Trajectory Optimization for Humanoid Arms

Maren Bennewitz, Rohit Menon
Humanoid Robots Lab, University of Bonn

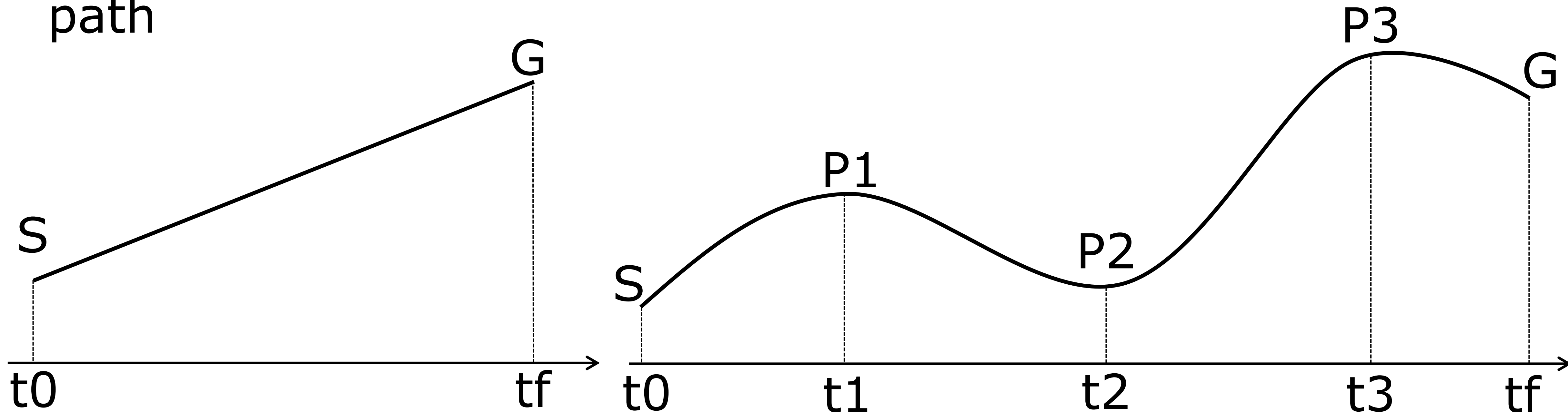
Trajectory Generation

Motivation

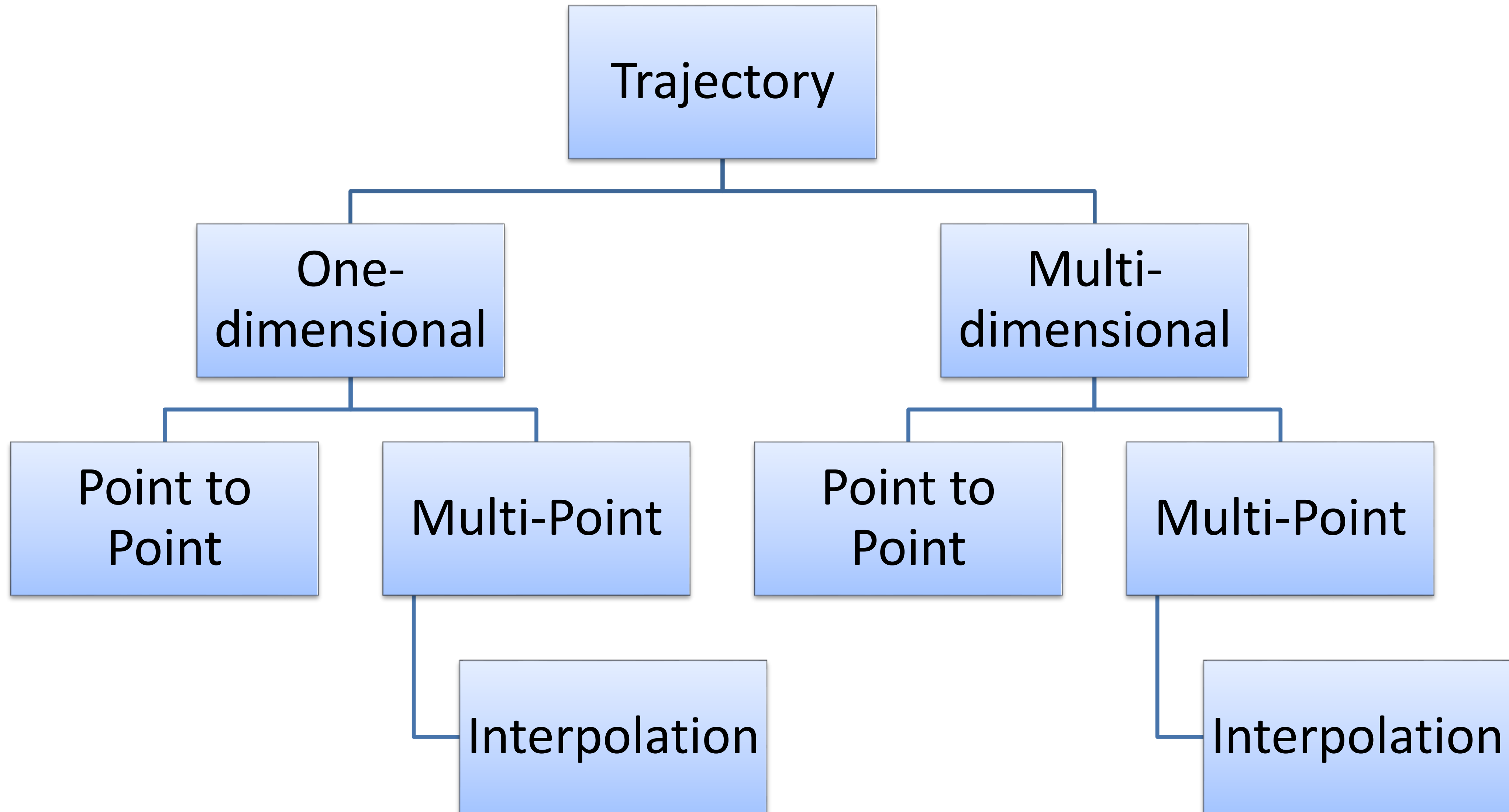
- Sampling based planners typically produce only paths
- It assumes static obstacles and finds a collision free path
- However, robots exist in the spatial and temporal world
- Hence the paths need to be parameterized with time i.e. trajectory
- The optimal trajectory depends on the constraints on the velocities, torques, etc of the robot joints

Trajectory

- Adds time parameterization to path
  - Initial and final times
  - Time optimality
- May specify velocity, acceleration, jerk or torques along path



Main Trajectory Categories



General Objectives of Trajectory Parameterization in Joint Space

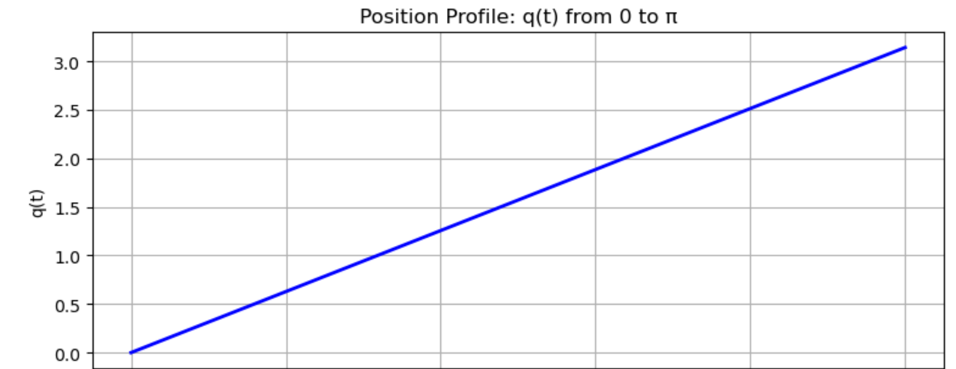
- Joint positions (q), and velocities ($\dot{q}$) should be differentiable
- Joint accelerations ($\ddot{q}$) should be at least continuous, or preferably differentiable for minimum jerk
- However, at the same time, the trajectory should be time-optimal

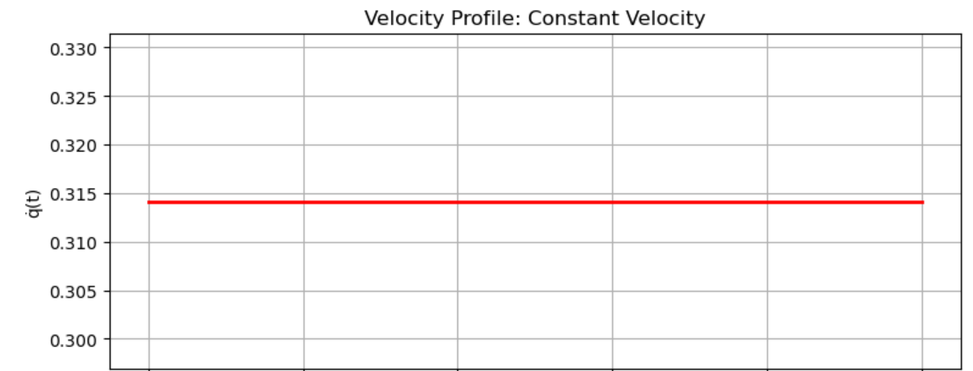
Point to Point Motion

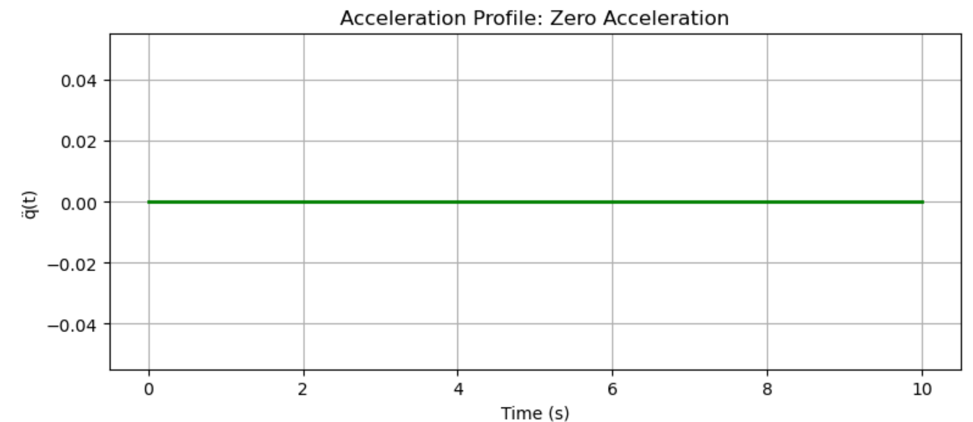
- Define initial and final positions
- No intermediate waypoints are required
- Specify start and end constraints
- Consider acceleration boundaries for smooth motion
- Simplest motion type
- Basis for more complex motion profiles.

Constant Velocity Profile

- Simplest naïve profile
- Linear
- $q(t) = a_0 + a_1(t - t_0)$
- We can specify either time constraints or max velocity

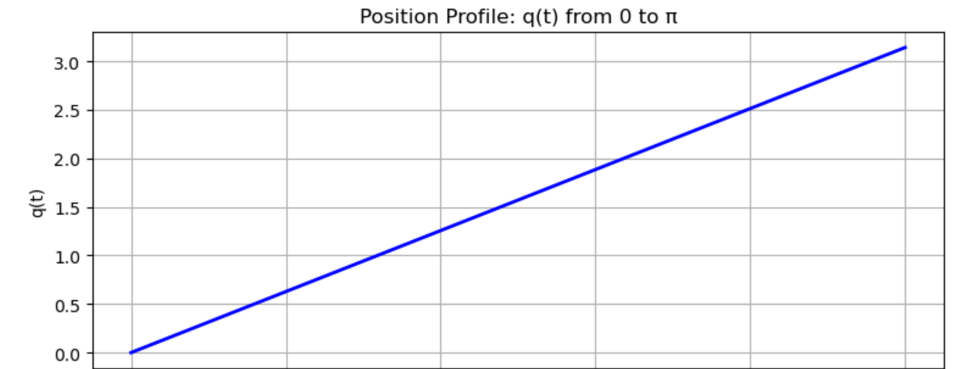


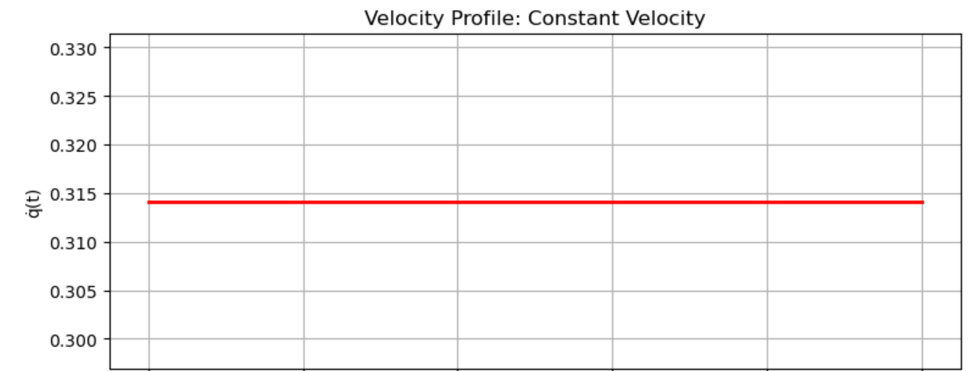


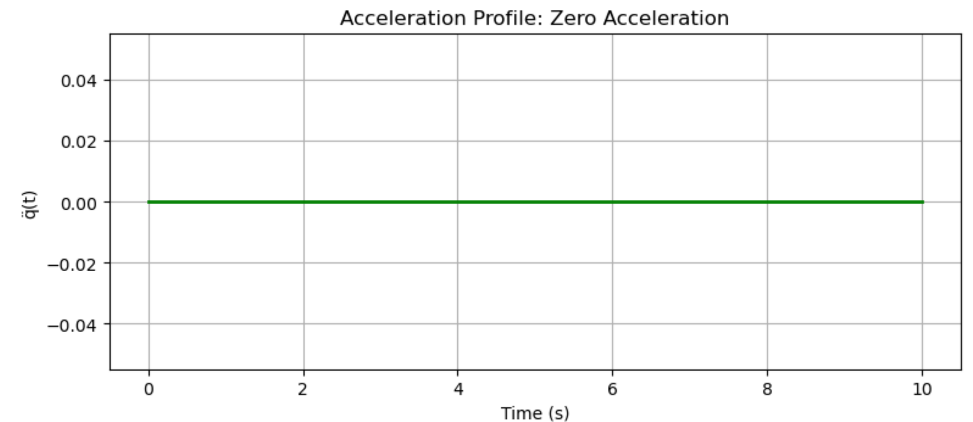


Constant Velocity Profile

- $q(t) = a_0 + a_1(t - t_0)$
- We consider time constraints
- Position q_0 at initial time t_0
- Position q_1 at final time t_0

