REACHABILITY PROJECT LOG :)

Contents

1. Problem Description

Our goal is to learn *where the robot should stand* given a desired grasp.

1.1. **Problem inputs.** Formulating the problem, we are given as inputs:

- $x_e \in SE(3)$: desired end-effector pose in world coordinates
- $b \in SE(2)$: wheeled mobile base pose (x, y, γ)
- $q \in \mathbb{R}^n$: arm joint configuration.

Date: January 29, 2026.

1.2. **Problem outputs.** We want to find a range of feasible whole-body configurations that satisfy the following requirements:

(1) **IK feasibility:** forward kinematics$(b, q) = x_e$
(2) **Visibility/sensing constraints:** The end-effector should lie within the camera's field of view and range (and un-occluded by the robot itself), i.e. visible$(b, q, x_e) = 1$
(3) **Ball of feasible configurations:** We want a region in joint space around (b, q) that keeps the hand in approximately the right pose and satisfies the above constraints.

Our target is to model some distribution $p(b, q \mid x_e)$ subject to these constraints ↑, ideally with some notion of pose quality (visibility, manipulability, clearance, etc.).

2. 2D robot v3: Point (x, y, ψ) base with a rotary 3-link arm

2.1. **Setup.**

- Robot base configuration: (x, y, ψ) where $\psi \in [0, 2\pi)$ is some arbitrary base heading (robot does *not* need to be facing the target)
- Robot arm configuration: $(\theta_1, \theta_2, \theta_3)$
- Target pose specification: (h_x, h_y, ϕ) (vector with tip at (h_x, h_y) and heading ϕ)

Complete robot configuration specification:

$$(x, y, \psi, \theta_1, \theta_2, \theta_3) \in \mathbb{R}^6$$

2.1.1. *Data generation pipeline.* Clean up the math in this section! Switch all the notation to use 6.4210 transform notation, and re-solve out the formulas cleanly. Also add notes on the rejection sampling / self-collision check we've implemented in the code.

The Math

The goal is to generate a data sampling pipeline for pairs (q, h) where:

$$q = (x, y, \theta_1, \theta_2, \theta_3) \in \mathbb{R}^6 \qquad h = (h_x, h_y, \phi) \in \mathbb{R}^3$$

Defining some notation:

- Frame W: The fixed world frame
- Frame B: The robot base frame
- Frame E: The robot end-effector frame (tip of the arm)
- Frame H: The target hand frame (where we want the end-effector to be)

The variables we have:

- Base configuration: (x_B, y_B, ψ)
- Arm configuration: $\theta = (\theta_1, \theta_2, \theta_3)$
- Target pose: $H = (h_x, h_y, \phi)$

Defining some math: A planar pose (x, y, α) is represented by the matrix $T \in SE(2)$ (the homogeneous transform):

$$T(x, y, \alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & x \\ \sin\alpha & \cos\alpha & y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R(\alpha) & p \\ 0 & 1 \end{bmatrix}$$

Transform inverse:

$$T(p, \alpha)^{-1} = \begin{bmatrix} R(\alpha)^\top & -R(\alpha)^\top p \\ 0 \quad 0 & 1 \end{bmatrix}$$

Defining some notation:

- ${}^A p^C$: Position of C measured from A (vector $\vec{AC}$ = tail at A, head at C), measured from A's origin A_0. We assume this position vector is expressed in A's basis.
- ${}^B R^A$: Orientation of frame A relative to frame B
- ${}^A T^B$: Pose of frame B in frame A
  - Pose = position (location) + orientation of a frame B with respect to another frame A
  - Transform = linear operator that maps a point whose location is known in frame B to that same point's location in frame A.

Using the introduced notation, we define the frame of the base in the world as:

$$^{W}T^{B} = T\left(\begin{bmatrix} x \\ y \end{bmatrix}, \psi\right)$$

Using forward kinematics, for a 3-link planar arm with link lengths $L_1, L_2, L_3 > 0$ and cumulative angles $\alpha_1 = \theta_1, \alpha_2 = \theta_1 + \theta_2, \alpha_3 = \theta_1 + \theta_2 + \theta_3$, we have the following end-effector position in the base frame:

$$^{B}p^{E}(\theta) = \begin{bmatrix} L_1 \cos\alpha_1 + L_2 \cos\alpha_2 + L_3 \cos\alpha_3 \\ L_1 \sin\alpha_1 + L_2 \sin\alpha_2 + L_3 \sin\alpha_3 \end{bmatrix}$$

The end-effector heading in base frame is:

$$^{B}R^{E} = {}^{B}\phi^{E}(\theta) = \alpha_3$$

So the base-to-EE transform is:

$$^{B}T^{E}(\theta) = T(^{B}p^{E}(\theta), {}^{B}R^{E}(\theta)) = {}^{B}R^{E\,E}T^{E} +^{B} p^{E}(\theta)$$

World end-effector transform (forward kinematics):

$$^{W}T^{E} = {}^{W}T^{B\,B}T^{E}(\theta)$$

The target label is the end-effector pose:

$$H = (h_x, h_y, \phi) \iff {}^{W}T^{H} = T\left(\begin{bmatrix} h_x \\ h_y \end{bmatrix}, \phi\right)$$

The kinematic constraint that the end-effector pose = target pose can be written as:

$$^{W}T^{H} = {}^{W}T^{E} = {}^{W}T^{B\,B}T^{E}(\theta)$$

Solving for the base pose:

$$^{W}T^{B} = {}^{W}T^{H}\left[{}^{B}T^{E}(\theta)\right]^{-1}$$

So, to generate (Q, H) data points, we can sample θ, sample a target H, and compute the base pose by one matrix inverse. Math-ing out the above: Let $p_H = \begin{bmatrix} h_x \\ h_y \end{bmatrix}$, and define:

$$^{B}p^{E} = {}^{B}p^{E}(\theta), \quad {}^{B}\phi^{E} = {}^{B}\phi^{E}(\theta) = \theta_1 + \theta_2 + \theta_3$$

Since T_{BE}^{-1} has rotation $-\phi_{BE}$ and translation $-R(\phi_{BE})^{\top} p_{BE}$, composing gives:

$$\text{Base heading: } \psi = \phi - \phi_{BE} \qquad \text{Base position: } \begin{bmatrix} x \\ y \end{bmatrix} = p_H - R(\psi) p_{BE}$$

So given $(H = [h_x, h_y, \phi], \theta = [\theta_1, \theta_2, \theta_3])$, base pose (x, y, ψ) is uniquely determined.

The Pipeline

Inputs/hyperparameters:

- Link lengths L_1, L_2, L_3
- Joint distributions with limits:

$$\theta_1 \sim \text{Unif}[a_1, b_1], \theta_2 \sim \text{Unif}[a_2, b_2], \theta_3 \sim \text{Unif}[a_3, b_3] \Leftarrow \mathcal{D}_\theta$$

- Target sampling distribution: we will assume uniform end-effector position and heading across the workspace

$$h_x, h_y \sim \text{Unif}([-w, w]^2), \quad \phi \sim \text{Unif}[0, 2\pi) \Leftarrow \mathcal{D}_H$$

Pipeline:

(1) Sample arm angles: $\theta = (\theta_1, \theta_2, \theta_3) \sim \mathcal{D}_\theta$

(2) Compute FK in base frame:

$$p_{BE}(\theta) = \begin{bmatrix} \sum_{i=1}^{3} L_i \cos\left(\sum_{j=1}^{i} \theta_j\right) \\ \sum_{i=1}^{3} L_i \sin\left(\sum_{j=1}^{i} \theta_j\right) \end{bmatrix}, \qquad \phi_{BE}(\theta) = \theta_1 + \theta_2 + \theta_3$$

(3) Sample a target pose: $H = (h_x, h_y, \phi) \sim \mathcal{D}_h$. Define $p_H = [h_x, h_y]^{\top}$.

