Topics Covered:

- Joint vs. Group Space Planning
- Straight-Line Paths
- 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

Discussion of Joint Space vs. Group (Task) Space

Below is an example from Lynch/Park Section 9.2 that illustrates the difference between planning a straight-line path in joint-space versus in the group (or they call it task) space:

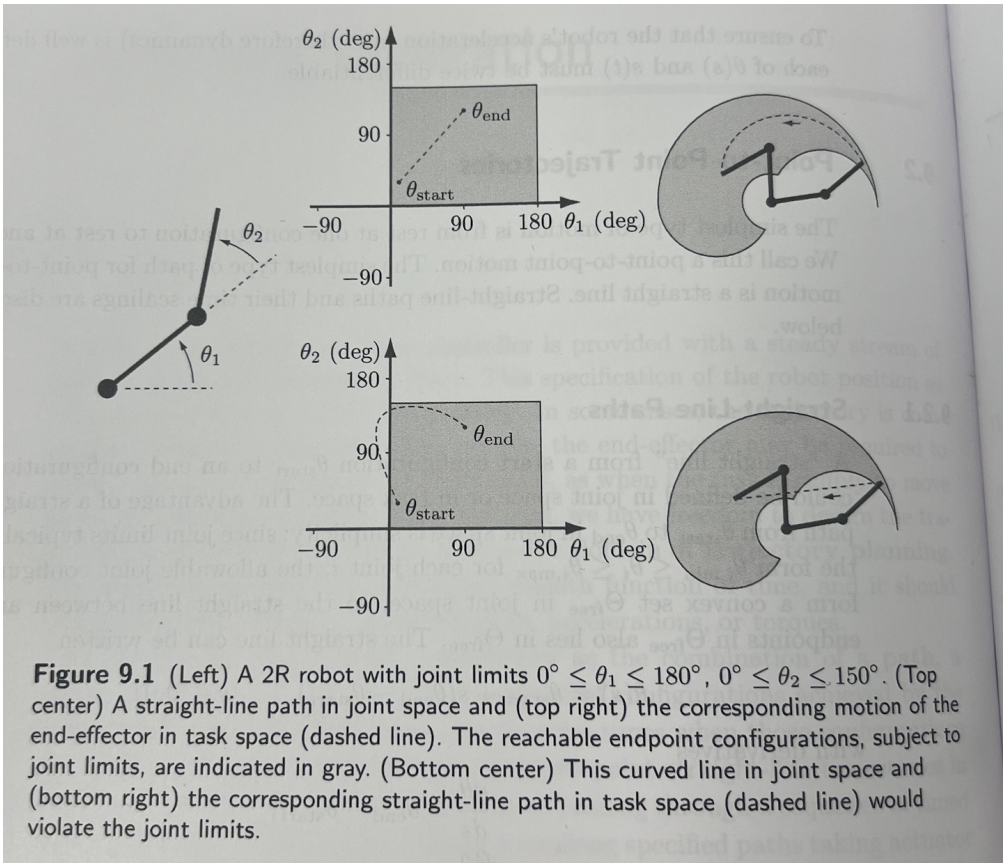


Figure 9.1 (Left) A 2R robot with joint limits $0° \leq \theta_1 \leq 180°$, $0° \leq \theta_2 \leq 150°$. (Top center) A straight-line path in joint space and (top right) the corresponding motion of the end-effector in task space (dashed line). The reachable endpoint configurations, subject to joint limits, are indicated in gray. (Bottom center) This curved line in joint space and (bottom right) the corresponding straight-line path in task space (dashed line) would violate the joint limits.

	Pros	**Cons**
Joint Space Planning	Simpler to compute Joint limits are easier to obey	May result in complex end-effector paths that are hard to predict
Group Space Planning	Ensures end-effector follows desired path in Cartesian space	More computationally intensive Requires solving inverse kinematics

Table 1: Comparison of Joint Space vs. Group Space Planning

Group Space Trajectories

Question: Given a trajectory in task space ($g_e^*(t) \in SE(3)$), how can we find the corresponding desired joint angles to send to achieve this path (i.e., $\theta^*(t)$ such that $g_e^*(t) = g_e(\theta^*(t))$)?

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.