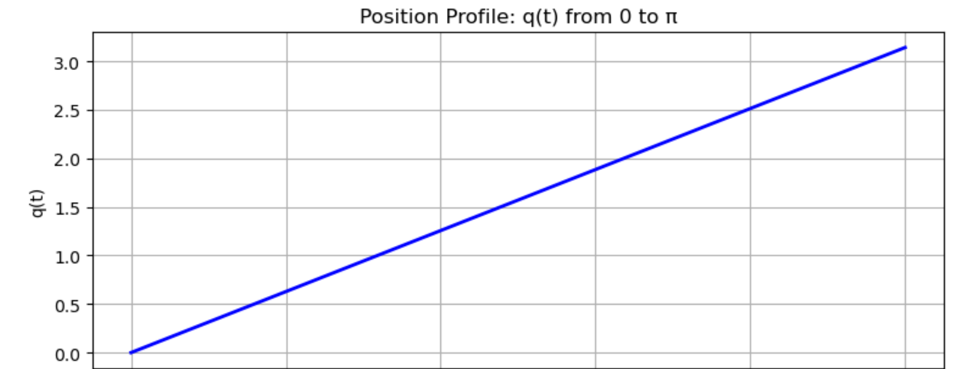


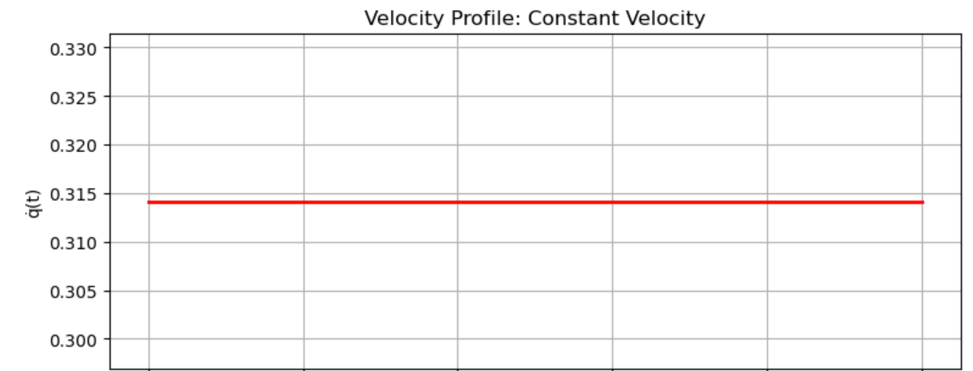


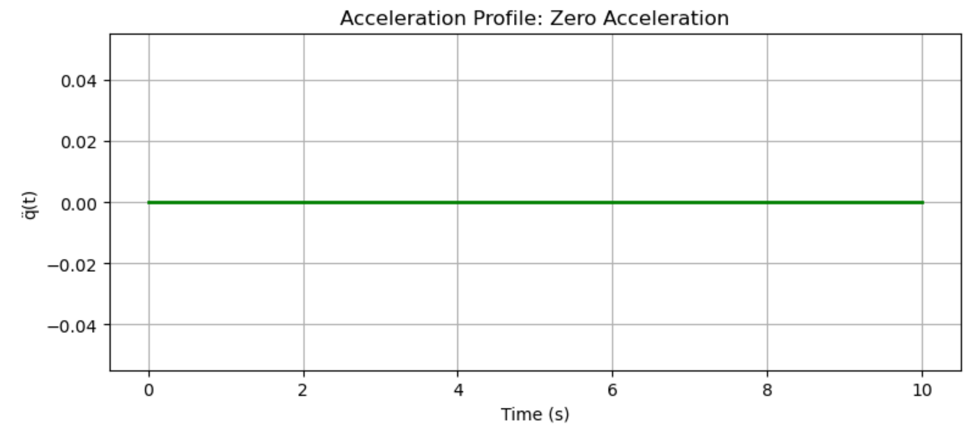


Constant Velocity Profile

- $q(t) = a_0 + a_1(t - t_0)$
- From Position q_0 at initial time t_0
- $q(t_0) = q_0 = a_0$
- $a_0 = q_0$

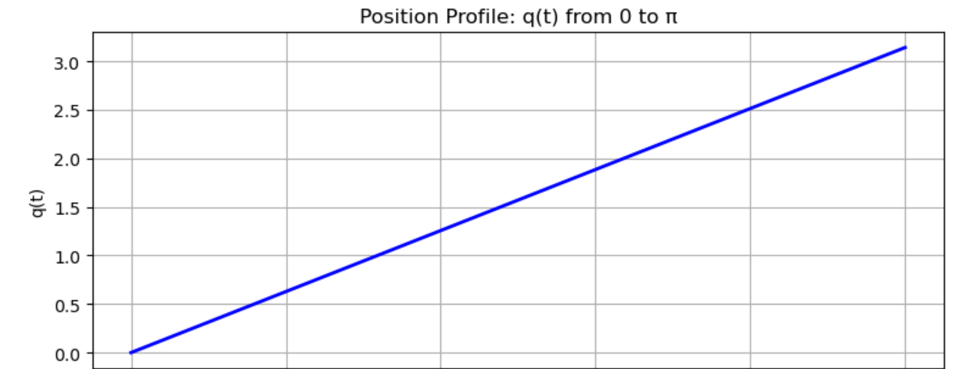


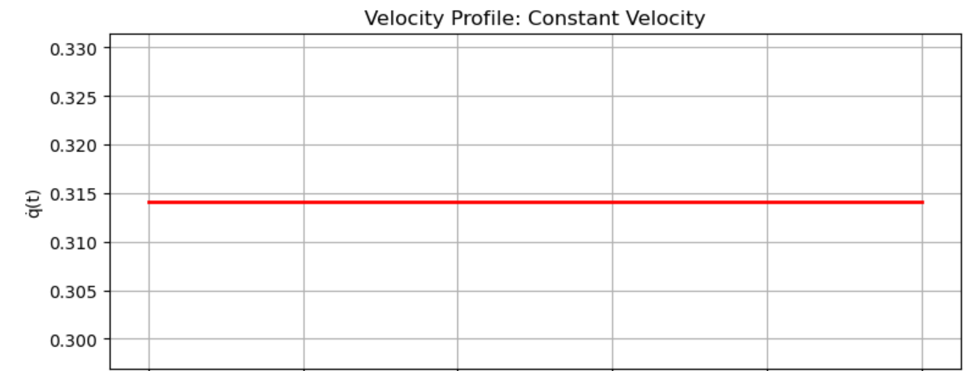


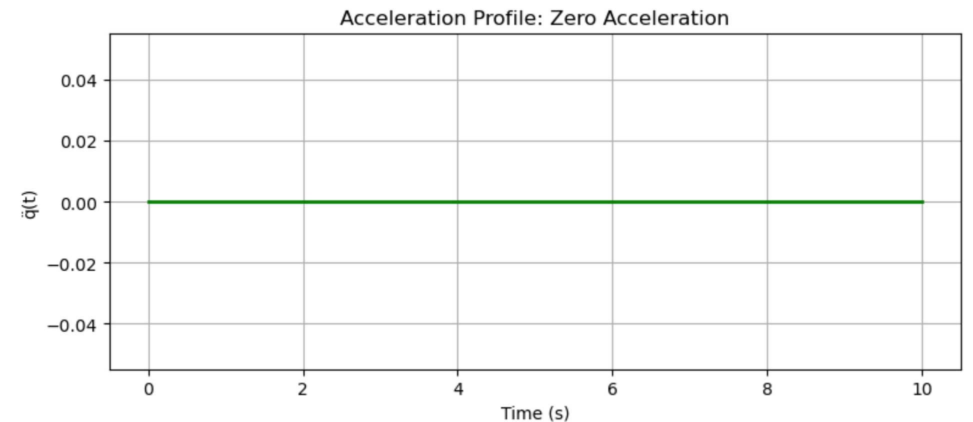


Constant Velocity Profile

- $q(t) = a_0 + a_1(t - t_0)$
- From Position q_1 at initial time t_1
- $q(t_1) = q_1 = a_0 + a_1(t_1 - t_0)$
- $a_1 = \frac{q_1 - q_0}{t_1 - t_0} = \frac{\Delta q}{\Delta t}$
- $\dot{q}(t) = \frac{\Delta q}{\Delta t} = a_1$(Constant Velocity)

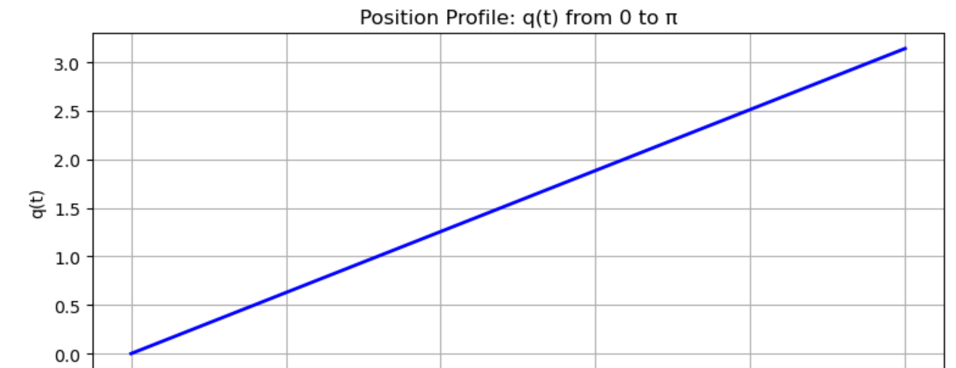


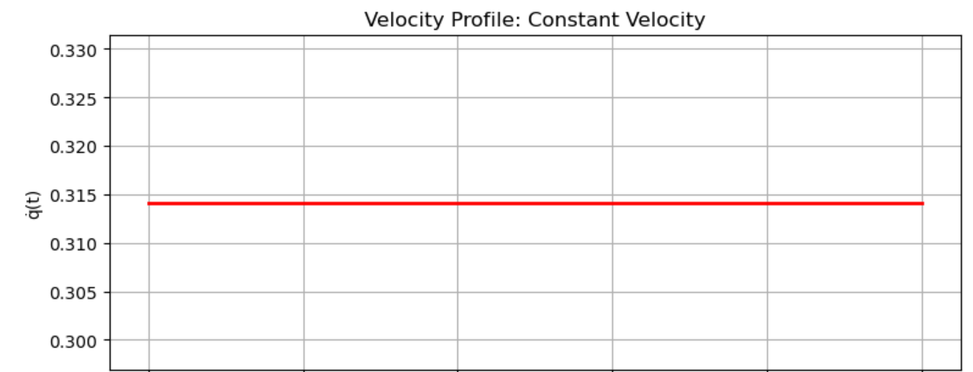


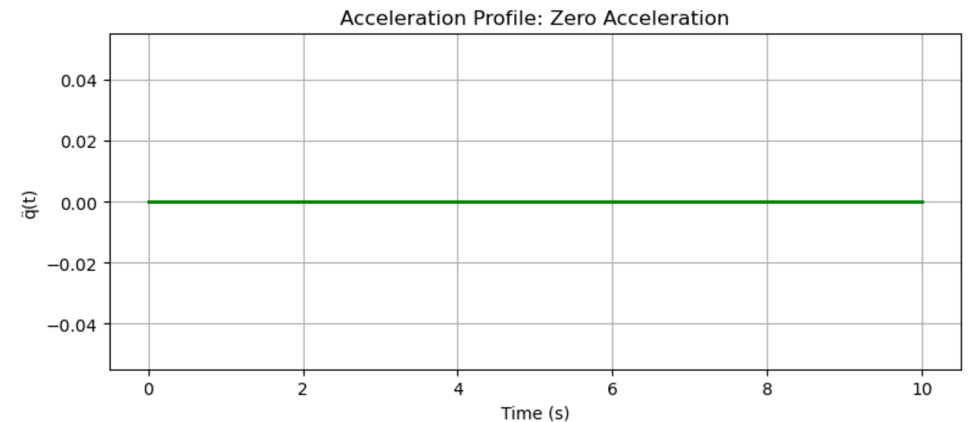


Constant Velocity Profile

- $q(t) = a_0 + a_1(t - t_0)$
- $\dot{q}(t) = \frac{\Delta q}{\Delta t} = a_1$(Constant Velocity)
- Acceleration $\ddot{q}(t) = 0 \; for \; t_0 < t < t_1$
- However, $\ddot{q}(t)$ is undefined for $t = t_0, t = t_1$
- This leads to jerks at the start and end
- Introduces heavy loads on the actuators