(4) Solve base pose by inversion:

$$\psi = \mathrm{wrap}_{2\pi}(\phi - \phi_{BE}(\theta)), \qquad \begin{bmatrix} x \\ y \end{bmatrix} = p_H - R(\psi) p_{BE}(\theta)$$

(5) Final configuration vector: $Q = (x, y, \psi, \theta_1, \theta_2, \theta_3)$. Store the pair (Q, H), and repeat until you have N samples.

TL;DR: For $i = 1, \ldots, N$, we sample from the joint $(\theta, H) \sim \mathcal{D}_\theta \times \mathcal{D}_H$:

(1) Sample $\theta^{(i)} \sim \mathcal{D}_\theta$
(2) Sample $H^{(i)} \sim \mathcal{D}_H$
(3) Deterministically compute $Q^{(i)} = f(\theta^{(i)}, H^{(i)})$

2.1.2. *Featurization.* Defining $R = L_1 + L_2 + L_3$ (robot reach),

$Q = (x, y, \psi, \theta_1, \theta_2, \theta_3) \mapsto$

$$\boxed{Q_{\text{feat}} = \left(\frac{\tilde{\Delta}_x}{R}, \frac{\tilde{\Delta}_y}{R}, \cos(\tilde{\psi}), \sin(\tilde{\psi}), \cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2), \cos(\theta_3), \sin(\theta_3) \right)} \in \mathbb{R}^{10}$$

$H = (h_x, h_y, \phi), C = (\ldots) \mapsto \boxed{C_{\text{feat}} = (0, 0)} \in \mathbb{R}^2$

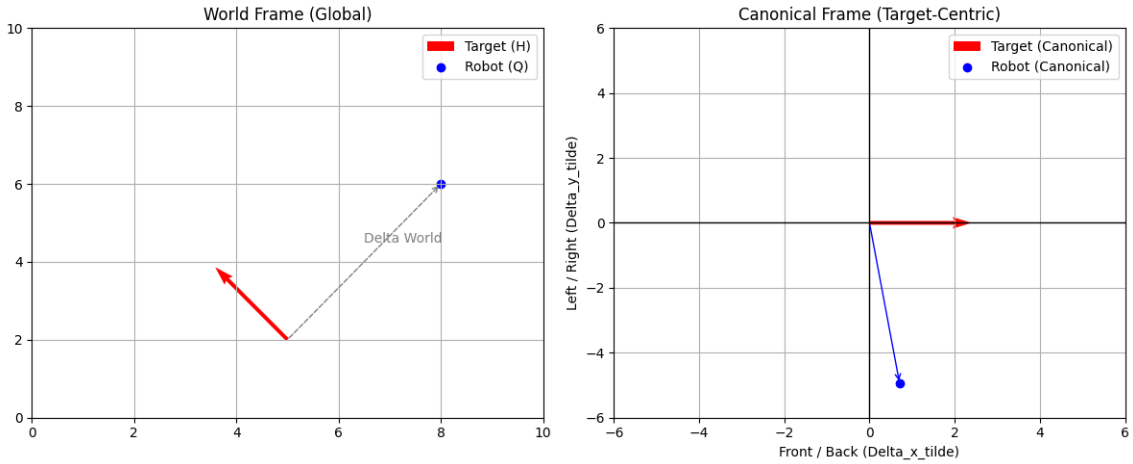
where

$$\tilde{\Delta} = R(-\phi) \cdot \begin{bmatrix} x - h_x \\ y - h_y \end{bmatrix} \qquad \tilde{\psi} = \psi - \phi$$

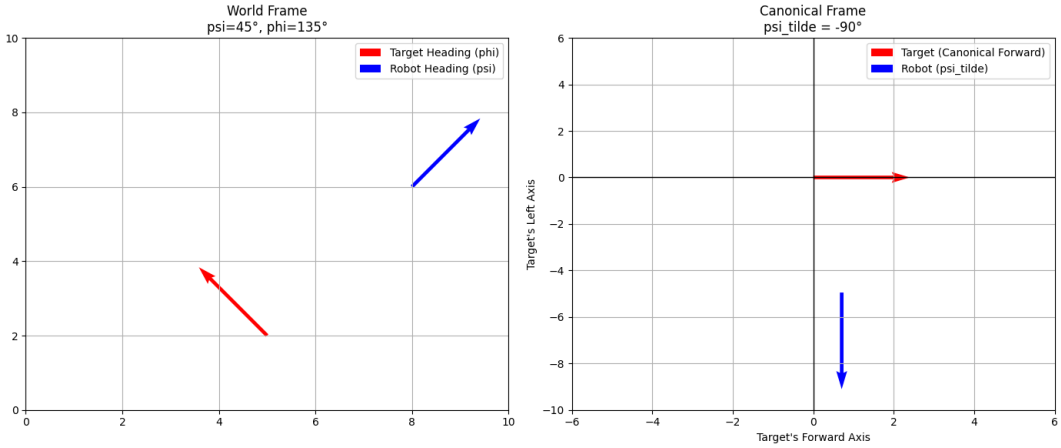
The featurization here canonicalizes the target: we take the robot's world-frame coordinates (x, y, ψ) and re-describe them from the target-frame.

$$\Delta_{\text{world}} = \begin{bmatrix} x - h_x \\ y - h_y \end{bmatrix}$$

This vector tells us how far the robot is from the target in the orientation of the room. To align the world with the target's heading, we multiply Δ_{world} with the inverse of the rotation matrix of the target's heading: this way, the target's heading becomes the new 0°.



In addition, we featurize $\tilde{\psi} = \psi - \phi$ to extract the relative angle between the robot base heading and the target heading: this relative angle is what matters to the model when deciding reachability (the compass of the world frame does not affect the relative relationship between robot and target). We see this transformation in the figure below:



2.2. **Evaluation metrics.** Come up with new evaluation metrics now that we have more joint links! (e.g. MMD, EMD, etc.)

3. Simplified problem v2.1: Round thing with a rotary 2-link arm, generalizable to different workspace sizes

Context: The previous setup specified in section 4 learns a model that is sensitive to workspace size. We fix this problem in this new training setup.

3.1. Setup.

- Data generation pipeline: Same as the data generation pipeline described in section 4.1.
- Modeling assumptions:
  - Robot heading should face the target H
- Featurization: defining $R = L_1 + L_2$ (robot reach),

$$Q = (x, y, \psi, \theta_1, \theta_2) \mapsto \boxed{Q_{\text{feat}} = \left(\frac{x - h_x}{R}, \frac{y - h_y}{R}, \cos(\psi), \sin(\psi), \cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2) \right)}$$

$$H = (h_x, h_y), C = (\dots) \mapsto \boxed{C_{\text{feat}} = (0, 0)}$$

  The feature vector H_{feat} is no longer necessary because H has been encoded into our featurization for (x, y) in Q_{feat}, but we keep it around in the code with `dCond = 0` in case we want to add conditioning features later (e.g., conditioning on pictures of the environment, obstacles, terrain, etc.).
- Model learning setup:
  - Input: Q_{feat} (+ empty conditioning vector C_{feat})
  - Goal: Model learns distribution $\boxed{p(Q_{\text{feat}} \mid C_{\text{feat}})}$ that maximizes the predicted probability of seeing the ground truth data distribution $(Q_{\text{feat}}, C_{\text{feat}})$ pairs.
- Things that the model should generalize across:
  - The learned model should be workspace invariant:
- Things that the model will not be capable of generalizing across:
  - Hardware parameters:
    - Link lengths (`link_lengths`)
    - Joint limits (`joint_limits`)
    - Number of links in robot arm (`n_links`)
  - Tolerance parameters:
    - `base_pos_eps` (this is the ϵ for how thin/thick the donut of allowed base positions is)
    - `base_heading_stddev` (large value means the robot can be loosely facing the target, small value means the robot should near exactly face the target)

3.1.1. *Code setup.* The goal is to learn a model for $p(Q_{\text{feat}} \mid C_{\text{feat}})$, where C_{feat} encapsulates the featurized form of all information we want to condition on.

