# Assignment 5: Smooth Trajectories and Control

Robot Kinematics and Dynamics

Prof. Jeff Ichnowski

Shahram Najam Syed

Yuemin Mao

# Contents

# 1 Overview

This assignment reinforces the following topics:

- Improved Trajectories for Robot Control
- Dynamics for Point Masses

# 2 Background

## 2.1 Trajectories

When controlling a robotic arm, it is usually important to handle things in a way that is more sophisticated than just 'banging' the arm from one configuration to another. The general idea is to not command anything a robot physically do. Although we have not yet discussed dynamics in the class, dynamics affect the ability to control the position or velocity of any manipulator in the physical world. In short:

- Real robots have mass, $m$
- $F = ma$
- $F$ is limited, so we should be careful about our desired $a$.

One way to mitigate effects of the dynamic limits of an arm is to use *trajectories* to smoothly move the arm from one configuration to another. (When thinking about trajectories, you can pose them as trajectories in work space or configurations space. Here we'll just think of trajectories in configuration space, but in the code problems you'll work with workspace trajectories as well.)

### 2.1.1 Bang-Bang Control (No Trajectories)

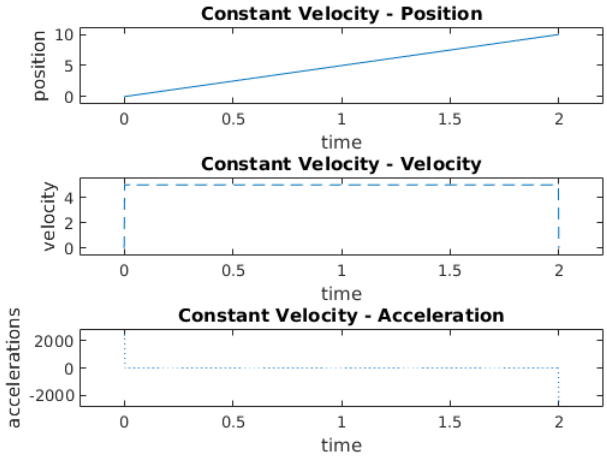
The simplest method is to ignore the pesky bounds of reality and rely on the underlying controller of the robot's actuators to 'catch up' to your commands. Depending on the specific application, and they way the low-level controllers are tuned, this can actually work. However, it has the disadvantage that the robot's motion is less predictable since the low-level controllers dictate the details of how the robot actually moves between the start and goal position – which is something that you often want to have control over! Additionally, with this approach you are usually 'jerking' the arm aggressively which can be dangerous and put unnecessary wear and tear on the arm's joints.

### 2.1.2 Constant Velocity (Better Than Nothing)

The next-simplest method you might think of would be to respect the reality that a robot's joints can't move with infinite speed. Looking at the case of moving a single DOF between two angles, we can choose an amount of time to take to move between the angles and command a corresponding constant velocity.

$T_{\text{move}} = (\theta_{\text{end}} - \theta_{\text{start}})/V_{\text{cruise}}$

This is a step in the right direction, but it has issues where we start and stop. To see the problem, let's think about the position, velocity, and acceleration.




Because we jump from stop to a constant velocity and back again, at the ending steps we see big spikes in acceleration. In theory, these spikes are infinite acceleration, and in practice we will see some large finite value depending on how finely we discretize time (size of our $\Delta t$ in the simulation or arm controller). Large accelerations are undesirable because if the arm has any mass, then we are commanding large torques that are usually outside the capabilities of the robot. Again, the goal of using trajectories is to always command things the arm can physically do, so that it is playing 'catch up' as little as possible.

## 2.2 Smoother Trajectories

Recall that when controlling a large motion of the joints of a robot arm, you can ensure your commands are more physically realizable (and therefore the robot will have a smoother resulting motion) if you subdivide the motion into small steps.

In the last assignment, we used path segments where the joints moved with a constant speed. This simple approach is a step in the right direction, but resulted in theoretically infinite accelerations at the step changes in velocity at the beginning and end of the motion.