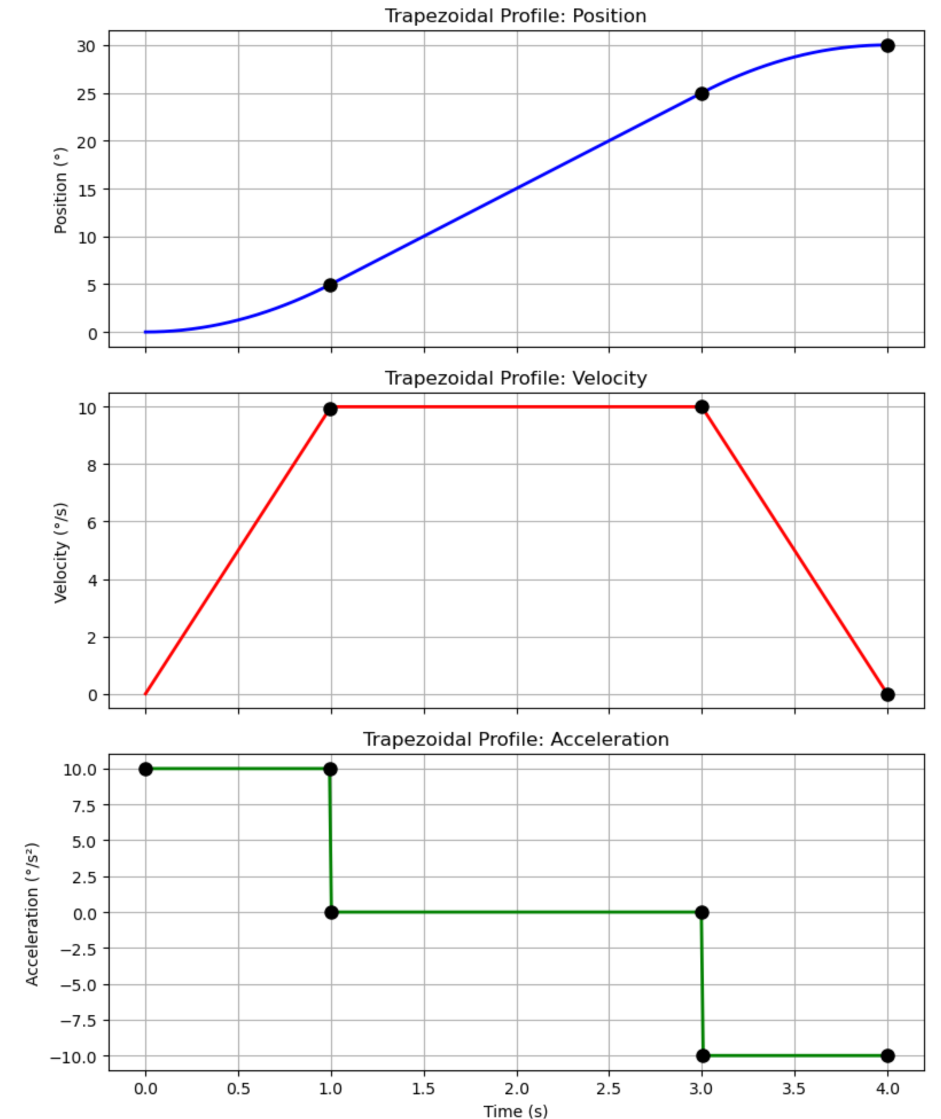






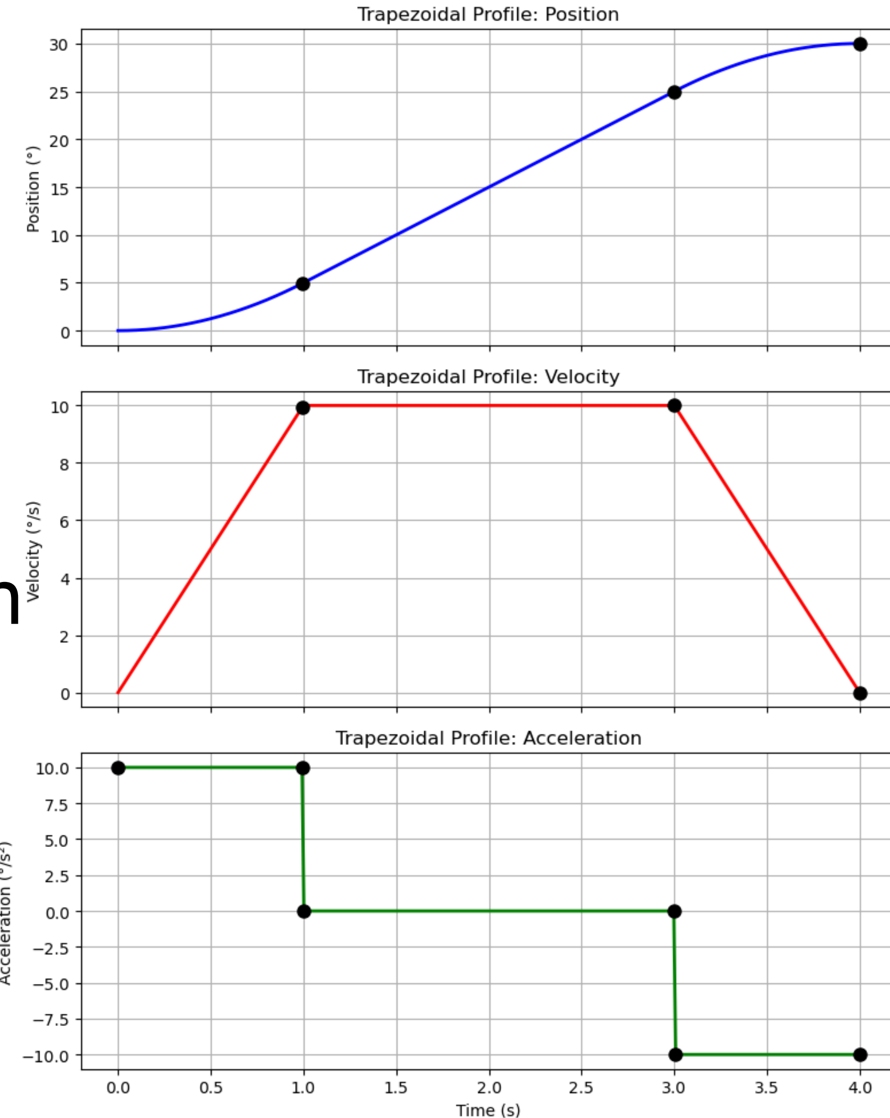
Trapezoidal Profile

- Motion divided into 3 phases
  - Acceleration phase
  - Constant velocity phase
  - Braking phase
- If motion duration too low, then no constant velocity phase and it becomes the earlier case



Trapezoidal Profile - Acceleration

- Acceleration $t \in [0, T_a]$
- $q_a(t) = a_0 + a_1 t + a_2 t^2$
- $\dot{q}_a(t) = a_1 + 2a_2 t$
- $\ddot{q}_a(t) = 2a_2$
- $a_0, a_1, a_2 \rightarrow$ constraints on initial position q_0, velocity v_0 and max velocity v_m
- $a_0 = q_0, a_1 = 0, a_2 = \frac{v_m}{2T_a}$



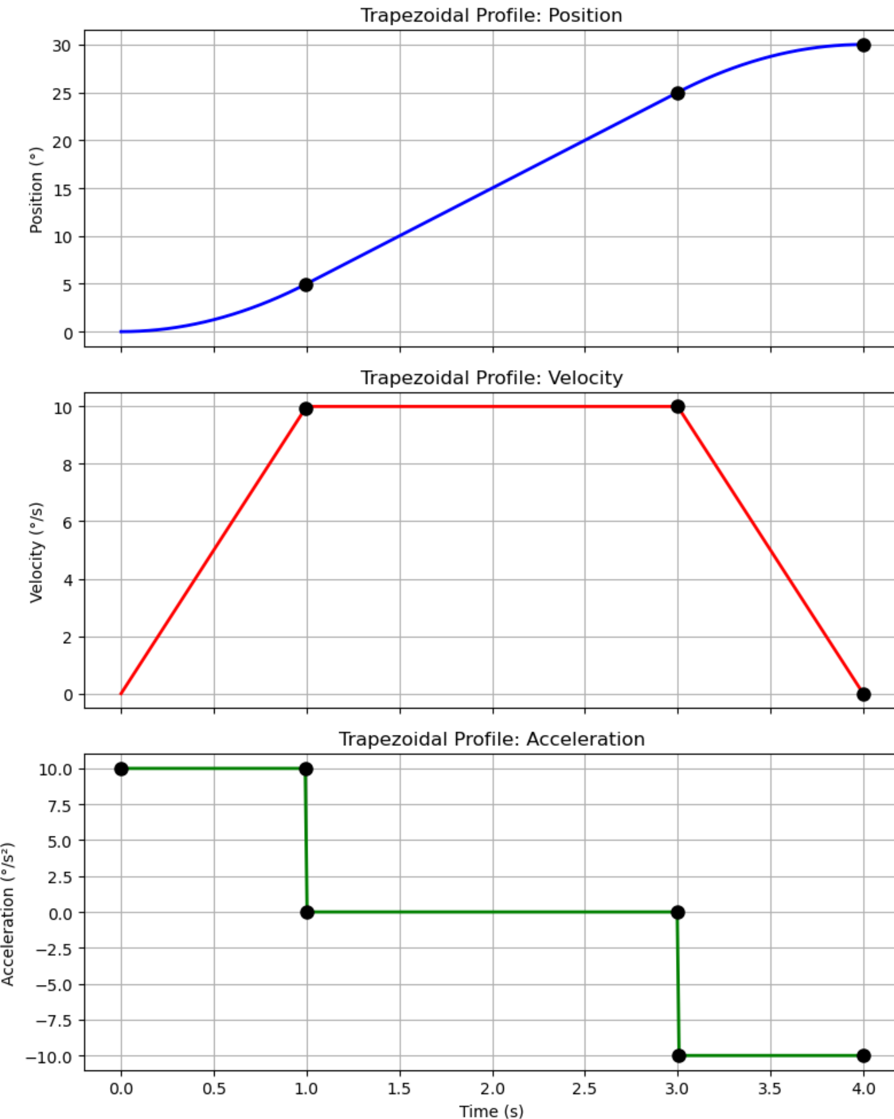
Trapezoidal Profile – Constant Velocity

- Constant velocity $t \in [T_a, t_1 - T_a]$
- $q_c(t) = c_0 + c_1 t$
- $\dot{q}_c(t) = c_1$
- $\ddot{q}_c(t) = 0$
- For velocity to be continuous,