Name	**Description**	**Code**: name; dimension
Q	Complete robot configuration (base position, arm configuration) $\in \mathbb{R}^5$	`q_world`; `d_q`
Q_{feat}	Featurized robot configuration [input to models] $\in \mathbb{R}^8$	`q_feat`; `d_q_feat`
H	Target robot hand poses H corresponding to Q [input to models] $\in \mathbb{R}^8$	`h_world`; `d_h`
C	Raw values for conditioning variables (robot end-effector target pose H) $\in \mathbb{R}^2$	`c_world`; `d_c`
C_{feat}	Featurized conditioning information [input to models] $= [0, 0] \in \mathbb{R}^2$	`c_feat`; `d_c_feat`
$\hat{Q}$	Given H, sampling some matching $\hat{Q}$ from the learned model $p(Q_{\text{feat}} \mid C_{\text{feat}})$	`q_sample` or `q_hat`

TABLE 1. Unified notation

3.2. **Featurization.**

$$Q = (x, y, \psi, \theta_1, \theta_2) \mapsto Q_{\text{feat}} = \left(\frac{x - h_x}{R}, \frac{y - h_y}{R}, \cos(\psi), \sin(\psi), \cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2) \right)$$

We now prove that this feature vector Q_{feat} is invariant to translation (moving the robot) and workspace boundary changes (changing the room size).

Proof: Translation invariance. Say we train the robot in a lab at base coordinates (x, y) and target coordinates (h_x, h_y). We deploy it in a warehouse at base coordinates (a, b) and target coordinates (h_a, h_b). We want the model to learn a translation invariant mapping, so to the model, $(x, y) \in [-10, 10]^2$ (assuming we bound training data within some arbitrary square) should be equivalent to $(a, b) \in [-10000, 10000]^2$ etc. for all the coordinates. We show below that our Q_{feat} feature transformation makes it s.t. coordinate scale does not matter to the model (hence achieving translation invariance).

Let the shift vector be $d = (d_x, d_y)$: $(a, b) = (x + d_x, y + d_y)$, $(h_a, h_b) = (h_x + d_x, h_y + d_y)$. By our definition of Q_{feat},

$$\begin{aligned} Q_{\text{feat}} &= \left(\frac{x + d_x - (h_x + d_x)}{R}, \frac{y + d_y - (h_y + d_y)}{R}, \text{angle features} \right) \\ &= \left(\frac{x - h_x}{R}, \frac{y - h_y}{R}, \text{angle features} \right) \Rightarrow \text{in distribution with training data for model!} \end{aligned}$$

□

Proof: Workspace boundary invariance. We can also see that our featurization makes the model learned over $p(Q_{\text{feat}})$ invariant to workspace boundary definition because no features in Q_{feat} are dependent on attributes of the training workspace:

$$Q_{\text{feat}} = \left(\frac{x - h_x}{R}, \frac{y - h_y}{R}, \cos(\psi), \sin(\psi), \cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2) \right) = \left(\frac{\Delta_x}{R}, \frac{\Delta_y}{R}, \dots \right)$$

By definition, Δ is invariant to workspace boundary (since all that matters is the relative vector between the base and the target pose). □

(Sidenote: the previous design of Q_{feat} in section 4.1 was translation invariant but *not invariant to workspace boundary.*)

3.3. **Notes on this setup.**

3.3.1. *Collapsing the problem from a conditional generative problem $p(Q \mid H)$ to an unconditional problem $p(Q_{feat})$.* Our definition of Q_{feat} essentially performs a change of variables (coordinate canonicalization):

- **The conditional view (allocentric/world frame):** We model $p(Q \mid H_{\text{global}})$. The density depends on H because the "cloud" of valid solutions moves around the room as H moves.
- **The relative view (egocentric/canonical frame):** We define $\Delta = Q[0:2] - H_{\text{global}}$ (normalized). We model $p([\Delta, \text{angle features}] \mid H_{\text{global}})$.

If the workspace is infinite and empty, the distribution of valid relative poses does not change depending on where you are in the room, i.e. $p([\Delta, \text{angle features}] \mid H_{\text{global}}) = p([\Delta, \text{angle features}])$. Our current workspace is infinite and empty, so we don't actually need to condition on H_{global} anymore $\rightarrow$ conditional model has become unconditional!

3.3.2. *Why we keep the conditional architecture.* Given the analysis in section 3.3.1, why do we keep the conditional architecture in the code?

While the problem is unconditional in an infinite void, it will become conditional when we move into the "real world."

E.g.:

- **Target orientation:** Currently, the target H is a point (x, y). In the real world, targets have orientation (e.g., the handle of a mug).
  - Relative position Δ is not independent of the mug's rotation $\rightarrow$ will need to condition $p([\Delta, \text{angle features}] \mid H_{\text{angle}})$
- **Obstacles:** In a real environment, the validity of a stance Δ depends on nearby obstacles.
  - If there is a wall 0.5m to the right of the target, you can't stand there $\rightarrow$ will need to condition on a "local occupancy grid" etc. $p([\Delta, \text{angle features}] \mid \text{LocalMap})$

etc.! So we keep our model with a conditional architecture despite not feeding any conditions in for the current infinite void setup. In the future, we can potentially condition on

$$H_{\text{context}} = [\cos(\theta_{\text{target}}), \sin(\theta_{\text{target}}), \text{dist\_to\_nearest\_obstacle}, \ldots]$$

3.3.3. *Egocentric learning (canonicalization) papers.* Read more on this!

- SE(3)-equivariant networks (manually enforce SE(2) invariance)
- Transporter networks (solve robotic manipulation by learning a "kernel" of the object in a canonical frame and then sliding it across the image (convolution) to find a match)
- Residual policy learning (often learns a delta-action in a local frame rather than absolute actions)

3.4. **Training.** Refer to section 4.3.

3.5. **Evaluation metrics.** In addition to the ones introduced in section 5.4, we add the following metrics:

- Maximum mean discrepancy (MMD): TL;DR, MMD is defined by the idea of representing distances between distributions as distances between mean embeddings of features.
- EMD:

4. Simplified problem v2.0: Round thing with a rotary 2-link arm

4.1. **Setup.**

- Data generation pipeline:

  (1) Sample the target $H = (h_x, h_y)$ from our desired task distribution:

  $$H \sim \text{Uniform(workspace bounds)}$$

  (2) Sample base position (x, y) that can reach H via sampling from the "donut" of valid positions:
  - For the arm to reach H, the base must be at a distance r away from H where $r \in [R_{\min}, R_{\max}]$ and $R_{\min} = |L_1 - L_2|$ (folded in), $R_{\max} = L_1 + L_2$ (fully extended).
  - We also want to avoid $R_{\min}$ and $R_{\max}$ exactly in the real world to avoid singularities, so we set some arbitrary ϵ as a proxy for manipulability (require the base sit some distance $r \in [R_{\min} + \epsilon, R_{\max} - \epsilon]$ away from H)
  - $\uparrow$ This sampling is implemented as:

  $$r \sim \text{Uniform}(R_{\min} + \epsilon, R_{\max} - \epsilon)$$
  $$\phi \sim \text{Uniform}(0, 2\pi)$$

  Computing base (x, y) coordinates from polar coordinates:

  $$\boxed{x = h_x + r\cos(\phi), \quad y = h_y + r\sin(\phi)}$$

  (3) We assume the robot should be somewhat facing the target:

  - Perfectly facing the target: $\psi_{\text{ideal}} = \text{atan2}(h_y - y, h_x - x)$
  - Add noise to prevent the model from collapsing to a single deterministic mapping:

$$\boxed{\psi = \psi_{\text{ideal}} + \delta}, \quad \delta \sim \mathcal{N}(0, \sigma^2)$$

    where heading variance σ^2 is defined at model initiation (can manually pick).