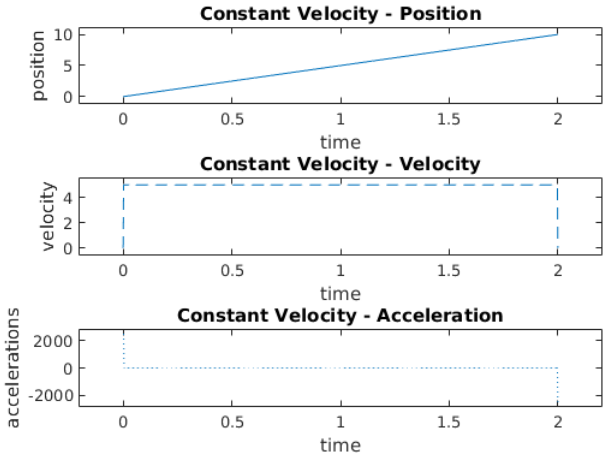
Here we look at the case of moving a single joint between two angles $\theta_0$ and $\theta_F$ over the time interval $[t_0, t_F] = [0, 2]$. (When moving multiple joints, this is done independently for each joint.)

The equations for position and velocity of the joint during the interval $[t_0, t_F]$ are:

$$\begin{aligned} q(t) &= v_c(t - t_0) + q_0 \\ \dot{q}(t) &= v_c \end{aligned}$$

The expression for $q(t)$ can be found by inspection (using the formula for a line), or by integrating $\dot{q}(t)$ with appropriate boundary conditions. Also note that given $q_0$ and $q_f$, you can easily solve for $v_c$.

Below we visualize the joint position, velocity, and acceleration for this *constant velocity* trajectory.

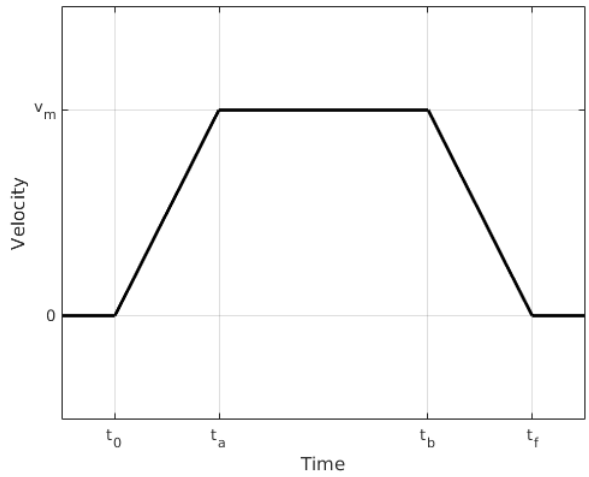



These large accelerations are undesirable because if the arm has any mass then we are commanding large torques that are usually outside the capabilities of the robot. Again, the goal of using trajectories is to always command things the arm can physically do.

### 2.2.1 Trapezoidal Velocity

A simple solution is to *ramp up* the velocity to the maximum velocity, so the velocity signal is continuous and therefore the acceleration is bounded. You can define the velocity as a piecewise signal, given a ramp time $t_r$ and maximum velocity $v_m$. (For simplicity, define $t_a = t_0 + t_r$ and $t_b = t_f - t_r$.)

$$\dot{q}(t) = \begin{cases} v_m(t - t_0)/t_r & t_0 \leq t \leq t_a \\ v_m & t_a < t \leq t_b \\ v_m(t_f - t)/t_r & t_b < t \leq t_f \end{cases}$$




Note that if $q_0$, $q_f$, $t_0$, and $t_f$ are specified, then $t_r$ fully determines $v_m$ (or $v_m$ determines $t_r$, depending on the problem constraints). For example, you may have a joint velocity limit that

cannot be exceeded. In other cases, you may define $t_f$, the length of the motion, given a particular maximum acceleration and desired ramp time $t_r$. Setting these values is application-specific, but we can determine a relationship among them by solving for $q(t)$.

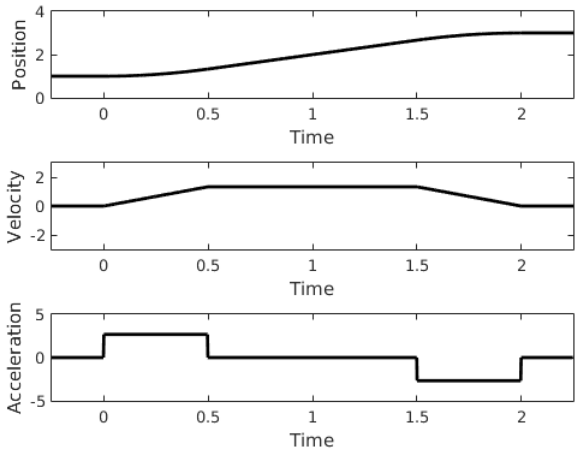
First, consider $q(t)$ for $t_0 \leq t \leq t_a$:

$$\begin{aligned} q(t) &= q_0 + \int_{t_0}^{t} \dot{q} \\ q(t) &= q_0 + \int_{u=t_0}^{t} \frac{v_m}{t_r}(u - t_0) \\ q(t) &= q_0 + \left( \frac{v_m}{t_r} \left( \frac{u^2}{2} - t_0 u \right) \Big|_{u=t_0}^{t} \right) \\ q(t) &= q_0 + \frac{v_m}{2t_r}(t - t_0)^2 \end{aligned}$$