$$\dot{q}_c(T_a) = \dot{q}_a(T_a) \rightarrow c_1 = v_m$$

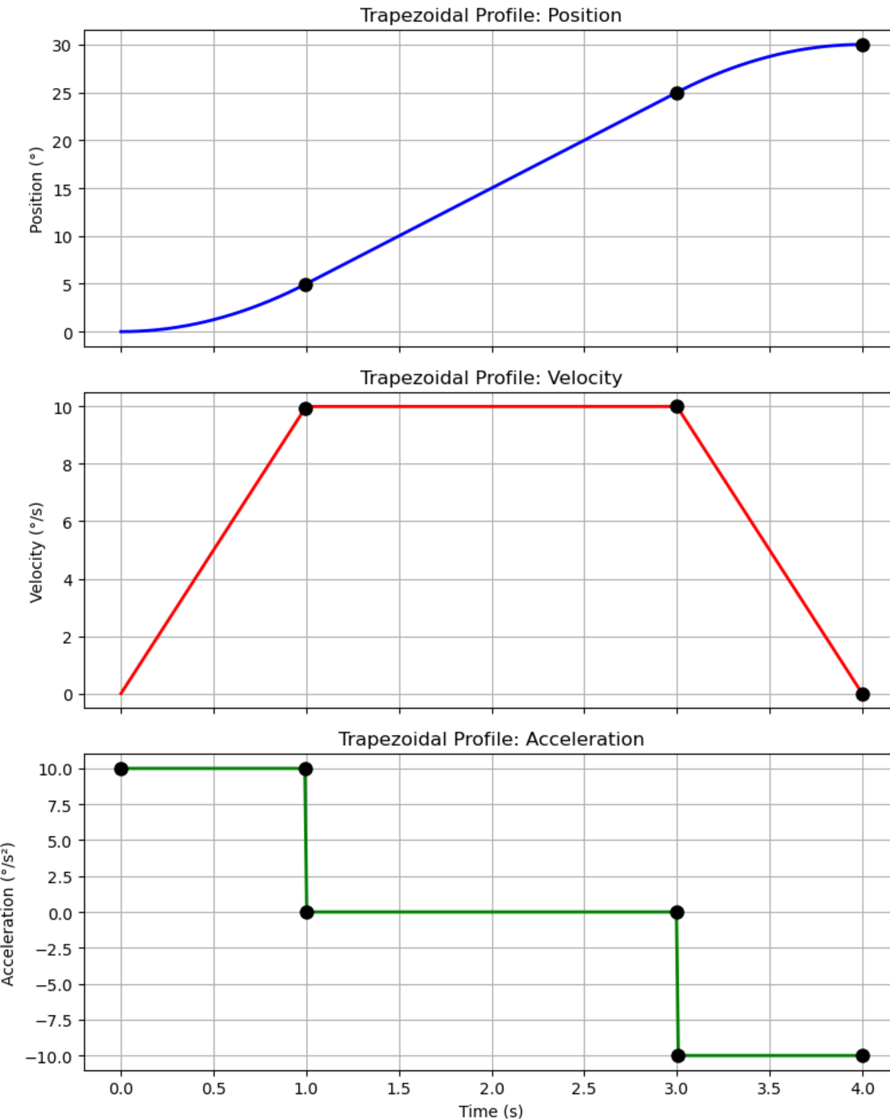
- For position to be continuous

$$q_c(T_a) = q_a(T_a) \rightarrow c_0 = q_0 - \frac{v_m T_a}{2}$$



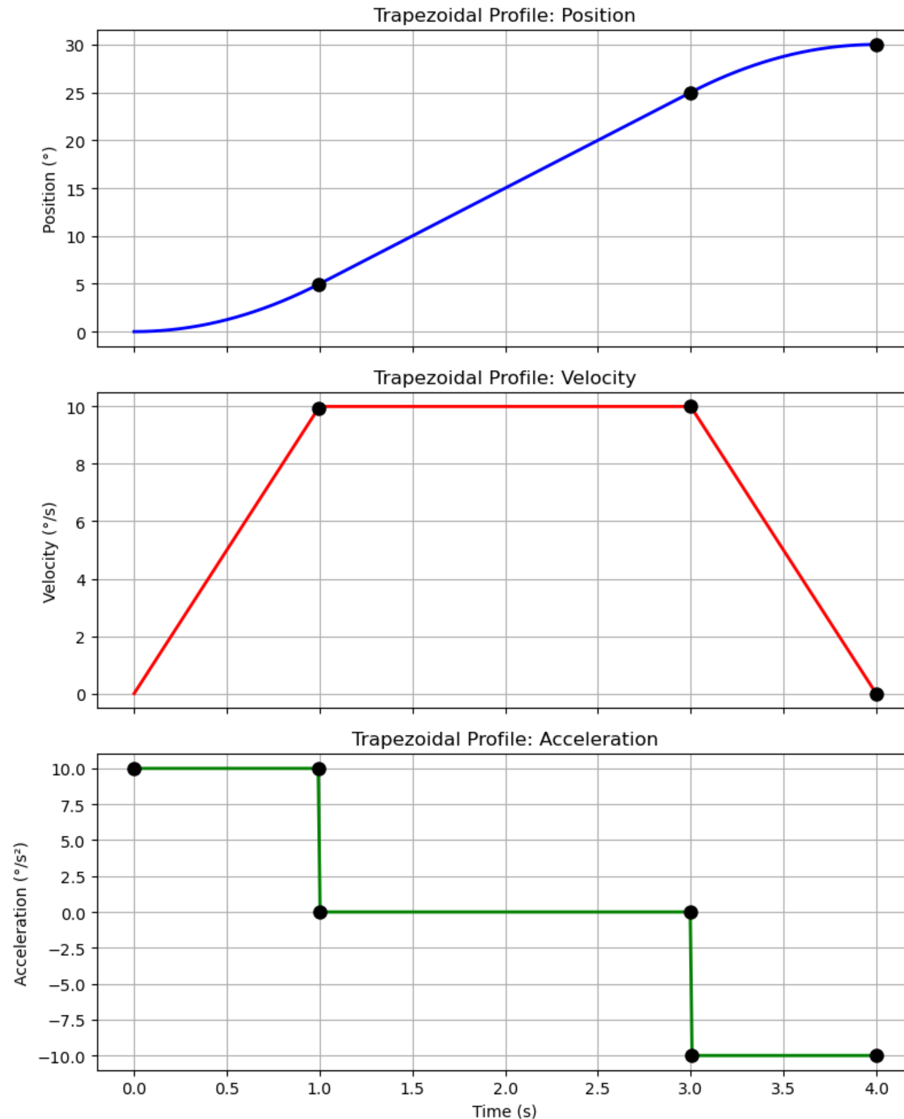
Trapezoidal Profile – Braking

- Braking/deceleration $t \in [t_1 - T_a, t_1]$
- $q_b(t) = b_0 + b_1 t + b_2 t^2$
- $\dot{q}_b(t) = b_1 + 2b_2 t$
- $\ddot{q}_b(t) = 2b_2$
- With $\dot{q}_b(t_1) = 0, \dot{q}_b(t_1 - T_a) = \dot{q}_c(t_1 - T_a)$ & $q_b(t_1 - T_a) = q_c(t_1 - T_a)$
- $b_0 = q_1 - \frac{v_m t_1^2}{2T_a}$, $\mathrm{b}_1 = \frac{v_m t_1}{T_a}$, $\mathrm{b}_2 = -\frac{v_m}{2T_a}$



Trapezoidal Profile

- Trapezoidal profile is commonly used in many robotic systems
- The different parameters especially T_a can be determined in different ways
  - Max velocity constraint
  - Max acceleration constraint



Multi-dimensional trajectories

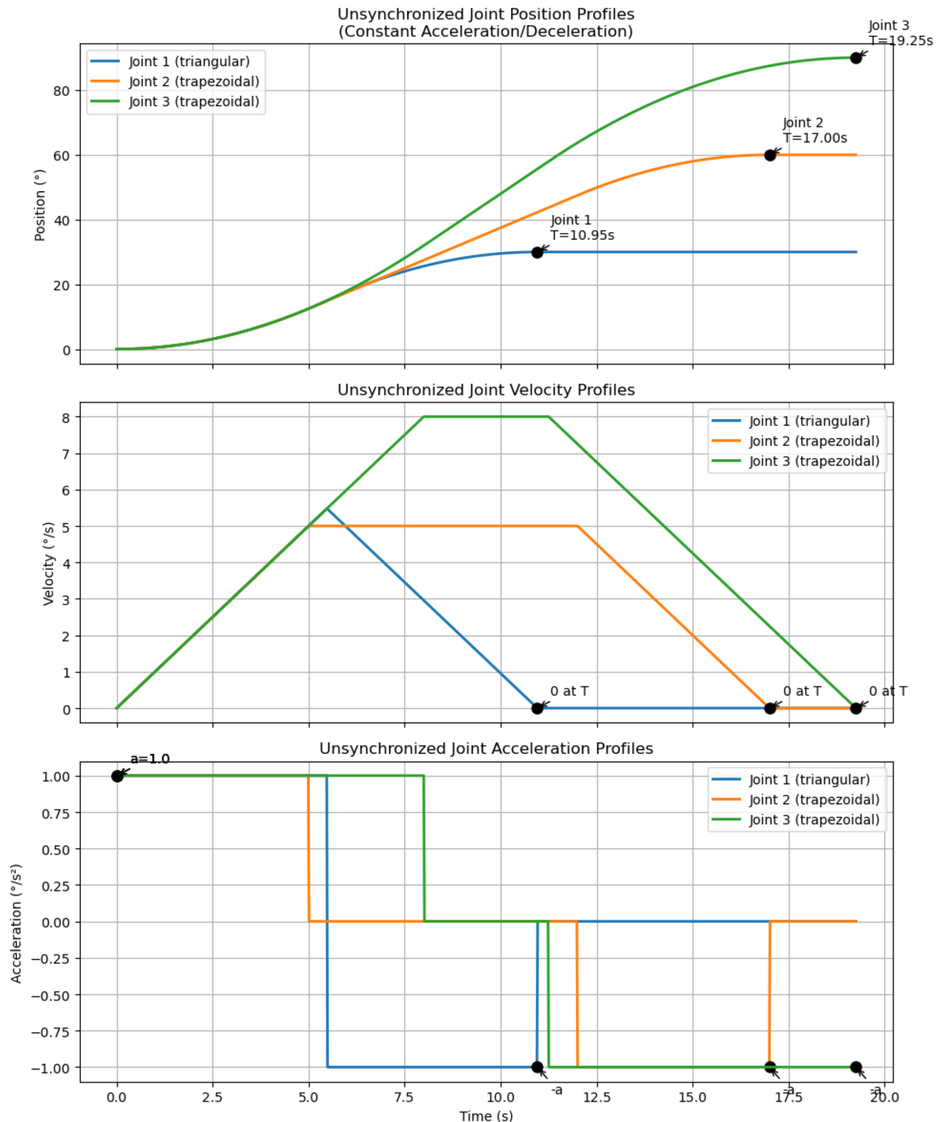
- Robots, especially arms have multiple degrees of freedom
- Typical robot trajectories span high-dimensional coordinated motions
  - e.g. for robot sketching, $(x, y, z, \alpha, \beta, \gamma)$ all have to be coordinated
  - e.g. for collision avoidance, all joint motions have to be coordinated

Time Synchronization of Multi-dimensional Trajectories in Joint Space

- Different joints can have different maximum velocities/accelerations, different displacements
- All joints share same motion duration
- Time synchronization ensures coordinated, simultaneous actions
- Common duration determined by joint with longest duration
- Energy conserved using minimal necessary acceleration and velocity

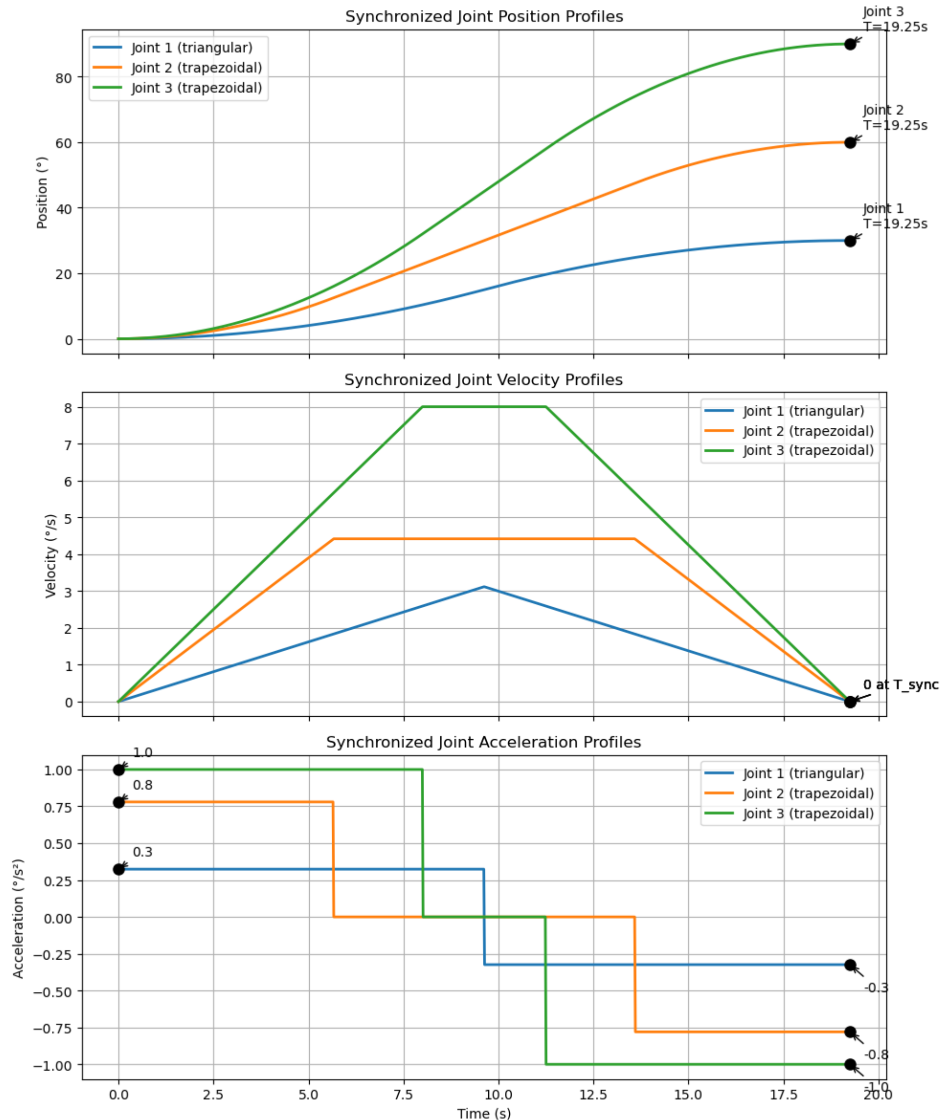
Unsynchronized Trajectories

- Assume robot has 3 joints J1, J2, J3
  - J1: $q1_0 = 0, q1_f = 30°, v1_{max} = 6deg/s$
  - J2: $q2_0 = 0, q2_f = 60°, v2_{max} = 5deg/s$
  - J3: $q3_0 = 0, q3_f = 90°, v3_{max} = 8deg/s$
- With unsynchronized motion, all joints reach max acceleration $a1_{max} = a2_{max} = a3_{max} = 1\text{deg/s}^2$
- J1 reaches final configuration quickest
- J3 reaches slowest