(4) Analytical arm IK (solve for θ_1, θ_2): Now that base $B = (x, y, \psi)$ is fixed, transform H into the robot's local frame:
   (a) Local target: $h_{\text{local}} = R(\psi)^\top (H - B[0:2]) \coloneqq (u, v)$
   (b) Solve θ_2 (elbow): Using law of cosines:

$$\cos(\theta_2) = \frac{u^2 + v^2 - L_1^2 - L_2^2}{2L_1 L_2} \coloneqq m$$

   There are two solutions $\uparrow$ $\boxed{\theta_2^+ = \arccos(m), \theta_2^- = -\arccos(m)}$ since $\cos(x) = \cos(-x)$
   (c) Solve θ_1 (shoulder):

$$\boxed{\theta_1 = \text{atan2}(u, v) - \text{atan2}(L_2 \sin\theta_2, L_1 + L_2 \cos\theta_2)}$$

(5) At this step, we have generated $Q = (x, y, \psi, \theta_1, \pm\theta_2)$ for H! To enforce joint limits, we can filter to only keep (Q, H) where $\theta_1, \theta_2 \in$ [joint limits] in the dataset.
(6) *Future additions:* If we want to address real world priors by biasing the dataset, i.e. adding cost functions for manipulability, clearance, etc.: we can incorporate these into dataset generation! For example, if we have some manipulability scoring function $S(Q)$, we can hard filter out Q where $S(Q) <$ threshold or soft filter accept Q with probability $P = \frac{S(Q)}{S_{\max}}$

- Modeling assumptions:
  - Robot heading should face the target H
- Featurization (elaborated in section 4.4.1, the way that we implement coordinate featurization results in model dependence on workspace size, as explained below $\rightarrow$ next step is to deprecate this normalization method!!)

$$Q = (x, y, \psi, \theta_1, \theta_2) \mapsto \boxed{Q_{\text{feat}} = (g(x), g(y), \cos(\psi), \sin(\psi), \cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2))}$$

$$H = (h_x, h_y) \mapsto \boxed{H_{\text{norm}} = (g(h_x), g(h_y))}$$

  where g is a coordinate-normalization function to bound x, y coordinates within a normalized range.
- Model learning setup:
  - Input: $Q_{\text{feat}}, H_{\text{norm}}$
  - Goal: Model learns distribution $p(Q_{\text{feat}} \mid H_{\text{norm}})$ that maximizes the predicted probability of seeing the ground truth data distribution $(Q_{\text{feat}}, H_{\text{norm}})$ pairs.

4.1.1. *Critical error with this setup.* **Problem:** This setup currently is dependent on the workspace being a nice rectangle for us to do normalization on the (x, y) coordinates. We have two methods $g(x, y)$ for normalizing coordinates right now that are mathematically equivalent except scaled, and both of them are problematic for learning a model that generalizes across varying workspace bounds:

- Bound normalization (`bound`):

$$x \mapsto 2 \cdot \frac{x - x_{\min}}{x_{\max} - x_{\min}} - 1, \quad y \mapsto 2 \cdot \frac{y - y_{\min}}{y_{\max} - y_{\min}} - 1$$

- Standardization with geometric z-score (`standardize`):

$$x_{\text{center}} = \frac{x_{\max} + x_{\min}}{2}, \quad y_{\text{center}} = \frac{y_{\max} + y_{\min}}{2}, \quad x_{\text{scale}} = \frac{x_{\max} - x_{\min}}{4}, \quad y_{\text{scale}} = \frac{y_{\max} - y_{\min}}{4}$$

$$x \mapsto \frac{x - x_{\text{center}}}{x_{\text{scale}}}, \quad y \mapsto \frac{y - y_{\text{center}}}{y_{\text{scale}}}$$

We can show that these two are the same normalization method up to a constant scale factor: define $W = x_{\max} - x_{\min}$, $C = (x_{\max} + x_{\min})/2$.

$$\text{Bound: } x \mapsto 2 \cdot \frac{x - x_{\min}}{x_{\max} - x_{\min}} - 1 = \frac{2(x - x_{\min})}{w} - \frac{w}{w}$$
$$= \frac{2(x - x_{\min}) + x_{\min} - x_{\max}}{W} = \frac{2x - x_{\min} - x_{\max}}{W} = \boxed{\frac{2(x - C)}{W}}$$

$$\text{Standardize: } x \mapsto \frac{x - x_{\text{center}}}{x_{\text{scale}}} = \frac{x - \frac{x_{\max} + x_{\min}}{2}}{\frac{x_{\max} - x_{\min}}{4}} = \frac{4x - (2x_{\max} + 2x_{\min})}{x_{\max} - x_{\min}} = \boxed{\frac{4(x - C)}{W}}$$

The problem with this normalization method is that we convert absolute meters into relative "room units": the model sees the robot's reach not as "1.5 meters," but as "0.15 room units."

- If we change the workspace size from 10×10m to 20×20m: the model still thinks the arm reaches "0.15 room units," 0.15 room units went from 1.5m in 10×10 room to 3m 20×20 room, yet the physical robot arm is still only 1.5 meters long. The result is that the model will place the base too far away from the target when we move to a bigger room.

Let us walk through what the model is learning in this situation:

- **The physical truth:** The robot has a fixed physical reach L_{arm} (e.g., 1.5m). For target h to be reachable from base position b, we must have:
$$||b - h|| \leq L_{\text{arm}}$$
- **The model's truth:** The model receives normalized coordinates $\hat{b}$ and $\hat{h}$. It learns a relationship between them based on the training data: *the model does not know meters, it only knows "normalized units."* The model effectively learns a normalized reach threshold K_{learned}: (to think about more: how would a cVAE/cINN represent this architecturally? what would the gaussians need to look like?)
$$||\hat{b} - \hat{h}|| \leq K_{\text{learned}}$$
- **The problem:** Now, let's substitute the normalization formula into the network's learned truth. Since $\hat{x} = \frac{4(x-C)}{W}$ (the scaling by 4 or 2 doesn't matter for the math below):
$$||\hat{b} - \hat{h}|| = \left|\left|\frac{4(b - C)}{W} - \frac{4(h - C)}{W}\right|\right| = ||b - h|| \cdot \frac{4}{W}$$
Rearranging to see what physical distance the model *thinks* corresponds to reach:
$$||b - h|| \leq K_{\text{learned}} \cdot \frac{W}{4}$$
↑ This shows a critical issue: The robot's physical reach is now defined as a percentage of the room width. A concrete example of this going wrong:
  - **Training:** You train in a 10m room ($W = 10$). The robot arm is 1.5m. The model learns $K_{\text{learned}} = 0.6$ (because $0.6 \times \frac{10}{4} = 1.5$)
  - **Testing:** You deploy in a 20m room ($W = 20$). The model still applies the rule $K_{\text{learned}} = 0.6$.
    - The model generates a base position such that the normalized distance $||\hat{b} - \hat{h}||$ is 0.6.
    - Converting back to physical units: distance $= 0.6 \times \frac{20}{4} = 3$ meters.
    - Result: The model places the robot 3 meters away from the target, but the robot arm is still only 1.5m long ⇒ robot cannot reach the target!

TL;DR: This model will not generalize to workspaces that are of different size than the one the training data came from!

4.1.2. *Fixing this setup.* See section 3.1.

4.2. **Forward kinematics.**

4.2.1. *`fk_mse_from_qfeat` loss function math.* Input to function: `mu_q` (μ_q = model's prediction of the robot's state) and H. We want to take this prediction, convert it into a valid physical configuration, calculate where the arms' tip should be in the world and compare that to a target position H. Write out nicely later.

4.3. **Training.**

4.3.1. *Nearest neighbors baseline.*

```
=== Results: NNDeterministicLookup ===
hand_err/mean 0.014401
hand_err/median 0.013445
hand_err/p95 0.028278
coverage/max_gap_mean 6.283185
coverage/max_gap_p95 6.283185
coverage/kl_to_uniform 0.019053
stochasticity/var_Q_fixed_H 0.000000
```

4.3.2. *Conditional VAE.* **Metrics to track:**

- Total loss
- Reconstruction loss term
- KL loss term
- Mean of $|\mu|$: if near 0 always, encoder might be collapsing
- Mean of logvar: if very negative → tiny standard deviation → near-deterministic encoder
- Gradient norm: detect instability or dead training
- KL per dimension to show if any of the latent dimensions are "dead" (near-zero KL contribution)

  Why track D_{KL}(Standard normal || latent dimension i) across all latent dimensions?