Similarly, we can integrate the other two piecewise linear segments of the velocity curve to get

$$q(t) = \begin{cases} q_0 + \frac{v_m}{2t_r}(t - t_0)^2 & t_0 \leq t \leq t_a \\ q(t_a) + v_m(t - t_a) & t_a < t \leq t_b \\ q(t_b) - \frac{v_m}{2t_r}(t_f^2 - 2t_f t + t^2 - t_r^2) & t_b < t \leq t_f \end{cases}$$

This results in the following joint position, velocity, and acceleration curves – note the bounded value for the acceleration.




Note that one can solve for $q(t_f)$, and substitute the expressions for $t_a$ and $t_b$:

$$\begin{aligned} q_f &= q(t_f) = q_0 + v_m \left( \frac{(t_a - t_0)^2 + (t_f - t_b)^2}{2t_r} + t_b - t_a \right) \\ q_f &= q_0 + v_m \left( t_f - t_0 - t_r \right) \end{aligned}$$

And finally, rearrange and solve for $v_m$ in terms of $t_f$, $t_r$, $q_0$, and $q_f$.

$$v_m = (q_f - q_0)/(t_f - t_0 - t_r)$$

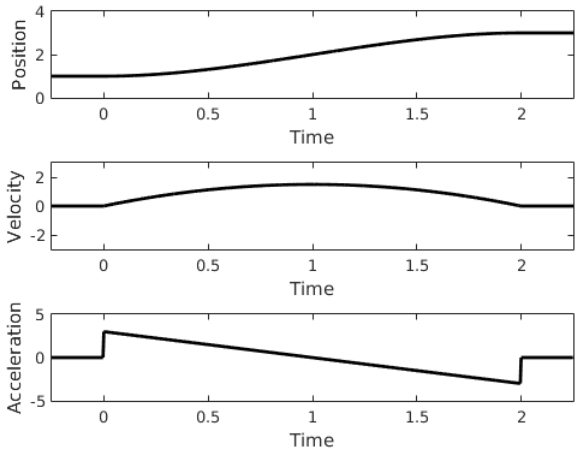
These trajectories are sometimes called S-curves, since the position follows an S-shape with a parabolic beginning / end and linear middle section. This is what a lot of industrial systems and robots do. Overall, it is a good balance between respecting the limits physical actuators without being too complex.

### 2.2.2 Spline Trajectories

Another method to achieve smooth motions through waypoints is to use a *spline* interpolation. Given a set of waypoints, a spline is a smooth curve that passes through these points, often with some boundary conditions at the start and end. Note that this interpolation gives a curve in joint space, not joint velocity space.

Another benefit to splines over trapezoidal velocity profiles is that for more than two waypoints, the velocity in a spline does not need to be zero at the "middle" (non end) waypoints.

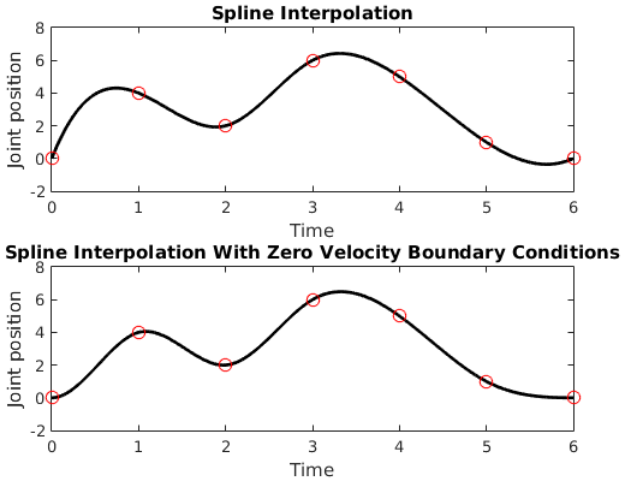
In the figure below, we show the position, velocity, and acceleration for a *cubic spline* between these two waypoints, along with the added boundary condition constraint of zero velocity at the endpoints. (Note – other implementations of spline interpolation allow you to add zero acceleration constraints at the endpoints as well, reducing the jump in acceleration at the ends.