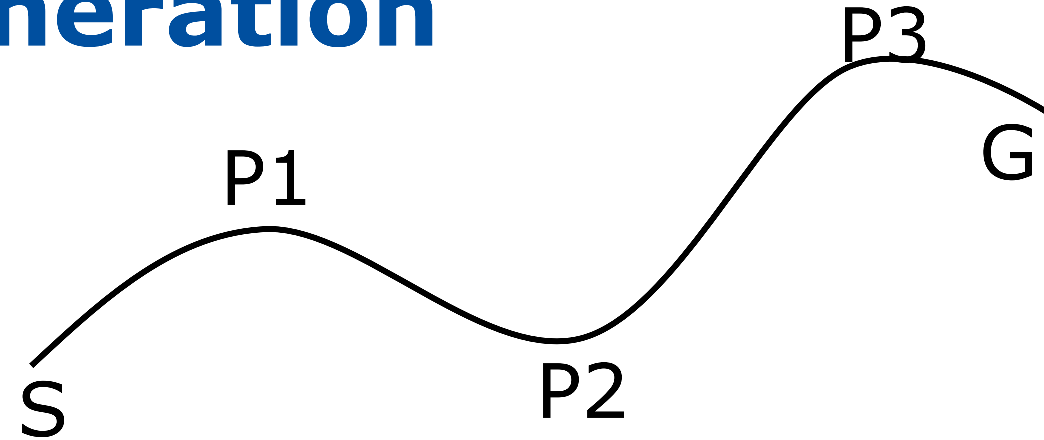
Time Synchronization of Trajectories

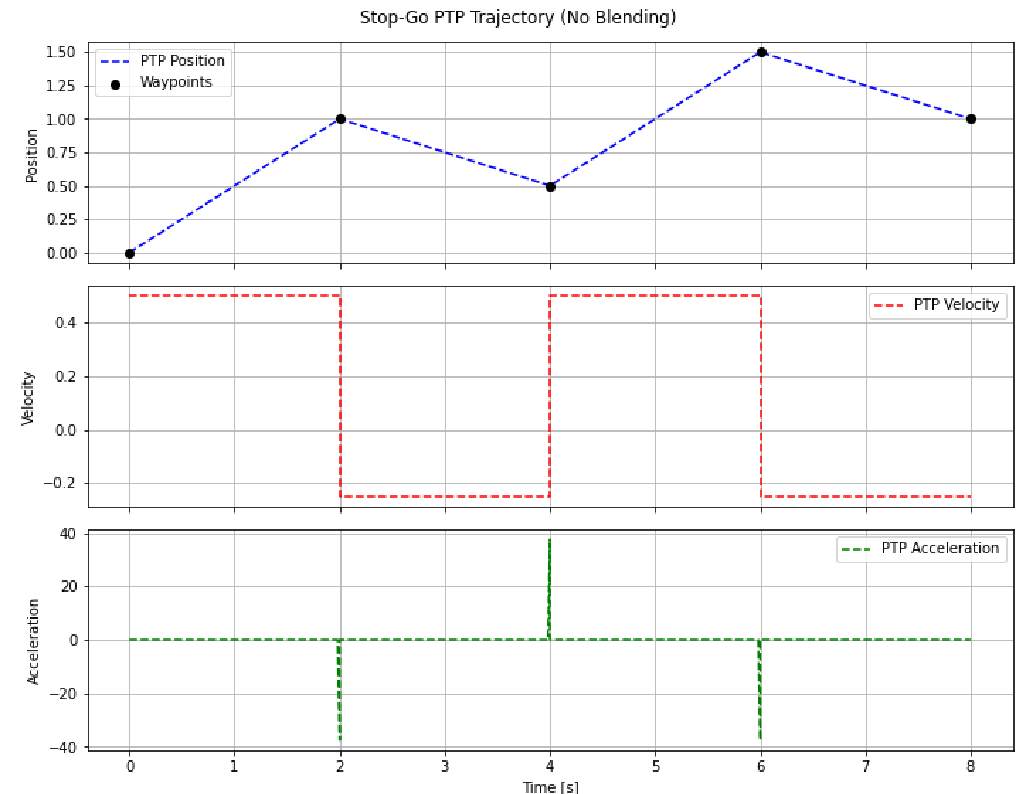
- Compute longest trajectory duration T_{sync}
- This joint has triangular velocity profile (J3)
- Other joints will have all 3 phases
- Calculate stretching factor $\tau_i = t_i / T_{sync}$ where t_i is the unsynchronized time for joint i



Multi-point Trajectory Generation

- Typically motion paths consists of not just S and G, but intermediate waypoints P1, P2, …
- Stopping at each point causes time inefficiency
- Real tasks need smooth, uninterrupted motion
- Discrete stops increase joint wear and energy use
- No need to zero velocity at intermediate points





Multi-point Trajectory Generation

- Use polynomials to smoothly connect waypoints
- Piecewise cubic trajectory between waypoints

$$q_i(t) = a_i(t - t_i)^3 + b_i(t - t_i)^2 + c_i(t - t_i) + d_i \tag{1}$$

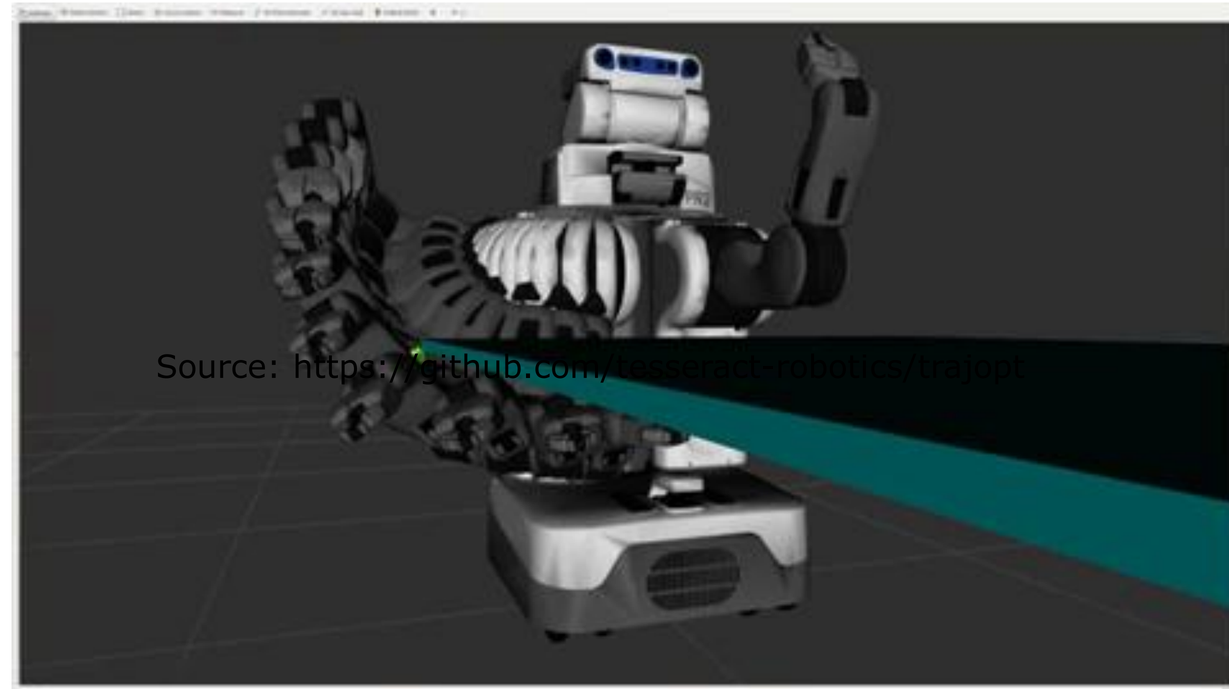
- 4 unknown coefficients per segment: a_i, b_i, c_i, d_i
- Solve for coefficients using boundary conditions
- Match position at each segment boundary

$$q_i(t_i) = q_i, \quad q_i(t_{i+1}) = q_{i+1}$$

Trajectory Optimization

Why Trajectory Optimization

- Optimize robot motion directly
- Encode costs and constraints
- Avoid path-then-smooth separation
- Connect planning, timing and control
- Important for manipulation and also locomotion



Source: https://github.com/tesseract-robotics/trajopt

Sampling Based Motion Planning and Trajectory Generation

- Plan path, then add timing
- Separates geometry from execution dynamics
- Uses geometric smoothing heuristically
- E.g. RRT for path planning + trajectory interpolator

Direct Trajectory Optimization

- Optimizes full motion simultaneously
- Couples planning and control naturally
- Enforces feasibility via mathematical constraints
- E.g. TrajOpt

Fundamentals of Optimization

- Mathematical formulation

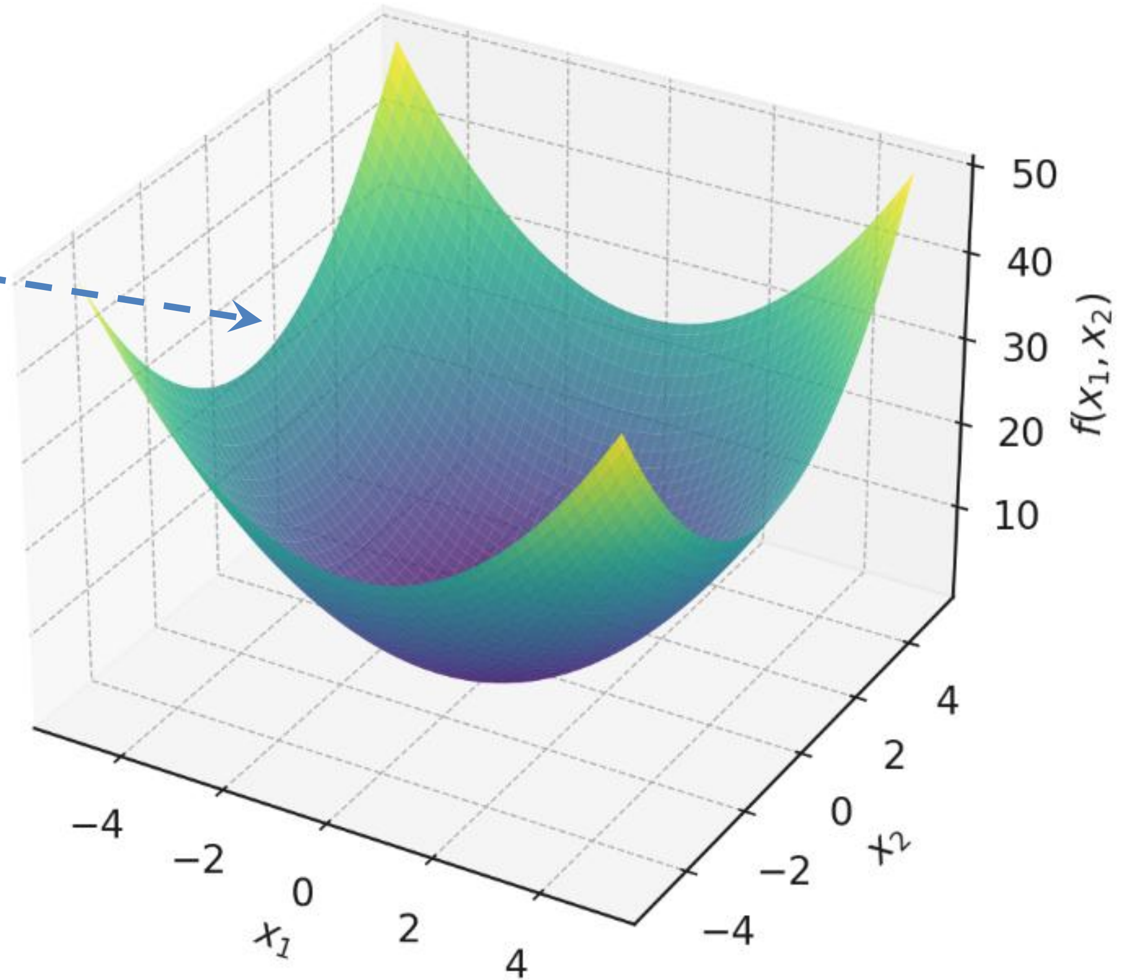
$$
\begin{array}{lll}
\underset{x \in \mathbb{R}^n}{\text{minimize}} & f(x) & \\
\text{subject to} & g(x) & = & 0, \\
 & h(x) & \geq & 0.
\end{array}
$$

- $f(x)$: Cost (objective) function
- $g(x)$: Equality constraint
- $h(x)$: Inequality constraint

Fundamentals of Optimization

- Example

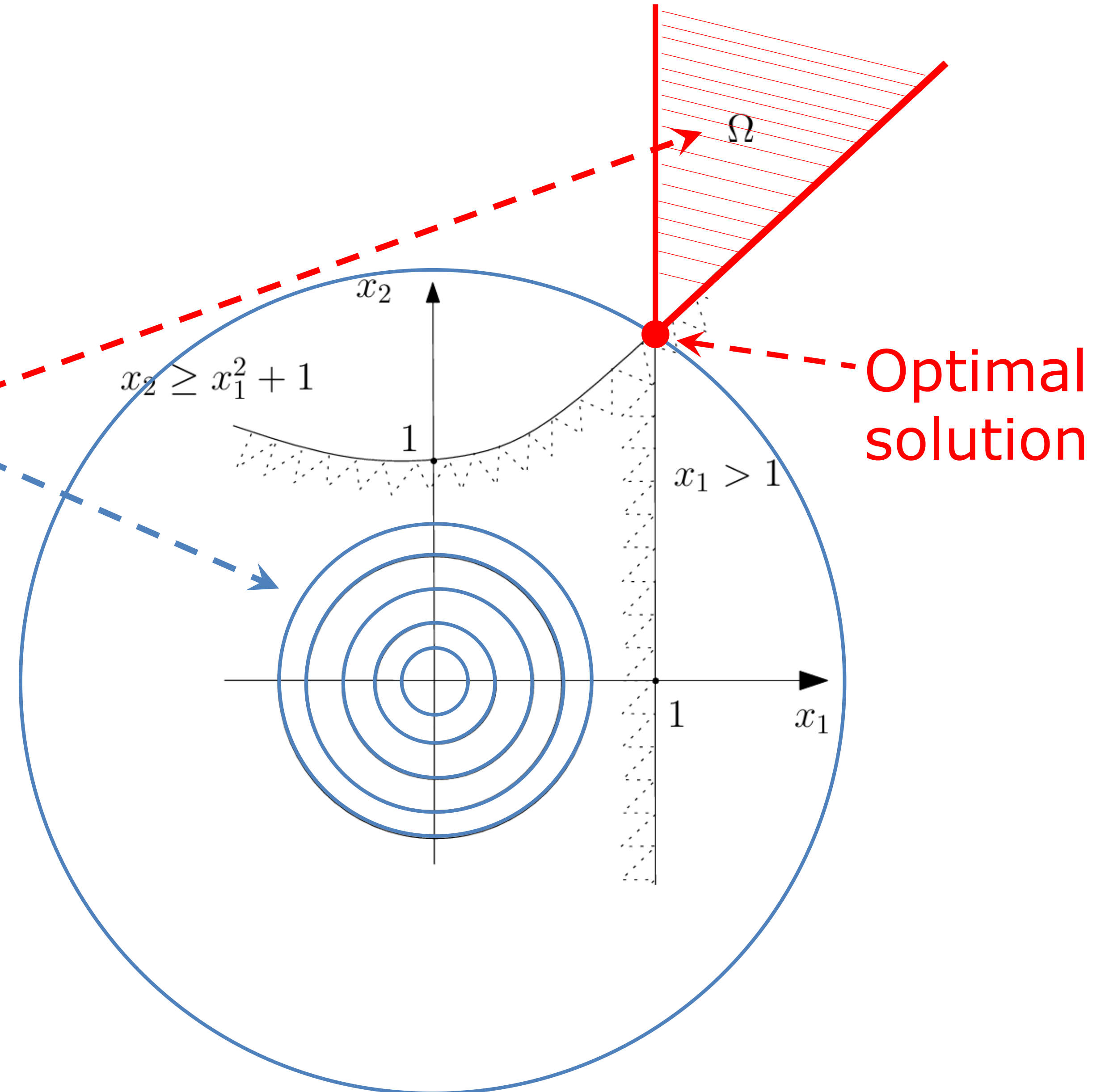
$$\begin{aligned} \underset{x \in \mathbb{R}^2}{\text{minimize}} \quad & x_1^2 + x_2^2 \\ \text{subject to} \quad & x_2 - 1 - x_1^2 \geq 0, \\ & x_1 - 1 \geq 0. \end{aligned}$$