  (1) This KL term measures the "cost" of sending information through the latent channel:
  - High KL: The encoder is writing a lot of information into this dimension (it deviates strongly from the prior)
  - Zero KL: The encoder is outputting prior $\mathcal{N}(0, 1)$ regardless of the input. This dimension is "dead."

  It is common for VAEs (especially with powerful decoders) to suffer from *posterior collapse* where the decoder decides it is powerful enough to guess the output without looking at the latent code z. When this happens, the encoder "gives up" and just outputs noise (the prior) to satisfy the KL loss perfectly [since the collective loss is reconstruction + KL — if reconstruction loss is the same regardless of what we have for z, the model will learn to output z that induces minimal KL loss, i.e. directly output the prior over z]. Thus, if we see all KL dimensions = 0, the model has collapsed: the encoder is learning nothing.

  (2) The manifold hypothesis (sparsity):
  - Real-world data usually lies on a lower-dimensional "manifold."
  - E.g., you are modeling a 2D robotic arm with 2 DoF. You set the model's latent dim to 10. Ideally, you want the model to discover that only 2 dimensions are needed.
    * Good result: 2 dimensions have high KL (encoding the angles), and 8 dimensions have KL ≈ 0 (noise).
    * Bad result: The information is smeared across all 10 dimensions (entanglement)

  We generally prefer disentangled representations where specific latent dimensions map to specific physical properties. If `kl_dim3` spikes only when the robot rotates its wrist etc., that dimensions has learned a specific semantic feature.

  (3) Auto-pruning: Because the VAE objective function is Loss = Reconstruction + $\beta \cdot$ KL, the $\beta \cdot$ KL term acts as a penalty for using latent capacity. It forces the model to be efficient. It effectively asks the model: "Is this extra dimension necessary to reduce reconstruction error? If not, set it to the prior (KL = 0) to save cost."

How is `z_dim` usually determined?

- Partly an art and partly science! Conceptually, `z_dim` represents the number of independent variables the model needs to describe the data. Some standard methods:

  (1) Physical approach (intrinsic dimensionality): e.g. count the number of DoF of the system! Set `z_dim` slightly higher than the intrinsic dimension (number DoF) to allow the model some "slack" to model noise/non-linearities that aren't perfectly captured by the physics.

  (2) The "active units" approach (pruning):
  (a) Overshoot: Set `z_dim` larger than necessary.

(b) Regularize: Use β-VAE (where $\beta > 1$)
(c) Inspect: Look at the `kl_per_dim` metrics.

Because the KL term penalizes information storage, the VAE will naturally try to "shut off" unnecessary dimensions (drive their KL to 0). E.g., if we set `z_dim = 16` and see that 5 dimensions have high KL and 1 have near-zero KL, the model has told us that the true dimensionality is 5. We can then retrain with `z_dim = 5` (or 6 etc.) for a more compact model.

(3) Hyperparameter sweep (the elbow plot): If we don't know the physics of the problem (e.g., generating images of faces), we can run a grid search. Train models with `z_dim = [2, 4, 8, 16, 32, 64]` and plot the reconstruction loss (NLL) vs `z_dim`.
   - Low `z`: High error (underfitting). The bottleneck is too tight
   - High `z`: Low error, but diminishing returns with increasing `z` at some point.
   - The "elbow": The point where adding more dimensions stops significantly lowering the reconstruction error = optimal `z_dim`!

What prevents a VAE from memorizing instead of learning when `z_dim` = exact number DoF in system?

- If you gave a standard Autoencoder `z_dim = 5`, it would simply memorize the dataset → it would essentially build a hash map where `Input A` maps to `Point A` in latent space.
- The VAE prevents this through (1) sampling noise and (2) the KL penalty.
  (1) Sampling noise: A standard autoencoder maps an input datapoint to a specific point in latent space (e.g., "This specific robot pose is exactly coordinate `(2.1, -0.5)`"). Contrarily, a VAE maps an input datapoint to a probability distribution in latent space ("This specific robot pose is somewhere in the region centered at `(2.1, -0.5)` with a radius of 0.1"). This prevent memorization: because of the sampling step ($z = \mu + \sigma \cdot \epsilon$), the decoder never sees the same input twice → in epoch 1, the latent z corresponding to x might be 2.1, then 2.05 in epoch 2, 2.11 in epoch 3, etc. → the decoder cannot just memorize "When I see `2.1`, output x," forcing generalization rather than memorization.
  (2) The KL penalty:
     - *Q: What prevents the encoder from making the cloud around μ really, really small?* If the encoder shrinks the variance to zero, the cloud becomes a single point, and the VAE turns back into a standard AE that can memorize everything.
     - ↑ *But the KL divergence term prevents this!* KL loss for a Gaussian $:= D_{KL} = 0.5 \sum(1 + \log(\sigma^2) - \mu^2 - \sigma^2) \implies$ if the model tries to memorize by making $\sigma \to 0$, $\log(0) \to -\infty$ and the loss goes to infinity.

  So for VAEs there is a tradeoff during training! To get low reconstruction error (memorization), the model wants to shrink σ to be precise. To get low KL error (regularization), the model wants to expand σ to be 1 (unit standard normal). TL;DR: Sampling + KL penalty collectively prevent the VAE to act like a lookup table even when `z_dim` $\geq$ true DoF in the system.

4.3.3. *Conditional INN.* cINNs are trained using MLE: we want to maximize the probability of the data point x given condition c. By using the change of variables formula, the log-likelihood is:

$$\log p_X(x \mid c) = \log p_Z(z) + \log |\det J|$$

where:

- $z = f(x; c)$ is the latent vector mapped to by the model
- $p_Z(z)$ is the base distribution (usually a standard Normal $\mathcal{N}(0, I)$)
- $\log |\det J|$ is the log-determinant of the Jacobian (measuring how much the network expands/contracts volume)

Since we minimize NLL, the loss function is:

$$\mathcal{L} = \underbrace{-\log p_Z(z)}_{\text{z\_term}} + \underbrace{(-\log |\det J|)}_{\text{logdet\_term}}$$

Metrics to track:

- Loss components (Track: `z_term`, `logdet_term`):
  - `z_term` is minimal when $p_Z(z) = 1$, i.e. $z = \mu(p_Z)$ (fit the mean of the prior Gaussian)

  - `logdet_term` prevents the model from collapsing all data to a single point: it force the model to maintain volume (read into the math behind this more)
    * Watch for: if `logdet_term` becomes extremely negative (e.g. -10^6) while `z_term` becomes massive, the model is expanding volume infinitely to "cheat" the loss.
- Latent statistics (Track: `z_sq_mean`, `z_norm_mean`)
  - We are forcing z to look like $\mathcal{N}(0, 1)$, so in theory, we know exactly what the statistics of z should be:
    * `z_sq_mean`: since the variance of a standard normal is 1, this value should converge to roughly 1.0
    * `z_norm_mean`: if this explodes (e.g., > 100), the model is mapping some inputs to very extreme regions (since supposedly $z \sim \mathcal{N}(0, 1)$), indicating instability or outliers in the training data.
- Jacobian statistics (Track: `log_detj_std`, `log_detj_min/max`)
  - This measures the "stiffness" of the transformation (read into the theory behind this more)
    * `log_detj_std`: if this is very high, the network is reacting very differently to different samples in the batch $\rightarrow$ this suggests the training surface is rugged or the batch size is too small.
    * `log_detj_min/max`: spikes here often precede `NaN` errors
- Gradient norms (Track: `grad_norm_prev_clip`, `grad_norm_post_clip`)
  - Measures how big the update steps are!
  - Health check: `grad_norm_prev_clip` should eventually settle down $\rightarrow$ if it consistently hits your clip value (causing `post_clip` to be capped), the learning rate is likely too high
- Validation NLL:
  - In theory, cINNs can memorize the training data and achieve arbitrarily low NLL by shrinking the volume around training points.
  - Track this using a held-out validation set during training: if train NLL keeps dropping but val NLL plateaus/rises, stop training!
- To add:
  - Reconstruction error (invertibility check): in theory, cINNs are strictly invertible. Ensure that $x \approx f^{-1}(f(x))$: if this error is high, the "generative" capabilities of the INN are broken, even if NLL is low.

```
with torch.no_grad():
    z = model(x, c)
    x_recon, _ = model.reverse(z, c) # Assuming reverse method exists
    recon_error = torch.mean((x - x_recon)**2)
    # Log "train/recon_mse": recon_error.item()
```