A *cubic* spline in particular is constructed from piecewise third-order polynomials which pass through the waypoints. The equations are straightforward but involved, so we will not expect you to write these yourself. Fortunately, MATLAB's built in 'spline' function can generate a spline interpolation for a given set of waypoints.

As a more complex example, the figure below shows a spline through a number of waypoints. The matlab code to get the y trajectory is below; notice we also show the addition of 'zero velocity' constraints by adding a first and last column of zeros to the waypoints, per the MATLAB 'spline' function documentation.

```
%% Spline through lots of points
waypoints = [0 4 2 6 5 1 0];
waypoint_times = 0:6;
interpolated_times = linspace(0,6,1000);
trajectory = ...
    spline(waypoint_times, waypoints, interpolated_times);
trajectory_zero_vel_ends = ...
    spline(waypoint_times, [0 waypoints 0], interpolated_times);
```




Two final notes: First, splines no longer constrain the motion to straight lines (in joint space) between waypoints, and so the robot can "swing out" as it passes through waypoints that are close together. However, this is often a non-issue.

Also, we won't cover this in the class, but if we keep going down into further derivatives, we see that we still have infinite spikes in the derivative of acceleration which is called jerk. In case you are wondering, the next 3 derivatives are called snap, crackle, and pop. I'm not making this up.

If we want to bound the jerk of a given move, we can specify beginning and end positions, velocities, and accelerations and solve for a 5th-order polynomial that meets these conditions. Such a polynomial will minimize the jerk, and thus command continuous accelerations throughout the trajectory.

We're not going to make you do this in your work, but it's worth knowing about if you want to try to be aware of, or implement, the state-of-the-art in motion control. This approach all started when researchers from MIT in the 1980s did some studies of humans and found that we roughly follow minimum jerk trajectories when we move our limbs. More recently, polynomial trajectories are showing up in other places in robotics. Minimum snap trajectories lie at the heart of a lot of the latest control techniques for quadrotors and small drones, and minimum crackle trajectories are used to gracefully control ballbots.

## 2.3 Lagrangian

Recall that the Lagrangian is the Kinetic Energy of the system minus the Potential Energy.

$$\mathcal{L}(q, \dot{q}) = T(q, \dot{q}) - V(q, \dot{q})$$

where $T$ stands for the Kinetic Energy and $V$ stands for the Potential Energy.

## 2.4 Euler-Lagrange Equations

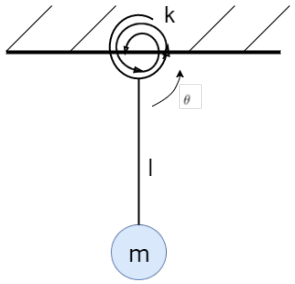
$$\frac{d}{dt}\left(\frac{d\mathcal{L}}{d\dot{q}_j}\right) - \frac{d\mathcal{L}}{dq_j} = \tau_j$$

where $\tau_j = 0$ for simplicity.

# 3 In Class Questions

The following questions will be done in class, as a part of a group. Your group's answer will still need to be turned in with the rest of your assignment, however unlike the rest of the work this is allowed to be done in groups.

1) Spring Pendulum Lagrangian Dynamics
Given the Spring Pendulum illustrated below.




a. [5 points] Derive the Kinetic Energy of the System $T$ in terms of $\theta$ and $\dot{\theta}$.
b. [5 points] Derive the Potential Energy of the System $V$ in terms of $\theta$ and $\dot{\theta}$.
c. [2 points] Derive the Lagrangian $\mathcal{L}$ in terms of $\theta$ and $\dot{\theta}$.
d. [5 points] Derive the Euler-Lagrange Equation using the Lagrangian.

# 4 Written Questions

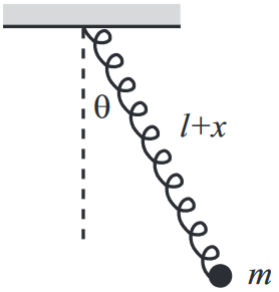
For the following problems, fully evaluate all answers unless otherwise specified.

Answers for written questions must be typed. We recommend LaTeX, Microsoft Word, OpenOffice, or similar. However, diagrams can be hand-drawn and scanned in.

Unless otherwise specified, **all units are in radians, meters, and seconds**, where appropriate.