Fundamentals of Optimization

- Example

$$
\begin{aligned}
\underset{x \in \mathbb{R}^2}{\text{minimize}} \quad & x_1^2 + x_2^2 \\
\text{subject to} \quad & x_2 - 1 - x_1^2 \geq 0, \\
& x_1 - 1 \geq 0.
\end{aligned}
$$



- Feasible solutions lie inside the region where constraints are satisfied (red dashed area)

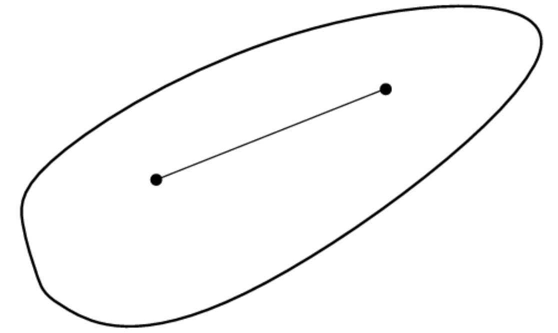
M. Diehl, "Numerical Optimization", Lecture Notes, 2022

Optimization in Robotics

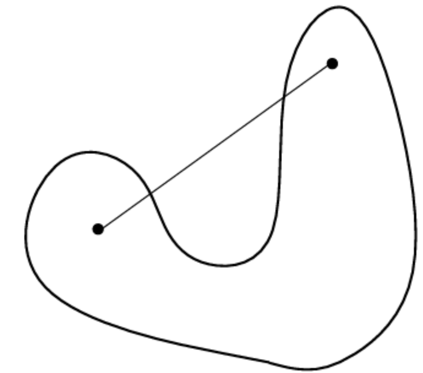
- **Nonlinear:** Objective function $f(x)$ is non-linear in x
  - Robot FK is a non-linear mapping from joint space to SE(3)
- **Constrained:** Imposes explicit mathematical bounds using $g(x)$ and $h(x)$
  - For e.g. joint limits as inequality constraints
- **Unconstrained:** Uses penalties but lacks strict feasibility guarantees → $g(x)$ and $h(x)$ are not present
  - Joint limits→ violations have high penalty but no absolute constraints

Optimization in Robotics

- **Convex set:** Any line connecting two points in the set lies inside the set

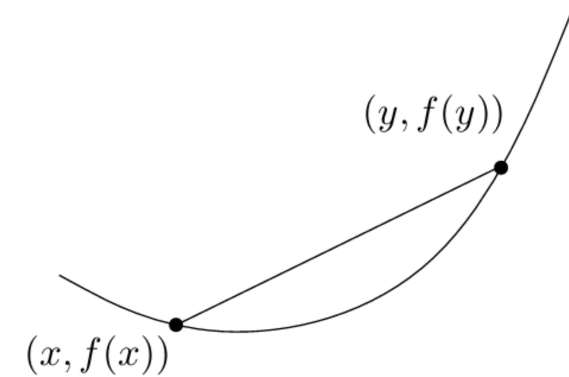


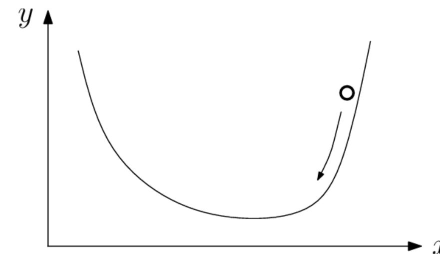
- **Non-Convex:** Obstacles create disconnected configuration spaces

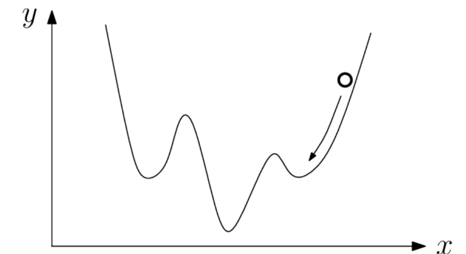


Optimization in Robotics

- **Convex function:** Function defined on a convex set
- Any line segment between two points on the graph of the function lies above or on the graph
- **Convexity:** Every local minimum is globally optimal.







Different Optimization Classes

- **Linear Programming** (LP)
- Linear cost function
- Linear constraints
- Only for local linearized functions

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & Ax - b = 0, \\ & Cx - d \geq 0. \end{aligned}$$

- **Quadratic Programming** (QP)
- Quadratic cost function
- Linear constraints
- Many robotics problems can be formulated as QP

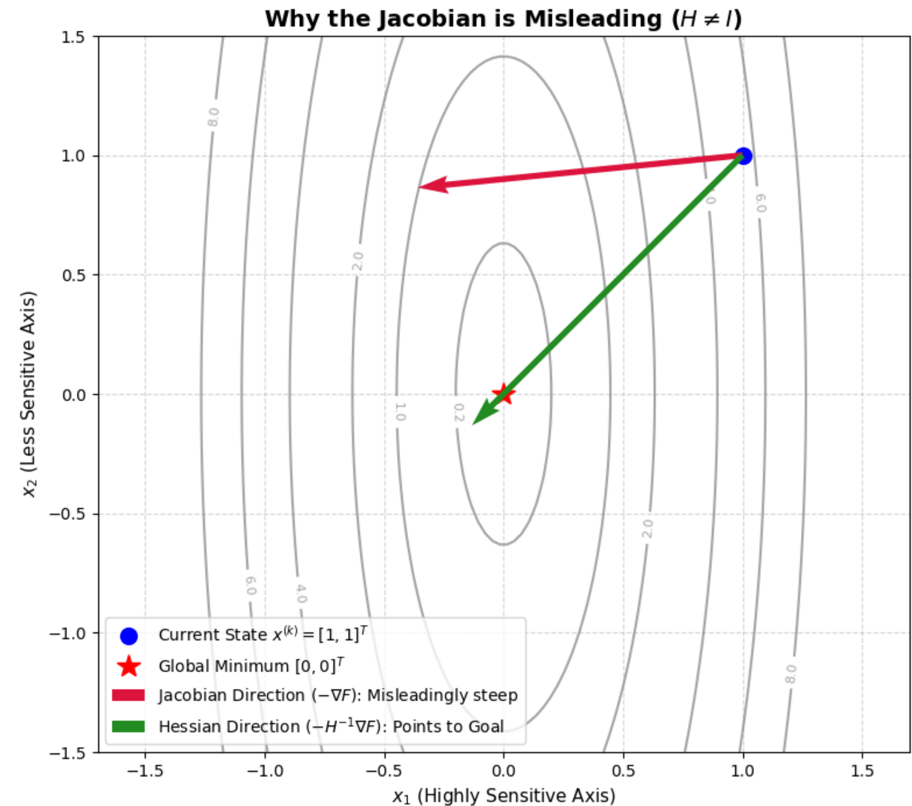
$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^T x + \frac{1}{2} x^T B x \\ \text{subject to} \quad & Ax - b = 0, \\ & Cx - d \geq 0. \end{aligned}$$

Jacobian (Recap)

- **Jacobian:** First order sensitivity
- $\mathrm{J(x)} = \frac{\partial F(x)}{\partial x}$, matrix of first-order derivatives
- Captures multi-dimensional slope
- Relates joint changes to workspace changes
  $\Delta x \approx J(q)\Delta \mathrm{q}$

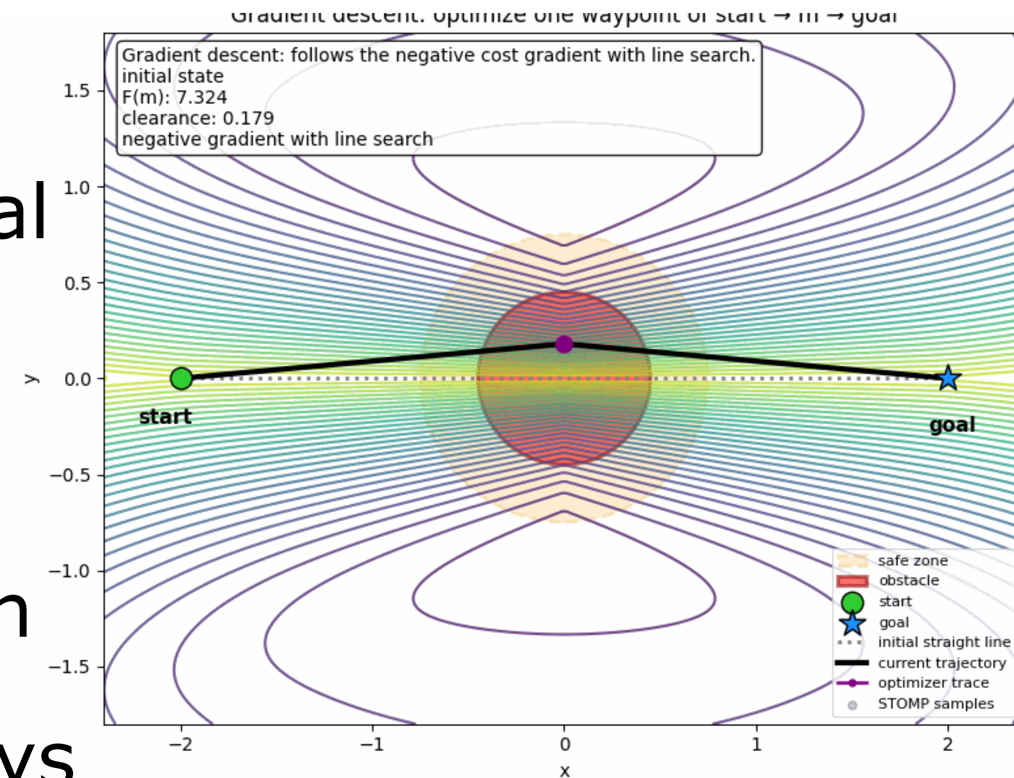
Hessian

- **Hessian:** Second-order curvature
- $\mathrm{H(x)} = \nabla^2 F(x) = \frac{\partial^2 F(x)}{\partial x^2}$, matrix of second-order derivatives
- Captures multi-dimensional curvature
- Indicates how the gradient changes locally
- $\nabla F(x + \Delta x) = \nabla F(x) + H(x)\Delta \mathrm{x}$



Gradient Descent: First Order Update

- Uses only first-order derivative gradients.
- Moves opposite to the steepest local increase
- $\mathbf{x}^{(\mathbf{k+1})} = \mathbf{x}^{(\mathbf{k})} - \boldsymbol{\alpha}\nabla F\left(\mathbf{x}^{(\mathbf{k})}\right)$
- Computationally cheap per iteration
- Convergence slows near local valleys
- Step size α dictates numerical stability.