  - Latent mean: We track `z` variance, but even if the model learns a variance of 1.0, if it centers the cloud at $z = 50$, this is still a failure mode $\rightarrow$ the mean of `z` should be 0: track `"train/z_mean": z.mean().item()` and ensure that it hovers around 0

4.3.4. *Conditional diffusion.* Diffusion models are trained using MSE loss: $||\epsilon - \epsilon_\theta(x_t, t)||^2$. Because the loss function forces the model prediction $\epsilon_\theta(x_t, t)$ to match ϵ, the model's output should also roughly approximate $\mathcal{N}(0, I)$.

Metrics to track:

- Predicted noise distribution: Since target noise ϵ is drawn from $\mathcal{N}(0, 1)$, the empirical mean and standard deviation of the model's prdictions $\hat{\epsilon}$ should be:

$$\mathbb{E}[\hat{\epsilon}_\theta] \approx 0, \quad \text{Std dev.}[\hat{\epsilon}_\theta] \approx 1$$

  - Mean $\neq 0$: The model is systematically biased (predicting drift)
  - Std $<<$ 1 (denoising too timid): The model is outputting values too close to zero. This is a common failure mode in diffusion called "regression to the mean," where the model gives up and predicts the average noise (0) instead of the actual noise structure.
  - Std $>>$ 1 (denoising too aggressive): Gradient explosion or scaling instability.
- Loss by timestep: The diffusion loss varies depending on the timestep t (bucket into time chunks)

  - High t (near max steps): The image is mostly noise. Loss is often high because x_t contains very little signal about x_0.
  - Low t (near 0): The image is mostly signal. Loss should be very low.
- Signal-to-noise ratio weighting:
$$\mathrm{SNR}(t) = \frac{\bar{\alpha}_t}{1-\bar{\alpha}_t}$$
where $\bar{\alpha}_t$ is the cumulative product of $1-\beta_t$ from the scheduler. Logging the SNR helps verify that the noise scheduler is actually destroying the signal correctly.

4.4. **Additional notes. Featurization section below is now deprecated**, featurizing this way causes the model to depend on workspace size.

4.4.1. *Featurization.* We featurize data $Q = (x, y, \psi, \theta_1, \theta_2) \in \mathbb{R}^5$ as:
$$Q_{\text{feat}} = (x_{\text{norm}}, y_{\text{norm}}, \cos(\psi), \sin(\psi), \cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2)) \in \mathbb{R}^8$$
We also featurize $H = (h_x, h_y) \mapsto H_{\text{feat}} = (h_{x,\text{norm}}, h_{y,\text{norm}})$ to ensure that the features being conditioned upon are in the same scale.

(1) Angle featurization of angle $\mapsto \cos(\text{angle}), \sin(\text{angle})$ is to bound the values within $[-1, 1]$ and avoid the wrapping problem (where the model outputting 0 is equivalent to 2π but numerically at the two ends).

(2) Base position (x, y) is normalized to avoid arbitrary weighing between rotation features vs. position features (i.e., 2000mm will look a lot "bigger" to the model than 2m, especially relative to the bounded angle features). Two normalization methods are implemented:
- Bounding (`bound` in the code): This method takes any rectangular workspace and squashes it down to a $[-1, 1]^2$ box
$$x \mapsto 2 \cdot \frac{x - x_{\min}}{x_{\max} - x_{\min}} - 1, \quad y \mapsto 2 \cdot \frac{y - y_{\min}}{y_{\max} - y_{\min}} - 1$$
- Standardization with geometric z-score (`standardize` in the code): Define
$$x_{\text{center}} = \frac{x_{\max} + x_{\min}}{2}, \quad y_{\text{center}} = \frac{y_{\max} + y_{\min}}{2}, \quad x_{\text{scale}} = \frac{x_{\max} - x_{\min}}{4}, \quad y_{\text{scale}} = \frac{y_{\max} - y_{\min}}{4}$$
$$x \mapsto \frac{x - x_{\text{center}}}{x_{\text{scale}}}, \quad y \mapsto \frac{y - y_{\text{center}}}{y_{\text{scale}}}$$
Notice the effect of this normalization is that:
$$x_{\min} \mapsto \frac{(x_{\min} - x_{\max})/2}{(x_{\max} - x_{\min})/4} = -2, \quad x_{\max} \mapsto \frac{(x_{\max} - x_{\min})/2}{(x_{\max} - x_{\min})/4} = 2$$
And the same for y! So the workspace has effectively been standardized down to $[-2, 2]$.

Empirically, we observe that method 2 (standardization) leads to much better cINN training performance (`hand_err_mean` of 0.28 vs. 0.062 when all other training conditions are held constant), whilst both lead to similar cVAE performance. Some theories as to why the cINN approach greatly prefers method 2 for base position normalization.

5. Simplified problem v1: Round thing with a fixed stick

5.1. **Setup.** Simple robot is a round thing with a fixed "stick" attached to it: a disk on the floor with a rigid stick of fixed length L glued to it, and the stick rotates with the disk.

- Configuration: $Q = x, y, \theta$ ($\mathbb{R}^2 \times S^1$)
- Hand target: $H = h_x, h_y$ (point in $\mathbb{R}^2$)

In this setup, $H \in \mathbb{R}^2$ is the desired 2D point on the floor where we want the tip of the stick to be.

5.2. **Forward kinematics.**

$$\text{hand}(x, y, \theta) \coloneqq f(Q) = \begin{bmatrix} x \\ y \end{bmatrix} + L \begin{bmatrix} \cos\theta & \sin\theta \end{bmatrix}$$

So the IK constraint ("hand hits the target") is:

$$H = \begin{bmatrix} h_x \\ h_y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + L \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$$

Rearranging shows that as θ varies, the base center (x, y) traces out a circle of radius L around the target point H: the set of feasible base positions for the robot to reach H is a circle centered at H.

Ground truth feasible set for a fixed $H \coloneqq \{Q : f(Q) = H\} = \{(h_x - L\cos\theta, h_y - L\sin\theta, \theta) : \theta \in [0, 2\pi)\}$

5.3. **Approaches.** We try the following approaches to model $p(Q \mid H)$:

(1) *Nearest neighbors* (baseline):

- Given a dataset $\mathcal{D} = \{(H_i, Q_i)\}_{i=1}^N$: for a query H',
  (a) Find indices of the k nearest H_i to H' (under $||H_i - H'||$)
  (b) Return their associated Q_i's (either all or sample one etc.)
- For a new H', we can return the Q_i associated to the nearest H_i, or sample from an empirical conditional distribution created based on the nearest neighbor graph: $\hat{p}_{\text{kNN}}(Q \mid H') = \sum_{j \in \mathcal{N}_k(H')} w_j(H')\delta(Q - Q_j)$ (where weight w_j is proportional to the distance between H' and H_i, i.e. $w_j \propto \exp(-||H_j - H'||^2/\sigma^2)$).

