is actually attainable in the RX200 workspace given the cost definitions that were provided. At this point, constraints on the input were not implemented yet, but the input was quadratically weighted in the cost. Thus it was anticipated that the high input values should not have been a preference in the solver. Nevertheless, the cost was unstable and so were the state and inputs at that time.

Many ideas were tested to prevent this issue. Creating a desired trajectory defined at each time step in the solution (thus making the problem more like the simultaneous method with discrete trajectory requirements), weighting the R matrix, getting rid of and/or adding constraints or terminal costs, testing partials again, etc... The most effective change was to weight the R matrix according to Bryson's method for LQR problems. After adjusting that, some oddly specific values were found such that the problem would no longer blow up.

Now being in this position, the solution still couldn't be extracted because of segmentation fault occuring after the solution was finalized by the solver. After some searching through the OCS2 code, part of a class constructor was found that did not properly handle cases in which a default value for an input trajectory was assigned, but not resized correctly. To form the solution, the OCS2 code expected that the input trajectory vector was the same size as the provided time and state trajectories provided alongside the input trajectory. An adjustment was made to alter this code, and then the primal solution could finally be extracted from the solver.

The solution obtained in OCS2 is far superior to the one the MATLAB implementation was able to achieve (remember that both implementations are far from being correct). The solution OCS2 yields got most of the way to the goal point, as can been seen below in figure (III-C). Other sets of inputs and goal poses were attempted, but the cost seems to blow up for anything but the most specific conditions, and clearly the inequality constraints are not held.

B. MATLAB: Direct Method Simultaneous Approach

While the MATLAB approach was adopted from one of the homework assignments earlier in the class, there were still plentiful issues in the resulting solution trajectories. There must be an undetected oversight in the implementation and changing of the previous code, especially considering how well-developed the MATLAB tools are for this kind of problem. Here, the optimization takes a rather long time to complete even 100 iterations. As can been seen in figure (III-C), the results really make no sense at all. The only guarantee about the solution is that it will start and end in the correct places. There is not much else of any significance regarding the results there.

C. Simulation Results

Here are the results of the solution trajectories for the attempted OCS2 and MATLAB solutions. Both images were

$$\frac{\partial L}{\partial x}^T = \begin{bmatrix} -2(q_{y_d}cos(\psi_{wt}) - q_{x_d}sin(\psi_{wt}))(d_2cos(\theta_e+\theta_s) + d_1cos(\theta_s) + d_0sin(\theta_s) + d_3cos(\theta_e+\theta_s+\theta_w)) \\ \\ 2(d_2cos(\theta_e+\theta_s) + d_1cos(\theta_s) + d_0sin(\theta_s) + d_3cos(\theta_e+\theta_s+\theta_w))(q_{z_d} + d_2sin(\theta_e+\theta_s) - d_0cos(\theta_s) + d_1sin(\theta_s) \\ +d_3sin(\theta_e+\theta_s+\theta_w)) - 2cos(\psi_{wt})(d_2sin(\theta_e+\theta_s) - d_0cos(\theta_s) + d_1sin(\theta_s) + d_3sin(\theta_e+\theta_s+\theta_w))(d_2cos(\theta_e+\theta_s)cos(\psi_{wt}) \\ -q_{x_d} + d_1cos(\psi_{wt})cos(\theta_s) + d_0cos(\psi_{wt})sin(\theta_s) + d_3cos(\theta_e+\theta_s+\theta_w)cos(\psi_{wt})) - 2sin(\psi_{wt})(d_2sin(\theta_e+\theta_s) - d_0cos(\theta_s) \\ +d_1sin(\theta_s) + d_3sin(\theta_e+\theta_s+\theta_w))(d_2cos(\theta_e+\theta_s)sin(\psi_{wt}) - q_{y_d} + d_1cos(\theta_s)sin(\psi_{wt}) \\ +d_0sin(\psi_{wt})sin(\theta_s) + d_3cos(\theta_e+\theta_s+\theta_w)sin(\psi_{wt})) \\ \\ 2(d_2cos(\theta_e+\theta_s) + d_3cos(\theta_e+\theta_s+\theta_w))(q_{z_d} + d_2sin(\theta_e+\theta_s) - d_0cos(\theta_s) + d_1sin(\theta_s) + d_3sin(\theta_e+\theta_s+\theta_w)) - 2cos(\psi_{wt}) \\ *(d_2sin(\theta_e+\theta_s) + d_3sin(\theta_e+\theta_s+\theta_w))(d_2cos(\theta_e+\theta_s)cos(\psi_{wt}) - q_{x_d} + d_1cos(\psi_{wt})cos(\theta_s) + d_0cos(\psi_{wt})sin(\theta_s) \\ +d_3cos(\theta_e+\theta_s+\theta_w)cos(\psi_{wt})) - 2sin(\psi_{wt})(d_2sin(\theta_e+\theta_s) + d_3sin(\theta_e+\theta_s+\theta_w))(d_2cos(\theta_e+\theta_s)sin(\psi_{wt}) \\ -q_{y_d} + d_1cos(\theta_s)sin(\psi_{wt}) + d_0sin(\psi_{wt})sin(\theta_s) + d_3cos(\theta_e+\theta_s+\theta_w)sin(\psi_{wt})) \\ \\ d_3q_{y_d}cos(\theta_e-\psi_{wt}+\theta_s+\theta_w) - 2d_2d_3sin(\theta_w) + d_3q_{x_d}sin(\theta_e-\psi_{wt}+\theta_s+\theta_w) \\ +2d_3q_{z_d}cos(\theta_e+\theta_s+\theta_w) - d_3q_{y_d}cos(\psi_{wt}+\theta_e+\theta_s+\theta_w) - 2d_0d_3cos(\theta_e+\theta_w) \\ +d_3q_{x_d}sin(\psi_{wt}+\theta_e+\theta_s+\theta_w) - 2d_1d_3sin(\theta_e+\theta_w) \end{bmatrix} \tag{48}$$