Newton Method: Second-Order Update

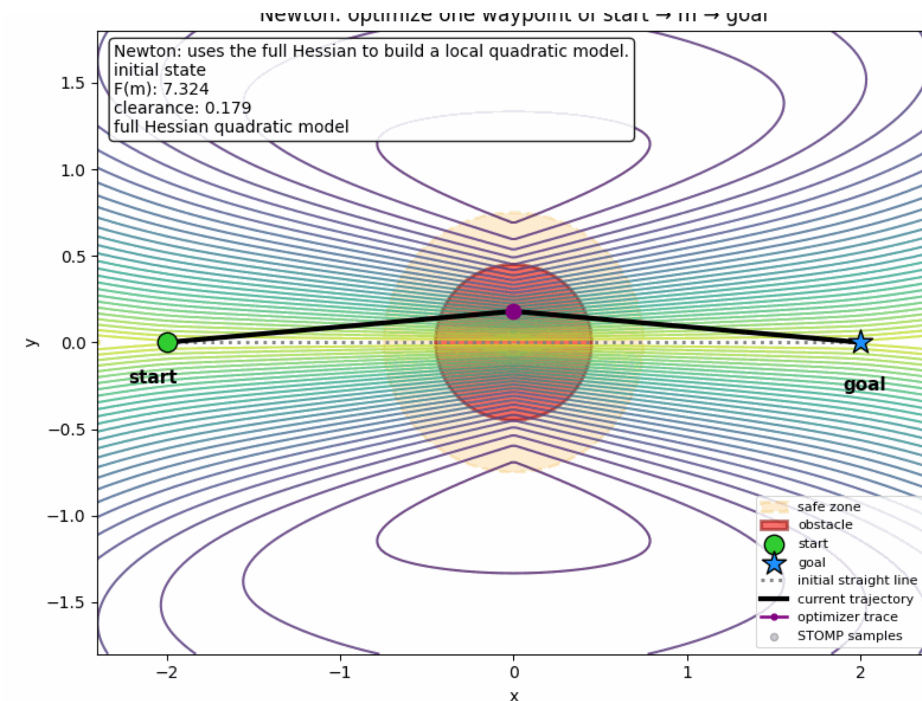
- Uses gradient and Hessian to build a local quadratic model

$$F(x + \Delta x) \approx F(x) + \nabla F(x)^T \Delta x + \frac{1}{2} \Delta x^T H(x) \Delta x$$

- Newton update:

$$H\left(x^{(k)}\right)\Delta x = -\nabla F(x^k),\ \mathbf{x^{k+1}} = \mathbf{x^k} + \mathbf{\Delta x}$$

- **Curvature-aware step: fast near a good solution, but sensitive to poor local models.**

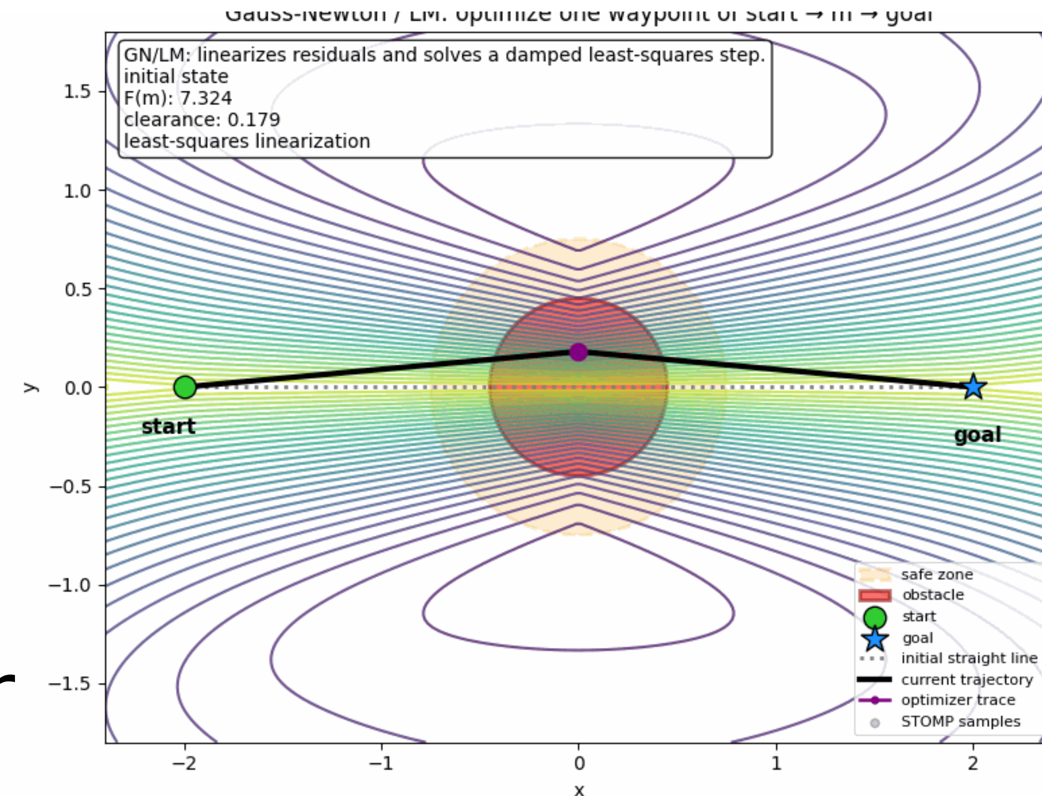


Gauss Newton Method

- **Least-squares problem** $x^* = \arg\min_x \frac{1}{2}\|r(x)\|^2$
- Linearize residual around current iterate $r\left(x^{(k)} + \Delta x\right) \approx r\left(x^{(k)}\right) + J^{(k)}\Delta x$, where $J^{(k)} = \frac{\partial r}{\partial x}|_{x^{(k)}}$
- Solve linearized least-squares problem $\Delta x^* = \arg\min_{\Delta x} \frac{1}{2}\left\|r^{(k)} + J^{(k)}\Delta x\right\|$
- => $J^{T^{(k)}} J^{(k)} \Delta x = -J^{T^{(k)}} r^{(k)}$ where $H^{(k)} = J^{T^{(k)}} J^{(k)}$
- $\mathbf{x^{k+1}} = \mathbf{x^k} + \mathbf{\Delta x}^*$
- Avoids computing complex second-order derivatives

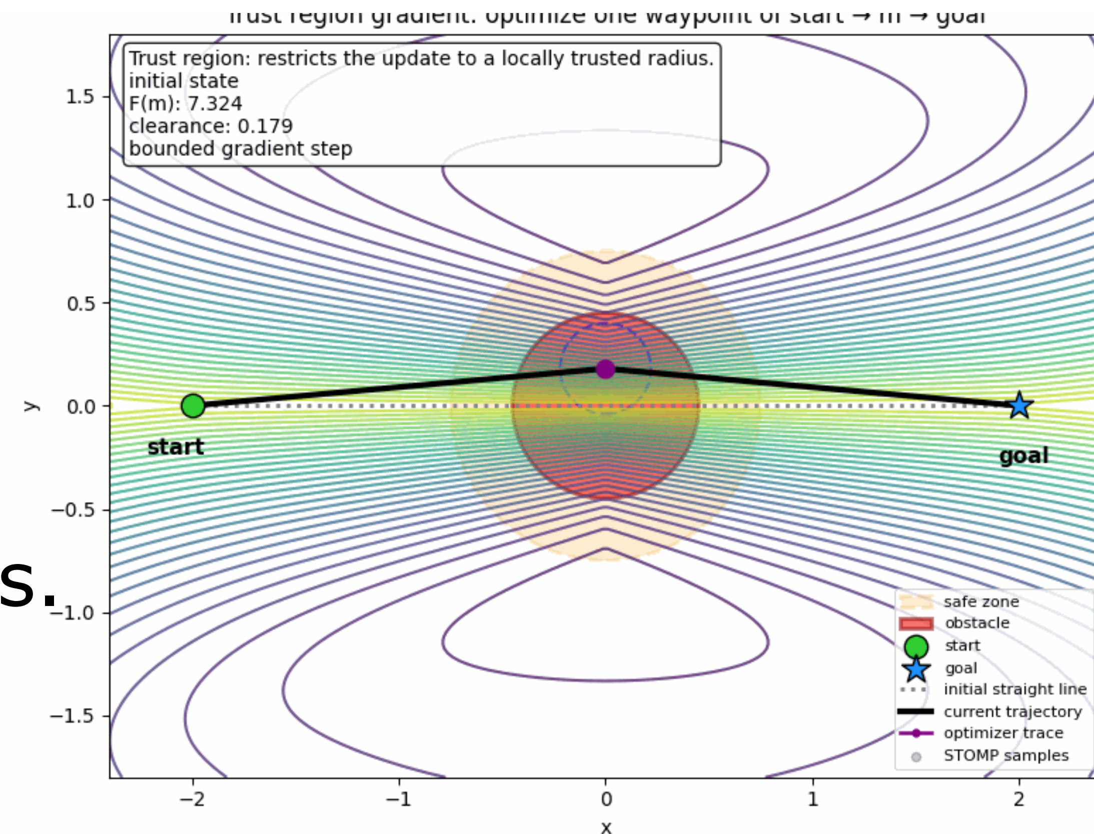
Levenberg-Marquardt (LM)

- Adds damping factor to Gauss-Newton
- $(J^TJ + \lambda I)\Delta x = -J^Tr$
  - Large λ: Behaves like gradient descent
  - Small λ: Behaves like Gauss-Newton.
- Prevents numerical instability near kinematic singularities.
- Common solver for robust numerical IK



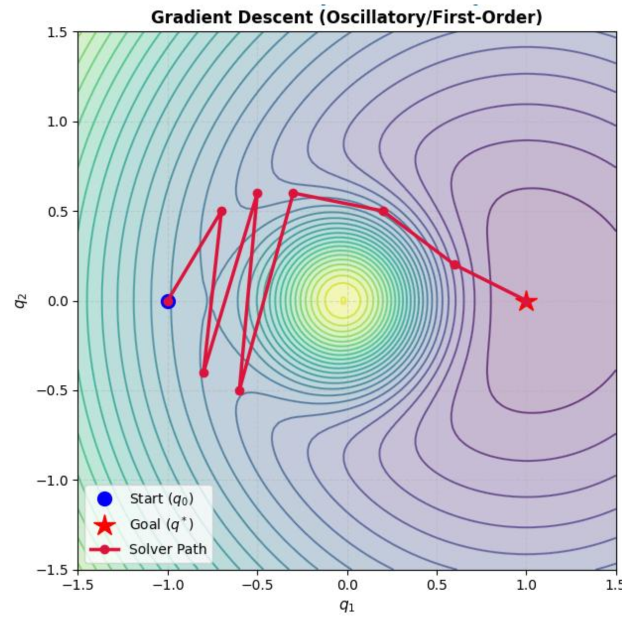
Trust Region

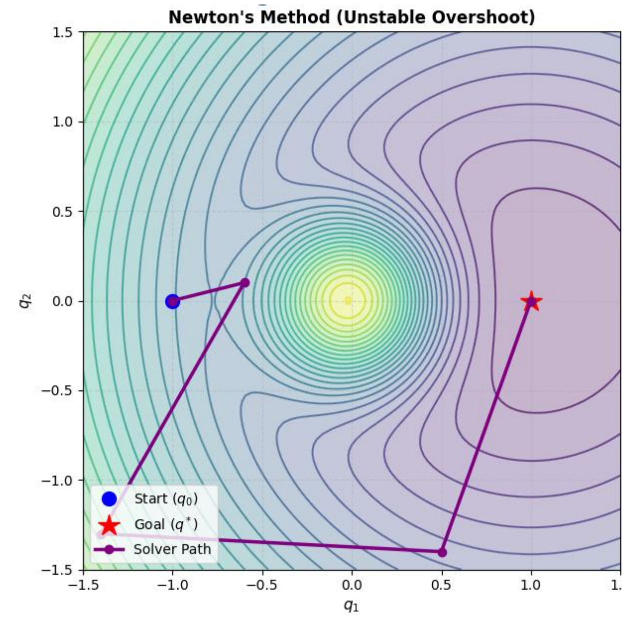
- Keeps updates near linearization point
- Restricts maximum parameter update magnitude
- $\min_{\Delta x} m(\Delta x), s.t. . \|\Delta x\| \leq \Delta$
- Radius Δ shrinks if prediction fails.
- Radius Δ expands on successful predictions.
- Provides convergence safeguards under suitable assumptions.



Solver Comparison for Goal Reaching

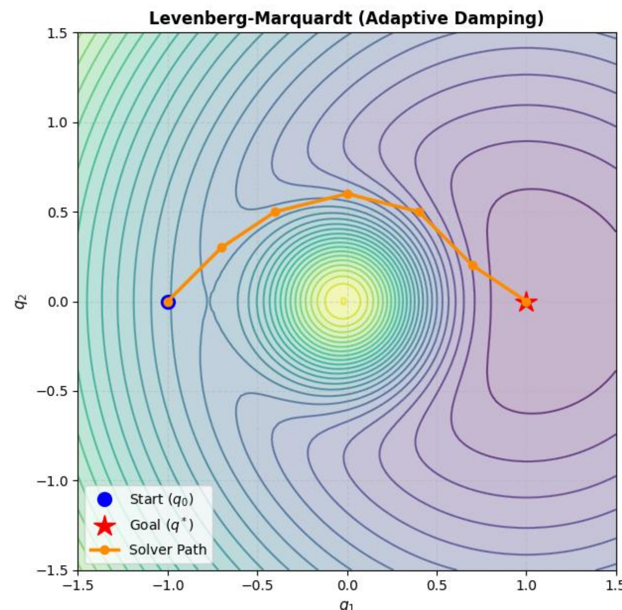
Gradient Descent

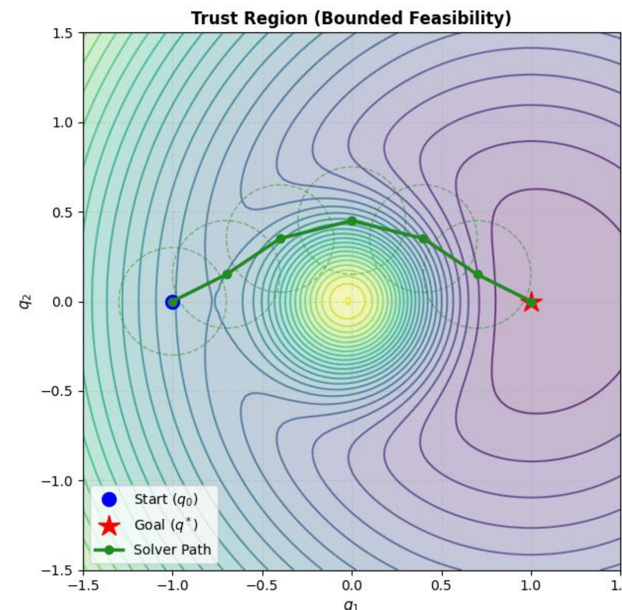




Newton Method

Levenberg-Marquadt





Trust Region

Differential IK vs Optimization-Based IK

- **Differential IK**
- Uses local Jacobian update
- Pose error drives update
- Often uses J^{-1}, J^{+} *or* J^{T}
- Constraints handled separately
- Sensitive near singularities
- E.g. KDL solvers