(2) *Conditional VAE:* Introduce latent $z \in \mathbb{R}^d$: $z \sim \mathcal{N}(0, I)$, $Q \sim p_\theta(Q \mid H, z)$

- Train an encoder $q_\phi(z \mid Q, H)$ and decoder $p_\theta(Q \mid H, z)$ via ELBO loss:

$$\log p_\theta(Q \mid H) \geq \mathbb{E}_{z \sim q_\phi(z|Q,H)}[\log p_\theta(Q \mid H, z)] - \text{KL}[q_\theta(z \mid Q, H) \,||\, \mathcal{N}(0, I)]$$

- Potential decoder choice: output $\mu_\theta(H, z)$ and diagonal $\Sigma_\theta(H, z)$; use Gaussian likelihood:

$$\log p_\theta(Q \mid H, z) = \log \mathcal{N}(Q; \mu_\theta(H, z), \Sigma_\theta(H, z))$$

[TODO: Read into math more], notes in 6.2

- Advantages of cVAE for this problem:
  - Multi-modality via latent variable z (different z's can map to different valid configurations on the circle) → addresses mode collapse relative to deterministic nets [assuming we avoid posterior collapse (decoder ignores z)]
  - Smooth generalization in H: theoretically, the encoder/decoder can learn continuous maps in H-space, allowing better interpolation than the nearest neighbors approach.
- Note: θ is periodic, so training a Gaussian likelihood $p_\theta(Q \mid H, z)$ (in the decoder) on raw $\theta \in [0, 2\pi)$ is awkward because $\theta \approx 0$ and $\theta \approx 2\pi$ are close physically but far numerically ($\hat{\theta} = 2\pi - 0.01$ is a pretty good prediction for ground truth $\theta = 0.01$). To avoid this, instead of modeling $\theta \sim \mathcal{N}(\mu, \sigma^2)$, we can model a Gaussian over Cartesian coordinates in $\mathbb{R}^2$:

$$(\cos\theta, \sin\theta) \sim \mathcal{N}(\mu_{cs}, \text{diag}(\sigma_{cs}^2))$$

We can recover θ via atan2 at inference time.

(3) *Invertible NN* (FrEIA): [TODO: Read into math more], notes in 6.3

- **Representation:** To avoid periodicity issues in θ, we model:

$$Q_{\text{feat}} = (x, y, \cos\theta) \in \mathbb{R}^4, \quad H = (h_x, h_y) \in \mathbb{R}^2$$

- **Conditional invertible map:** We learn an invertible mapping (for each condition H)

$$f_\theta(\cdot; H) : \mathbb{R}^4 \leftrightarrow \mathbb{R}^4, \quad z = f_\theta(Q; H), \quad Q = f_\theta^{-1}(z; H)$$

with a simple base distribution $z \sim \mathcal{N}(0, I_4)$. In code this is implemented as a stack of conditional affine coupling blocks (FrEIA `SequenceINN` + `AllInOneBlock`) where each block receives H as a conditioning input.

- **Training:** Using the change-of-variables formula, the conditional density is:

$$p_\theta(Q \mid H) = p_Z(f_\theta(Q; H)) \left| \det \frac{\partial f_\theta(Q; H)}{\partial Q} \right|$$

Therefore:

$$\log p_\theta(Q \mid H) = \log p_Z(z) + \log |\det J_{f_\theta}(x; H)|$$

With $p_Z = \mathcal{N}(0, I)$, the per-example negative log-likelihood (dropping constants) is:

$$\mathcal{L}_{\text{NLL}}(Q, H) = \frac{1}{2}||z||^2 - \log |\det J_{f_\theta}(Q; H)|$$

We minimize $\mathbb{E}_{(Q,H)\sim\text{data}}|\mathcal{L}_{\text{NLL}}(Q, H)|$.

- **Sampling:** At test time, for a given H we sample:

$$z \sim \mathcal{N}(0, I_4), \qquad Q = f_\theta^{-1}(z, H)$$

then convert $Q = (x, y, \cos\theta, \sin\theta)$ back to (x, y, θ) with $\theta = \text{atan2}(\sin\theta, \cos\theta)$.

- **Optional FK regularization (sample-space constraint):** Since the true conditional support lies on the IK manifold (all physically valid configurations for a given target must satisfy the IK constraint), we can add a penalty that directly enforces that generated samples reach H:

$$\mathcal{L} = \mathbb{E}_{(Q,H)\sim\mathcal{D}}\left[\mathcal{L}_{\text{NLL}}(Q, H)\right] \; + \; \lambda_{\text{FK}}\, \mathbb{E}_{H\sim\mathcal{D},\, z\sim\mathcal{N}(0,I)}\left[\|f_{\text{FK}}(f_\theta^{-1}(z; H)) - H\|^2\right].$$

In our implementation we compute FK error on inverse samples $\hat{Q} = f_\theta^{-1}(z; H)$ during training (rather than on the training Q itself, as $\text{FK}(Q) = H$ by virtue of being in the dataset), so the regularizer shapes the *sampled* conditional distribution.

5.4. **Evaluation.** Assuming that the ground-truth $p^*(Q \mid H)$ is a uniform distribution over the feasible circle set, i.e.: $\theta \sim \text{Unif}[0, 2\pi) \to x = h_x - L\cos\theta$, $y = h_y - L\sin\theta$:

- Accuracy: $e_{\text{hand}}(Q, H) \coloneqq ||f(Q) - H||$
- Diversity/coverage: convert each sample to its implied angle on the circle:

$$\theta^{(s)}_{\text{implied}} \coloneqq \text{atan2}\left(h_y - y^{(s)}, h_x - x^{(s)}\right)$$

Check how well the empirical distribution over θ matches the target (here we assume $\theta \sim \text{Unif}[0, 2\pi)$): for histogram $\hat{p}(\theta)$, we can compute:
  - KL divergence to uniform: $D_{\text{KL}}(\hat{p}(\theta)\,||\,\text{Unif})$
  - Max angle gap: sort angles $\theta^{(s)}$ around the circle, compute max gap $\Delta_{\max} = \max_i(\theta_{i+1} - \theta_i)$
    Large $\Delta_{\max}$ = model is missing big arcs (collapse)
- "Uses latent" test (for cVAE/cINN): fix H, sample many z's, then measure output variance $\text{Var}(Q \mid H)$. If the model ignores z, this variance will be small.
  [↓ TODO: Implement]
- Generalization in H: make a test set where H is (1) in-distribution, (2) near boundary, (3) out-of-distribution (slightly outside training workspace). Compare success and coverage!
- Inference speed per sample

5.5. **Extending to more general cases.** Ideally, would be nice if this toy problem gave us an idea of the advantages/disadvantages of different approaches in regards to the following challenges:

- *Mode collapse:* Test: fixed H, sample 1k solultions, compute $\Delta_{\max}$, KL-to-uniform, etc.
- *Generalization:* Test: distribution of $e_{\text{hand}} = ||f(Q) - H||$
- *Bias from data collection* (forward vs. inverse sampling):

5.6. **Potential next-step extensions.**

- Disconnected feasible sets: modify toy with a "visibility" constraint such as
$$\text{visible}(Q, H) = 1 \iff \theta \in [\alpha_1, \beta_1] \cup [\alpha_2, \beta_2]$$
Now the true $p^*(Q \mid H)$ is two separated modes. **Test:** cluster sampled angles into arcs and compute mode recall: fraction of samples that land in each arc.
- Scalability with dimension: add DoF to the toy (e.g., 2-link arm, variable stick length). **Test:** track how coverage/error degrades with dimension.

6. Extra notes

6.1. **Mode collapse.** A generative model has mode collapse if, for a fixed condition H, the samples $Q \sim \hat{p}(Q \mid H)$ cover only a small subset of the true support of $p^*(Q \mid H)$.

- In this toy problem, the support is the full circle parameterized by θ. Mode collapse looks like the model always outputting $\theta \approx 0$ (i.e. stands in one favorite location) or outputting only a few discrete angles, ignoring the rest the circle, etc.