Straight-Line Paths

If we want a straight-line path, we will run into the previously mentioned issues with vibrations caused from instantaneous changes in velocity. So, to avoid these vibration issues, we can instead define our polynomial over our time-scaling function $s(t)$ that maps your time interval $[t_i, t_f]$ to $[0, 1]$, i.e., $s : [t_i, t_f] \to [0, 1]$.

$$s(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

with the constraints:

$$s(0) = 0, \quad \dot{s} = 0, \quad s(T) = 1, \quad \dot{s}(T) = 0. \qquad (T = t_f - t_i)$$

Note: time-scaling can also just be used with polynomials over $p(t)$, but here the choice of $s(t)$ could be linear:

$$s(t) = \frac{t - t_i}{t_f - t_i}$$

But, using our polynomial function $s(t)$, a twice-differentiable "straight-line" path can be created as the convex combination between two points p_i and p_f:

$$\begin{aligned} p(s) &= (1 - s)p_i + sp_f \\ &= p_i + s(p_f - p_i) \end{aligned}$$

This procedure still gives us a twice-differentiable path with the coefficients:

$$a_0 = 0, \quad a_1 = 0, \quad a_2 = 3/T^2, \quad a_3 = -2/T^3$$

Using this time-scaling, we can now define our straight-line path as the convex combination of two points p_i and p_f. For straight-line paths in the joint space, this is:

$$\theta(s) = \theta_i + s(\theta_f - \theta_i), \quad s \in [0, 1]$$

with the velocity:

$$\begin{aligned}\dot{\theta}(s) &= \dot{s}(\theta_f - \theta_i) \\ &= (a_1 + 2a_2 t + 3a_3 t^2)(\theta_f - \theta_i) \\ &= \left(\frac{6t}{T^2} - \frac{6t^2}{T^3}\right)(\theta_f - \theta_i)\end{aligned}$$

A straight-line path in group space $g = (R, p) \in (SE(3))$ is a bit more complicated since:

$$g(s) = g_i + s(g_f - g_i)$$

does not generally lie in $SE(3)$. In other words, this convex combination does not necessarily follow the physical laws of motion. Instead, we must use our previous knowledge about tranformations to derive our path:

$$g_{if} = g_{wi}^{-1} g_{wf}$$

with i being the initial frame, f being the final frame, and w being the world (or spatial) frame. We can then use our exponential map to obtain specific configurations along this path. This is done by first noting that the twist associated with this transformation is:

$$\xi = \ln(g_{wi}^{-1} g_{wf}),$$

so taking the exponential map gives us:

$$g(s) = \underbrace{g_{wi}}_{g_i^*} \exp(\ln(g_{wi}^{-1} \underbrace{g_{wf}}_{g_f^*})s).$$

Finally, we can construct $g(s)$ by individually constructing paths for the Cartesian position and the rotation:

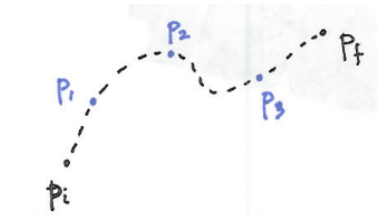
$$\begin{aligned}p(s) &= p_i + s(p_f - p_i) \\ R(s) &= R_i \exp(\ln(R_i^T R_f)s)\end{aligned}$$

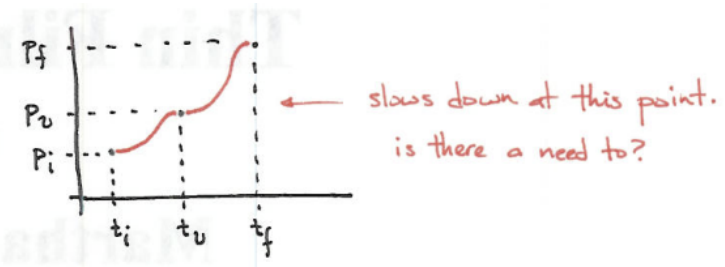
Using this method, the full procedure for generating a straight-line path in group space is:

1. Start with g_i and g_f.
2. Obtain your cartesian path and rotation path $p(s)$ and $R(s)$ from g_i and g_f.
3. Obtain your scaling function $s(t)$ as a cubic polynomial.
4. Conduct inverse-kinematics at either waypoints or continuously along $g(s)$.
5. Interpolate/move along $\theta^*(s)$ and $\dot{\theta}^*(s)$.

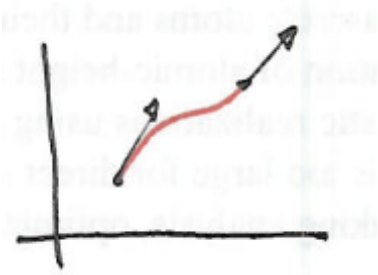
Connecting Waypoints

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.

Solving for Coefficients of Multiple Waypoints

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');
```