$$\frac{\partial \phi}{\partial x} = \frac{\partial L}{\partial x} \tag{49}$$

$$\frac{\partial L}{\partial u} = 2r\begin{bmatrix}\dot{\psi}_{wt} & \dot{\theta}_s & \dot{\theta}_e & \dot{\theta}_w\end{bmatrix}, \text{ where } r \text{ is a constant, the diagonal element of } R \tag{50}$$

$$\frac{\partial \phi}{\partial u} = \mathbf{0} \in \mathbb{R}^4 \tag{51}$$

$$\frac{\partial f(x,u,t)}{\partial x}^T = \mathbf{0} \in \mathbb{R}^4 \tag{52}$$

$$\frac{\partial f(x,u,t)}{\partial u}^T = \mathbf{1}_4 \tag{53}$$

generated in MATLAB. The trajectory information from OCS2 results were simply stored as a CSV file then loaded into MATLAB for plotting.

The OCS2 solution looks ”nice”. The solid red line shows the trajectory of the solution, where the red dotted line is the straight-path solution that was expected from the optimization. The OCS2 solution actually does do a fair job of heading towards the desired goal pose along its whole trajectory. The MATLAB solution on the other hand is definitely not nice. The starting and ending positions are dead on as they were constrained to do so (at least the MATLAB constraints could be verified to work), but everything else in between has no meaning whatsoever. It seems that the links of robot could immediately have jumped to a ”good” solution, then iterated over similarly costing other ”good” solutions.

The moral of the story here is that easier probably means better for a first-time somewhat-self-guided optimal control problem definition, creation, and solution. Given the same amount of time in MATLAB that was put into figuring out OCS2 would likely have resulted in a more complete and competent solution of this minimum distance problem.

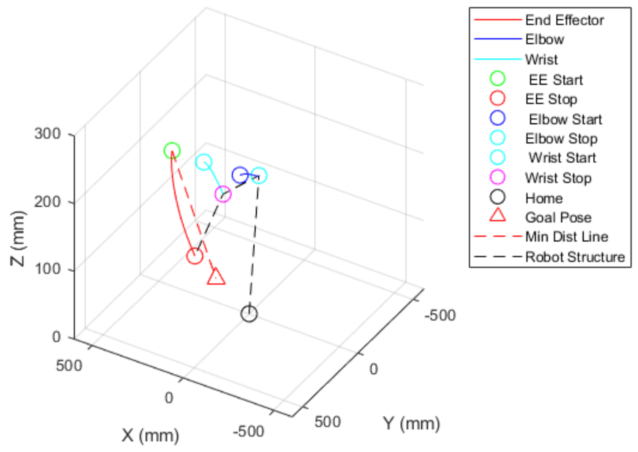

Fig. 3. OCS2 Solution

IV. Conclusion

Through the course of attempting to finish this project, I was able to learn about kinematics and how complex setting up a system of even the most simple dynamics can be. There was far more time spent in the latter to no avail in either OCS2 or MATLAB. The setup developed for OCS2 was able to produce a somewhat reasonable trajectory even though the problem was not solved completely. The MATLAB implementation has a lot of problems. Those persisting problems can be attributed to the distinct lack of time remaining to complete the project and still provide a semi-decent report regarding my efforts.