- **Optimization-based IK**
- Minimizes weighted objective
- Multiple residuals drive soln.
- Use a non-linear solver
- Constraints become bounds or soft penalties
- Damping improves robustness
- E.g. DawnIK

DawnIK

DawnIK:

Decentralized Collision-Aware Inverse Kinematics Solver for Heterogeneous Multi-Arm Systems

Salih Marangoz Rohit Menon Nils Dengler Maren Bennewitz
Humanoid Robots Lab, University of Bonn

Optimization IK to Trajectory Optimization

- IK optimizes one **state** whereas Trajectory optimizes **sequence of waypoints jointly**
- **Trajectory Optimization:**
  - Encode **task goals**, **smoothness**, and **clearance**
  - Enforce joint **limits** and other **feasibility constraints**
  - Map workspace errors back to joint updates
  - **Linearize** nonlinear terms around current trajectory
  - Solve repeated local QPs using sequential convex optimization (SCO)

What is Optimized? (Notation)

- Full trajectory is a stacked vector of waypoints: $\mathbf{Q} = [\mathbf{q}_1^{\mathrm{T}}, \mathbf{q}_2^{\mathrm{T}}, \ldots, \mathbf{q}_M^{\mathrm{T}}] \in \mathbb{R}^{MN}$ where M: Number of waypoints, N: number of joints
- Each waypoint is a joint configuration $\mathbf{q}_1 \in \mathbb{R}^{\mathbf{N}}$
- Forward kinematics: $\mathbf{x} = f(\mathbf{q}_\mathrm{i}) \in \mathbf{SE(3)}$
- Jacobian matrix $J(\mathbf{q}_\mathrm{i})$: $\mathbf{\Delta x} = J(\mathbf{q}_\mathrm{i})\Delta\mathbf{q}_\mathrm{i}$
- Objective function minimizes cost over the whole trajectory: $\mathbf{Q}^* = \arg\min_{\mathbf{Q}} F(\mathbf{Q})$
- Solver iteration k uses superscript: $\mathbf{Q}^{\mathbf{k}}$

The Composite Objective Function

- Combines requirements into one cost F(Q).

$$\mathbf{F(Q)} = \mathbf{w_s F_s(Q)} + \mathbf{w_c F_c(Q)} + \mathbf{w_g F_g(Q)}$$

- **Smoothness** cost penalizes large accelerations using finite differences $F_s(Q)$: $\sum \|q_{i+1} - 2q_i + q_{i-1}\|^2$
- **Goal** cost pulls optimized pose to goal pose $F_g(Q)$: $\left\|r_{pose}\left(f(q_M), x_{goal}\right)\right\|^2$
- **Collision** cost penalizes low obstacle clearance using a hinge loss $F_c(Q)$: $\sum \left(max\left(0, d_{safe} - d\left(f(q_i)\right)\right)\right)^2$

- Weights ($\boldsymbol{w}$) define the motion trade-off

Workspace to Joint Mapping

- Goals and obstacles are defined in workspace
- The optimizer updates joint configurations
- Let $\boldsymbol{p}(\boldsymbol{q}) \in \mathbb{R}^{\mathbf{3}}$ robot body point, $\mathrm{d}(\mathbf{p})$ be signed distance to obstacles (Chapter 2)
- The chain rule maps workspace gradients to joint-space gradients using translational Jacobian $J_p(\mathbf{q})$:
  $\boldsymbol{\nabla}_{\mathbf{q}}\mathbf{d}\big(\mathbf{p}(\mathbf{q})\big) = \mathbf{J}_{\mathbf{p}}(\mathbf{q})^{\mathbf{T}}\boldsymbol{\nabla}_{\mathbf{p}}\mathbf{d}(\mathbf{p})$
- This gives the joint update direction for improving clearance

Why Sequential Convex Optimization (SCO)

- FK and collision distances are non-linear
- Collision avoidance is non-convex
- Hence trajectory optimization is non-convex

$$\mathbf{Q}^* = \arg\min_{\mathbf{Q}} F(\mathbf{Q})$$

- **Graph intuition**
  - Replace non-convex cost by a local convex approximation
  - Keep updates local, then re-linearize at the new trajectory

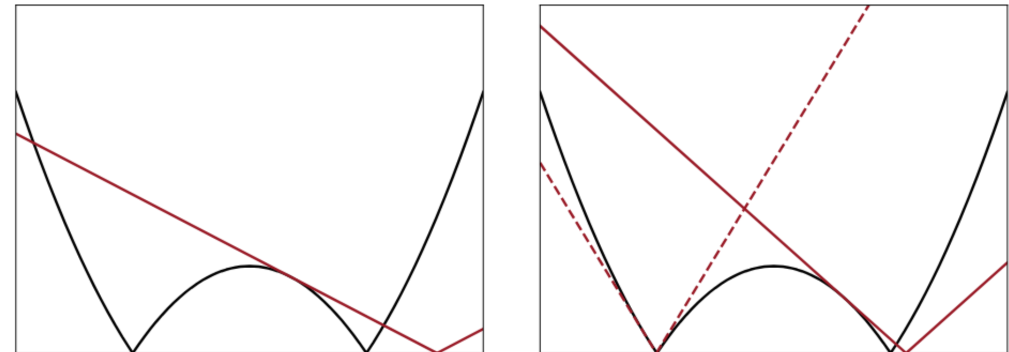


Figure 3. The function $f(x) = |x^2 - 1|$, along with convex approximations to it, local to the point x, given by $f_x(y) = |x^2 + 2x(y - x) - 1|$. This is the composite model of f given

Sequential Convex Optimization (SCO)

- Start from current trajectory $\mathbf{Q}^{(k)}$
- Optimize only a local update $\Delta\mathbf{Q}$
- Solve a convex approximation around $\mathbf{Q}^{(k)}$

$$\mathbf{Q}^{(k+1)} = \mathbf{Q}^{(k)} + \Delta\mathbf{Q}$$

- Approximate it by a convex subproblem $\Delta Q^* = \arg\min_{\Delta\mathbf{Q}} m_k(\Delta\mathbf{Q})$

s.t. linearized constraints, $\|\Delta\boldsymbol{Q}\| \leq \Delta$

Algorithm 1 Sequential Convex Optimization for Trajectory Optimization

Require: Initial trajectory $Q^{(0)}$, trust-region radius Δ, tolerance ϵ

1: Set iteration index $k \leftarrow 0$
2: **repeat**
3: Linearize the objective and constraints around $Q^{(k)}$
4: Construct a local convex model $m_k(\Delta Q)$
5: Solve the convex subproblem:

$$\Delta Q^* = \arg\min_{\Delta Q} m_k(\Delta Q)$$

s.t. linearized constraints,

$\|\Delta Q\| \leq \Delta$

6: Update the trajectory:

$$Q^{(k+1)} \leftarrow Q^{(k)} + \Delta Q^*$$

7: $k \leftarrow k + 1$
8: **until** $\|\Delta Q^*\| < \epsilon$
9: **return** locally optimized trajectory $Q^{(k)}$

From Collision Linearization to QP (Part 1)

- Collision distance is nonlinear in joint space.
- Linearize signed distance around current trajectory

$$\mathbf{d}(\mathbf{p}(\mathbf{q})) \approx \mathbf{d}\left(\mathbf{p}\left(\mathbf{q}^{(\mathbf{k})}\right)\right) + \nabla_{\mathbf{p}}\mathbf{d}(\mathbf{p}_{\mathbf{k}})^{\mathbf{T}}\mathbf{J}_{\mathbf{p}}\left(\mathbf{q}^{(\mathbf{k})}\right)\left(\mathbf{q} - \mathbf{q}^{(\mathbf{k})}\right)$$

- Obtain linearized joint space collision constraints using workspace to joint-space mapping

$$\mathbf{d}(\mathbf{q}) \approx \mathbf{d}(\mathbf{q}_{\mathbf{k}}) + \nabla\mathbf{d}(\mathbf{q}_{\mathbf{k}})^{\mathbf{T}}\Delta\mathbf{q}$$

From Collision Linearization to QP (Part 2)

- Smoothness cost remains a quadratic objective.
- Joint limits remain hard linear constraints.
- Trust region limits steps where approximation is unreliable.
- Result: each SCO step becomes a local QP

$$\boldsymbol{\Delta Q^{*} = \arg\min_{\Delta Q} m_{k}(\Delta Q)}$$

- s.t. linearized collision constraints, joint limits, $\|\Delta \boldsymbol{Q}\| \leq \Delta$
- Optimization becomes a sequence of QPs.

Continuous Collision Checking

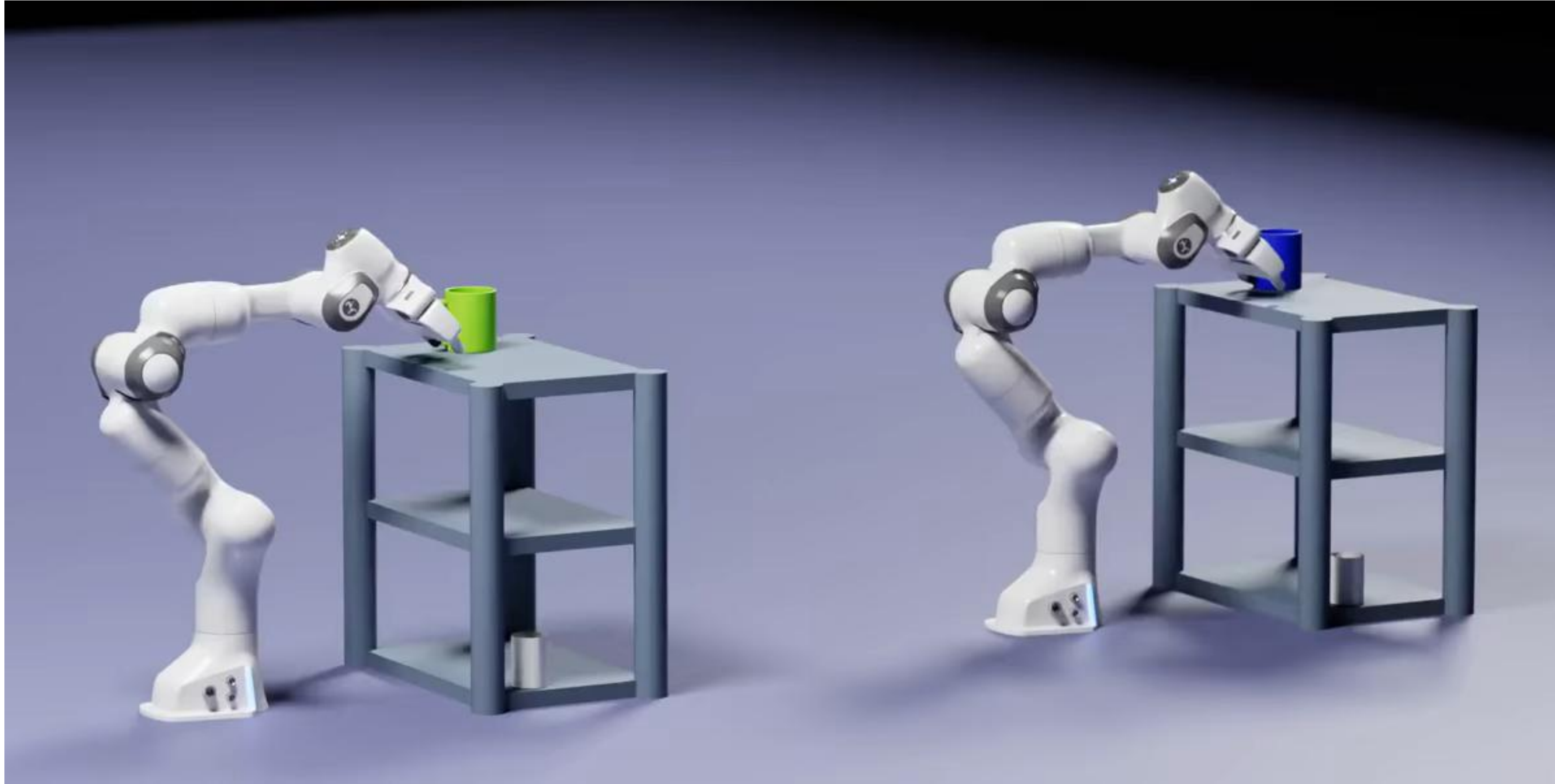
- Discrete waypoints miss intermediate continuous collisions.
- Define continuous segment parameter s ∈ [0, 1].
- $q(s) = q_i + s(q_{i+1} - q_i)$
- Evaluate constraints along swept arm volumes.
- **Reduces missed collisions between discrete waypoints.**

Trajectory Optimization Overview

- **Initialize trajectory** from path, IK, or interpolation
- **Define costs**: goal, smoothness, clearance
- **Define constraints**: joint limits, collision, trust region
- **Linearize** FK and collision terms.
- **Solve** local QPs and **re-check** continuous collisions

Initialize → Evaluate costs/constraints → Linearize → Solve QP → Update → Check collisions

Trajectory Optimization Example: Curobo



Summary

- Trajectory generation adds timing and physical feasibility.
- Trajectory optimization directly refines the full motion.
- Costs encode preferences (smoothness, reaching, clearance).
- Constraints encode limits (joint boundaries, collisions).
- Obstacle avoidance is generally a non-convex problem.
- SCO solves this using QP subproblems and trust regions.