Topics Covered:

- Manipulator Trajectories
- Multiple Waypoints w/ Desired Velocities

Additional Reading:

- LP Chapter 9
- Craig Chapter 7

Review

So far we have covered how to:

1. convert two joint configurations (θ_i and θ_f) to a sufficiently smooth curve, described using a polynomial of the appropriate degree.
2. choose the duration of the trajectory (in time) to satisfy actuation limits.

Reasons to complicate this further:

- if we want the desired end-points to be specified in the end-effector group space (not joint space)
- if the workspace has obstacles, need to ensure that they are avoided
- other workspace constraints such as actuation limits and joint limits that further restrict the space of feasible trajectories

Group Space Trajectories

Question: What if the trajectory is given the end-effector group space ($g_e \in SE(3)$) and not the joint space ($\theta \in M$)? We will denote this desired trajectory as $g_e^*(t)$.

- Need to find $\theta^*(t)$ such that $g_e^*(t) = g_e(\theta^*(t))$ (or is as close as possible)

Answer: One approach is to "spline" together the inverse kinematics solution for a set of waypoints/knotpoints to get a candidate joint trajectory.

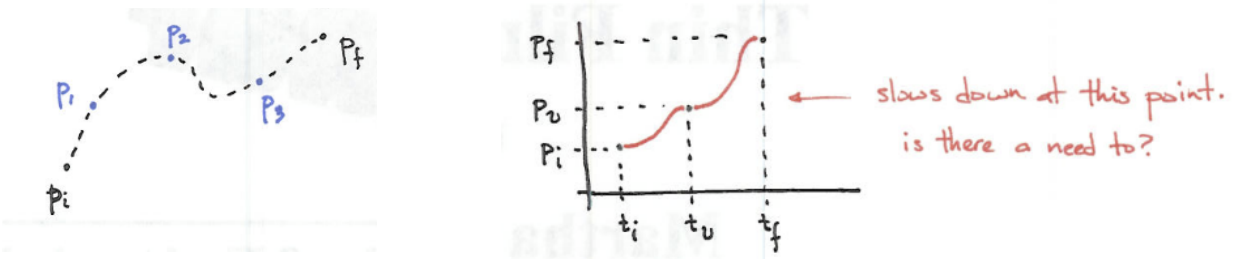
1. solve an inverse kinematics problem for each $g^*(t_k)$ to get $\theta^*(t_k)$ for $t_0, t_1, \ldots, t_n$ where $n =$ total waypoints
2. interpolate/concatenate smooth trajectories $\theta^*(t)$ through the $\theta^*(t_k)$.

In this approach, we need to be aware of inverse kinematics solutions, since they may not be unique; could lead to joint configuration flip-flopping.

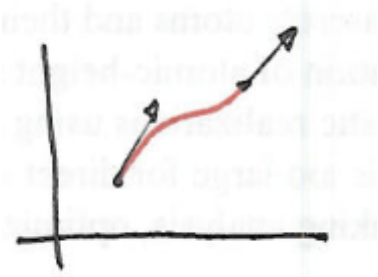
Let's assume that step 1 is done (we have $\theta^*(t_k)$ for $t_0, t_1, \ldots, t_n$). Let's consider how to interpolate/concatenate these trajectories.

Scalar Case

If we connect cubic splines between each pair of adjacent way points, we get a trajectory that connects the initial and final configurations with a series of cubics with zero velocity conditions. While this works, the trajectory is not ideal since it slows down at each waypoint.



Instead, we can achieve non-trivial velocities at waypoints by constraining the end-points of each spline. These constraints generalize to:



$$p(0) = p_i \qquad p(t_f) = p_f$$
$$\dot{p}(0) = \dot{p}_i \qquad \dot{p}(t_f) = \dot{p}_f$$

This leads to the general problem from the last lecture:

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix}}_{A(t_f)} \underbrace{\begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{Bmatrix}}_{\vec{a}} = \underbrace{\begin{Bmatrix} p_i \\ p_f \\ \dot{p}_i \\ \dot{p}_f \end{Bmatrix}}_{\vec{p}_0}$$

$$\underbrace{\begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{Bmatrix}}_{\vec{a}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -3/t_f^2 & 3/t_f^2 & -2/t_f & -1/t_f \\ 2/t_f^3 & -2/t_f^3 & 1/t_f^2 & 1/t_f^2 \end{bmatrix}}_{P(t_f)} \underbrace{\begin{Bmatrix} p_i \\ p_f \\ \dot{p}_i \\ \dot{p}_f \end{Bmatrix}}_{\vec{p}_0}$$