Clearly, any further development for this project would immediately turn to actually having either of the simulation environments work. From there, the intention was to generate trajectories that could be published via ROS nodes, visualizing

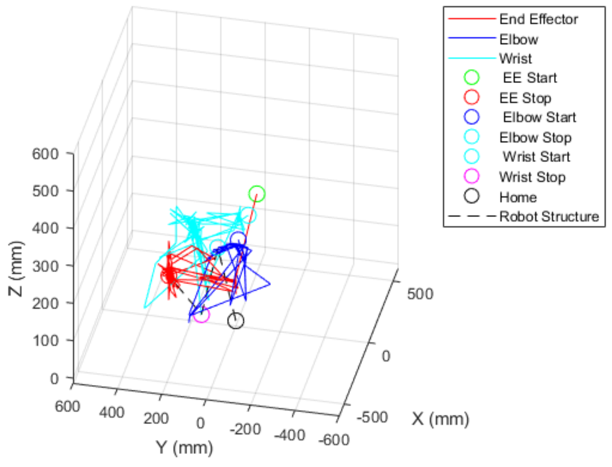

Fig. 4. MATLAB Solution

the RX200 in ROS's RVIZ tool. If that capability was implemented, I would develop a more continuous and parameter-based way to run the simulation without having to edit code and rebuild every time a change was made. Improvements after that could be made by including models for self-collision avoidance and other obstacle avoidance along the trajectory. Such a feature may imply a MPC-based approach.

Appendix I
Link to GitLab Repository

Go here to access the GitLab code for this project

References

[1] Taylor, G. (n.d.). 3D Kinematics. SI475: Intelligent Robotics. Retrieved May 2, 2023, from https://www.usna.edu/Users/cs/taylor/courses/si475/notes/3dkinematics.pdf
[2] ReactorX 200 robot arm. X-Series Robotic Arm. (n.d.). Retrieved May 2, 2023, from https://www.trossenrobotics.com/reactorx-200-robot-arm.aspx
[3] ROBOTIS. (n.d.). Robotis e. ROBOTIS e-Manual. Retrieved May 2, 2023, from https://emanual.robotis.com/docs/en/dxl/x/xm430-w350/
[4] de Wolf, T. (2017, August 28). The iterative linear quadratic regulator algorithm. studywolf. Retrieved May 2, 2023, from https://studywolf.wordpress.com/2016/02/03/the-iterative-linear-quadratic-regulator-method/
[5] Y. Tassa, N. Mansard and E. Todorov, "Control-limited differential dynamic programming," 2014 IEEE International Conference on Robotics and Automation (ICRA), Hong Kong, China, 2014, pp. 1168-1175, doi: 10.1109/ICRA.2014.6907001.
[6] Farbod Farshidian — IEEE Xplore author details. (n.d.). Retrieved May 2, 2023, from https://ieeexplore.ieee.org/author/37085428006
[7] Farshidian, F. (n.d.). Overview. Overview - OCS2 1.0.0 documentation. Retrieved May 2, 2023, from https://leggedrobotics.github.io/ocs2/overview.html
[8] Leggedrobotics. (n.d.). Leggedrobotics/OCS2: Optimal Control for Switched Systems. GitHub. Retrieved May 2, 2023, from https://github.com/leggedrobotics/ocs2
[9] Samuellab. (n.d.). Magatanalyzer-MATLAB-analysis/jacobianest.m at master · Samuellab/Magatanalyzer-MATLAB-analysis. GitHub. Retrieved May 2, 2023, from https://github.com/samuellab/MAGATAnalyzer-Matlab-Analysis/blob/master/utility%20functions/DERIVESTsuite/jacobianest.m
[10] sanju6890. (2021, February 11). Equation of a line in 3D. GeeksforGeeks. Retrieved May 2, 2023, from https://www.geeksforgeeks.org/equation-of-a-line-in-3d/#