1) Different Spring Pendulum

We went over the derivations for the mass spring and pendulum systems in the lecture videos. This time we shall combine the two systems with a spring pendulum system. The spring lies in a straight line where it is wrapped around a rigid massless rod. The equilibrium length of the spring is $l$, but at time $t$, the length of the spring is $l + x(t)$. The angle of the spring with respect to the vertical is $\theta(t)$.




a. [5 points] Derive the Kinetic Energy of the System $T$ in terms of $x, \dot{x}, \theta, \dot{\theta}$.

b. [5 points] Derive the Potential Energy of the System $V$ in terms of $x, \dot{x}, \theta, \dot{\theta}$.

c. [2 points] Derive the Lagrangian $\mathcal{L}$ in terms of $x, \dot{x}, \theta, \dot{\theta}$.

d. [5 points] Derive one Euler-Lagrange Equation by differentiating with respect to $x$.

e. [5 points] Derive the other Euler-Lagrange Equation by differentiating with respect to $\theta$.

f. [3 points] Please give a simple explanation for each of the components in both of the Euler-Lagrange equations you derived above.

# 5 Code Questions

Copy the Code Handout folder to a location of your choice. Open your Python IDE or terminal and navigate to that location. Whenever you work on the assignment, make sure you're in this directory and set up the necessary environment by running any required setup files.

In the previous part, we focused on straight-line trajectories, both in workspace and in joint configuration space. We found that defining paths in the workspace is more computationally intensive than defining paths in the joint configuration space. Although carefully controlling workspace paths can be important, it is often enough to define some primary waypoints in the workspace, identify the corresponding joint angles for these waypoints, and then restrict ourselves to joint space to simplify the generation and following of the trajectory.

We also found that when controlling robots using trajectories with piecewise constant velocities, the abrupt changes in velocity can lead to infinite "spikes" in the desired accelerations for the joints, which is physically unrealistic and can lead to unstable behavior.

The focus of this section will be generating trajectories that pass through specified configuration space waypoints. In addition to the basic constant-velocity trajectories from the previous assignment, we will add a function to generate trajectories with trapezoidal velocity profiles and a function to generate cubic spline interpolation of waypoints.

Each of these functions will take as input a matrix of joint (configuration) space waypoints. The first column of this matrix is the starting waypoint, the last column is the ending waypoint, and any intermediate columns are intermediate points the robot must pass through.

The other input for these functions is a vector of times at which to hit each point (the starting time is assumed to be zero) and a control frequency (how many joint configurations per second should be in the resulting trajectory matrix).

1) Piecewise Constant Velocity Trajectories *15 points*

In the folder 'ex_01', you will fill in 'trajectory_const_vel.py'. This should create a series of straight-line, constant velocity trajectories between the waypoints that are passed in. Feel free to use the code from your previous assignment to help with this problem.

To validate your work, run 'validate_trajectory_const_vel.py'. Your results should overlay the examples for each test case. **Include your plots in your writeup!**

2) Trapezoidal Velocity Trajectories *30 points*

In the folder 'ex_02', you will fill in 'trajectory_trap_vel' function. This should create a series of trapezoidal velocity trajectories between the waypoints that are passed in.

This function has an additional argument, the "ramp duty cycle." This is used to define the percent of each segment used to ramp up (and also the percent used to ramp down) the velocity. This argument must be between 0 (constant velocity trajectory) and 0.5 (triangular velocity profile), and can be used to find the ramp time for each segment.

To validate your work, run 'validate_trajectory_trap_vel.py'. Your results should overlay the examples for each test case. **Include your plots in your writeup!**

3) Cubic Spline Interpolation *15 points*

In the folder 'ex_03', you will fill in 'trajectory_spline' function. This should create a single spline (for each joint) that passes through the given waypoints.

At the first and last waypoint, the velocity should be zero. Refer to the example in the background material for instructions on how to enforce this constraint with Python's 'scipy.interpolate.CubicSpline' function.

To validate your work, run 'validate_trajectory_spline.py'. Your results should overlay the examples for each test case. **Include your plots in your writeup!**

# 6 Submission Checklist

- ☐ Create a PDF of your answers to the written questions named writeup.pdf.
- ☐ Ensure writeup.pdf is in the same folder as your submission scripts.
- ☐ Create a compressed file handin-5.tar.gz containing your code and writeup.
- ☐ Upload handin-5.tar.gz to the assignment gradescope.
- ☐ Upload writeup.pdf separately to gradescope.
- ☐ Congratulations on finishing this assignment!