Explicitly, this is

$$a_0 = p_i \qquad a_2 = \frac{3}{t_f^2}(p_f - p_i) - \frac{2}{t_f}\dot{p}_i - \frac{1}{t_f}\dot{p}_f$$

$$a_1 = \dot{p}_i \qquad a_3 = \frac{2}{t_f^3}(p_i - p_f) + \frac{1}{t_f^2}(\dot{p}_i + \dot{p}_f)$$

Here, we would obtain $\dot{p}_i$ and $\dot{p}_f$ from the specification of our desired trajectory.

Using the Jacobian Pseudo-Inverse for Manipulators

For manipulators, we can simplify this process a bit by using our manipulator Jacobian to solve for $\dot{\theta}$. The process is as follows:

1. As before, we start with $g_e^*(t)$ and $\dot{g}_e^*(t)$
2. Then, for waypoint times $t_0, t_1, \ldots, t_n$, we solve the inverse kinematics problem to get $\theta^*(t_k)$.
3. To obtain $\dot{\theta}^*(t_k)$, we can use the psuedo-inverse of the body manipulator Jacobian:

   $$\dot{\theta}^*(t_k) = (J^b(\theta^*(t_k)))^\dagger \xi_e^b(t_k)$$

   with the twist calculated using:

   $$\xi_e^b(t_k) = (g_e^*(t_k))^{-1}\dot{g}_e^*(t_k)$$

Notes on Inverting the Manipulator Jacobian

As we saw, the general form thats used with the inverse manipulator Jacobian is:

$$\dot{\theta} = (\text{inverse of } J^b(\theta))\xi^b$$

When J^b is not square, we use the **Moore-Penrose Pseudo-Inverse** to obtain the pseudo-inverse. This inverse is defined as:

1. **Kin. Redundant:** If $J^b \in \mathbb{R}^{n\times m}$ is "fat" ($n < m$) and is full (row) rank, then:

$$J^\dagger = J^T(JJ^T)^{-1} \qquad \text{(Right Inverse)}$$

2. **Kin. Insufficient:** If $J^b \in \mathbb{R}^{n\times m}$ is "tall" ($n > m$) and has full (column) rank, then:

$$J^\dagger = (J^TJ)^{-1}J^T \qquad \text{(Left Inverse)}$$

As noted in a previous lecture, the Left Inverse case should be avoided since kinematically insufficient manipulators unless one is very careful about keeping ξ in the range space of $J/J^b/J^s$.

Multiple Waypoints with Desired Velocities

Next, we will further explore how to systematically determine the polynomials that connect multiple waypoints with matching velocity. In this setting, we will assume that we are given the following desired features:

- $p_0, p_1, \ldots, p_n, p_{n+1}$ (initial, intermediate, and final positions)
- $0, \dot{p}_1, \ldots, \dot{p}_n, 0$ (initial, intermediate, and final velocities)
- $0, t_1, \ldots, t_n, t_f$ time points
- $T_0, T_1, \cdots, T_{n-1}, T_n$ (durations of each segment, with $T_k = t_{k+1} - t_k$ and $T_n = t_f - t_n$)

Using a similar approach as before, we can construct $n + 1$ polynomials, each of which has the form:

$$p^k(\tau) = a_0^k + a_1^k\tau + a_2^k\tau^2 + a_3^k\tau^3.$$

with the input τ being defined on the range for that waypoint segment $\tau = [0, T_k]$.

We can then represent the complete polynomial as:

$$p(t) = p^k(t - t_k) \text{ where } k = \text{floor}(t/T)$$

assuming that $T_0 = T_1 = \cdots = T_n = T$. If any T_k are different, then we would instead need to find k such that $t \in [t_k, t_{k+1}]$.

To actually solve for the coefficients of each polynomial, we can set up a large system of equations with matching velocity constraints:

$$\begin{aligned}
p^0(0) &= a_0^0 && = p_0 \\
p^0(T_0) &= a_0^0 + a_1^0 T_0 + a_2^0 T_0^2 + a_3^0 T_0^3 && = p_1 \\
\dot{p}^0(0) &= a_1^0 && = 0 (= \dot{p}_0) \\
\dot{p}^0(T_0) &= a_1^0 + 2a_2^0 T_0 + 3a_3^0 T_0^2 && = \dot{p}_1 \\
&\vdots && \vdots \\
p^k(0) &= a_0^k && = p_k \\
p^k(T_k) &= a_0^k + a_1^k T_k + a_2^k T_k^2 + a_3^k T_k^3 && = p_{k+1} \\
\dot{p}^k(0) &= a_1^k && = \dot{p}_k \\
\dot{p}^k(T_k) &= a_1^k + 2a_2^k T_k + 3a_3^k T_k^2 && = \dot{p}_{k+1} \\
&\vdots && \vdots \\
p^n(0) &= a_0^n && = p_n \\
p^n(T_n) &= a_0^n + a_1^n T_n + a_2^n T_n^2 + a_3^n T_n^3 && = p_{n+1} \\
\dot{p}^n(0) &= a_1^n && = \dot{p}_n \\
\dot{p}^n(T_n) &= a_1^n + 2a_2^n T_n + 3a_3^n T_n^2 && = 0 (= \dot{p}_{n+1})
\end{aligned}$$

Solving for the coefficients of each polynomial is decoupled, so we can use the same solution as introduced for the "individual" polynomial case. In code, we would use a for loop to loop through each polynomial segment. Below are two different MATLAB examples. The first shows how we oculd code up what we've presented so far. The second shows how you can use MATLAB built-in functions to create and evaluate splines.

However! Connecting in this way only matches velocities. It does not match accelerations yet. If we wanted to also constrain matching accelerations, we would have to add constraints on $\ddot{p}^k(T_k) - \ddot{p}^{k+1}(0) = 0$.

Example MATLAB Code

```
%% Method 1 - Define your polynomial
pvec = [0, 2, 0, 2, 0];
pdotvec = [0, 0, 0, 0, 0];
tvec = [0, 1, 2, 3, 4];

% Define matrix P
P = @(tf) [1 0 0 0; ...
           0 0 1 0; ...
           -3/tf^2 3/tf^2 -2/tf -1/tf; ...
           2/tf^3 -2/tf^3 1/tf^2 1/tf^2];

amat = zeros(4,length(pvec)-1);
for i = 1:length(pvec)-1
    pi = pvec(i);
    pf = pvec(i+1);
    vi = pdotvec(i);
    vf = pdotvec(i+1);
    p0 = [pi; pf; vi; vf];

    tf = tvec(i+1) - tvec(i);
    Pcur = P(tf);

    a0 = Pcur*p0;
    amat(i,:) = a0';

end

% Convert the coefficients into the order that polyval wants
amat = fliplr(amat);

% Evaluate polynomial using
figure(1); clf; hold on;

for i = 1:length(pvec)-1
    % Have to flip the order of coefficients based on how polyval is
    % defined:
    fplot(@(t) polyval(amat(i,:), t-tvec(i)),[tvec(i),tvec(i+1)])
end
```

Example MATLAB Code

```
%% Alternative method using csape (cubic spline
% Define the x-values and corresponding y-values (positions)
pvec = [0, 2, 0, 2, 0];
tvec = [0, 1, 2, 3, 4];

% Define velocity constraints at the endpoints
% Velocity at x = 0
v_start = 0;
% Velocity at x = 4
v_end = 0;

% Create a cubic spline with velocity constraints at the endpoints
pp = csape(tvec, [v_start, pvec, v_end], 'clamped');

% Plot the spline
xx = linspace(min(tvec), max(tvec), 100);
yy = ppval(pp, xx);  % Evaluate the spline at these points
figure(2)
plot(tvec, pvec, 'o', xx, yy, '-');
xlabel('p');
ylabel('t');
title('Cubic Spline with Velocity Constraints');
```

Intro to Resolved Rate Trajectory Generation

So far, we have discussed how to connect initial and desired end-effector configurations with a trajectory using splines and inverse kinematics.

Problems with this method include:

1. multiple solutions to inverse kinematics can cause problems
2. kinematically redundant arms do not have unique/finite inverses

To avoid these problems, we can leverage an alternative technique called **resolved rate trajectory generation**. This method is based on the idea that we can directly control the end-effector velocity and acceleration.

We will detail this method in the next lecture, but in general, the idea is to:

- Specify a desired end-effector trajectory $g_e^*(t)$ and its associated velocity $\dot{g}_e^*(t)$
- Integrate $\dot{\theta}(t) = J^\dagger \xi_e^b(t)$ to get:

$$\theta(t_{k+1}) = \theta(t_k) + \Delta t J^b(\theta(t_k))\xi_e^b(t_k)$$