6.2. **Conditional VAE theory.** The goal is to model a multi-modal conditional distribution over robot configurations Q given target H: $p^*(Q \mid H)$. In our simple robot setup, the true conditional has a 1D manifold of solutions (a circle in (x, y) paired with corresponding θ), so it's not unimodal in the usual sense (rather than being a single blob in $\mathbb{R}^d$, for fixed H, probability mass $p^*(Q \mid H)$ is concentrated along a ring in (x, y) with a corresponding orientation θ at each point).

6.2.1. *Motivation.* Because $p^*(Q \mid H)$ is not unimodal, if we model $p(Q \mid H)$ as a single Gaussian/predict a single mean, we face mean collapse: the mean of points around a circle is the center of the circle, but the center is invalid. So the "best" unimodal prediction is often physically invalid/low-probability under the true distribution.

6.2.2. *Model architecture.* We introduce latent $z \in \mathbb{R}^{d_z}$, which ideally will be the variable that "chooses" *which* solution the model means among the many possible solutions on the ring for a given H. Define a prior $p(z) = \mathcal{N}(0, I)$. There are two parts to the model (decoder and encoder):

- Decoder (generative model): The decoder is a conditional distribution $p_\theta(Q \mid H, z)$. For the architecture of the decoder, we use the diagonal Gaussian (potential next step – try other architectures):
$$p_\theta(Q \mid H, z) = \mathcal{N}(Q; \mu_\theta(H, z), \text{diag}(\sigma_\theta^2(H, z)))$$
where $\mu_\theta(H, z)$ and σ_θ^2 are the predicted mean/variance of each coordinate of Q.
- *Interlude:* $p_\theta(Q, z \mid H) = p(z \mid H)p(Q \mid z, H) = p(z)p_\theta(Q \mid H, z)$ [assuming $p(z \mid H) = p(z)$]. Bayes' rule gives:
$$p_\theta(z \mid Q, H) = \frac{p_\theta(Q, z \mid H)}{p_\theta(Q \mid H)} = \frac{p(z)p_\theta(Q \mid H, z)}{p_\theta(Q \mid H)}$$
Expanding the denominator: $p_\theta(Q \mid H) = \int p(z')p_\theta(Q \mid H, z')\, dz'$ (consider all possible latent explanations z', weight how likely each would produce Q, then sum them up). Plugging this in,
$$p_\theta(z \mid Q, H) = \frac{p(z)p_\theta(Q \mid H, z)}{\int p(z')p_\theta(Q \mid H, z')\, dz'}$$
The denominator $\int p(z')p_\theta(Q \mid H, z')\, dz'$ is intractable: $p_\theta(Q \mid H, z') = \mathcal{N}(Q; \mu_\theta(H, z'), \Sigma_\theta(H, z'))$, so we effectively have
$$p(z')p_\theta(Q \mid H, z') \Rightarrow \mathcal{N}(z'; 0, 1) \times \mathcal{N}(Q; \mu_\theta(H, z'), \Sigma_\theta(H, z'))$$
For fixed Q and H, $\mathcal{N}(Q; \mu_\theta(H, z'), \Sigma_\theta(H, z'))$ becomes some complicated nonnegative function of $z' \to g(z')$. Then,
$$p_\theta(Q \mid H) = \int \mathcal{N}(z'; 0, I)g(z')\, dz'$$
This is very hard to compute analytically: recall the multivariate Gaussian density is given by:
$$f_{\mu,\Sigma}(Q) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\tfrac{1}{2}(Q - \mu)^\top \Sigma^{-1}(Q - \mu)\right)$$

Here, we plug in $\mu = \mu_\theta(H, z'), \Sigma = \Sigma_\theta(H, z')$. These are neural-net outputs, i.e. complicated nonlinear functions of z', which makes the integral difficult to compute.

Since the true posterior is intractable, introduce a tractable approximation $q_\phi(z \mid Q, H) : (Q, H) \mapsto p(z)$ which essentially answers "If this Q came from some latent z, what must that z have been?"

- Encoder (inference model): The encoder is the conditional distribution $q_\phi(z \mid Q, H)$.

6.3. **FrEIA INN theory.** The goal is to learn $p(Q \mid H)$, where $Q = (x, y, \cos\theta, \sin\theta) \in \mathbb{R}^4$ (equivalent form of (x, y, θ)) and $H = (h_x, h_y) \in \mathbb{R}^2$. A normalizing flow models an invertible map (for each fixed condition h):

$$f_\theta(\cdot; h) : \mathbb{R}^4 \to \mathbb{R}^4, \quad z = f_\theta(q; h), \quad q = f_\theta^{-1}(z; h)$$

We pick the base density for z to be the multivariate standard normal:

$$p_Z(z) = \mathcal{N}(0, I_4)$$

Then, $p_\theta(q \mid h)$ can be defined via "push q through f to get z, and measure how likely that z is, corrected by volume change" $\to$ correction is the Jacobian determinant.

6.3.1. *Hand-wavy intuition.* For each condition h, the true data q lives in some weird, potentially multi-modal shape (ring of solutions in our simplified problem). A Gaussian cannot represent that shape directly.

- *Core idea:* Instead of trying to fit a complicated distribution directly, we learn a warp of space that makes the complicated shape look Gaussian.
  - Forward direction: $z = f_\theta(q; h)$ takes real samples q and maps them into "Gaussian space"
  - Training enforces: "after mapping, these z's should look like samples from $\mathcal{N}(0, I)$"
  - Warping space changes volume $\to$ probability densities change:
    - If f expands a region of q-space, then points there become less dense in q-space.
    - If f compresses a region, points become more dense.

After learning an invertible NN model, we can generate realistic q by:

(1) Sample $z \sim \mathcal{N}(0, I)$
(2) Invert: $q = f_\theta^{-1}(z; h)$

Fun way of thinking about it: Imagine data for q's is shaped like a bent pretzel in $\mathbb{R}^D$. A flow learns a continuous, invertible deformation of space that morphs the pretzel into a D-dim. Gaussian blob. Forward pass = "unbend the pretzel into a Gaussian." Inverse pass = "bend a Gaussian blob back into a pretzel." The Jacobian term is bookkeeping of how the deformation stretches space.

6.3.2. *Why layers need to be invertible.*

(1) Invertible layers allow flow models to compute an exact probability for any observed training point: if Q is a real solution from the dataset for some H, we can compute exactly how likely the model thinks it is (flows are the only model out of GANs, VAEs, and diffusion models that can compute the exact log-likelihood of a new sample)

$$\Pr(\mathbf{f}(\mathbf{Q}) \mid \mathbf{H}, \phi) = \left| \frac{\partial \mathbf{f}(\mathbf{z}, \mathbf{H}, \phi)}{\partial \mathbf{z}} \right|^{-1} \cdot \Pr(\mathbf{z}) \text{ where } \mathbf{z} = \mathbf{f}^{-1}(\mathbf{x}, \mathbf{H}, \phi) \text{ and } \phi = \text{learned model params}$$

(2) Invertibility allows sampling to be one inverse pass rather than iterative denoising, sampling chains, etc.: once trained, we can directly (1) sample $z \sim \mathcal{N}(0, I)$ (2) compute $Q = f_\theta^{-1}(z; H)$

6.3.3. *More concrete math.* In separate PDF (`flow model FrEIA implementation notes.pdf`).

6.4. **Normalizing flow.** In separate PDF (`normalizing flow notes.pdf`).

6.5. **Diffusion models.**

6.6. **Conceptual comparison of different methods.** Given the data is shaped like a pretzel in $\mathbb{R}^D$ ...

6.6.1. *Flow.* Learn an invertible deformation conditioned on H that straightens the pretzel into a Gaussian $\mathcal{N}(0, I_D)$. Sampling works by running the inverse deformation on $z \sim \mathcal{N}(0, I_D)$ and H to obtain Q^*.

6.6.2. *cVAE.* Learn a conditional generator that uses Gaussian knobs to choose how to output the pretzel; training teaches the knobs to stay Gaussian while reconstructing data.

6.6.3. *Diffusion.* Instead of unbending the pretzel into a Gaussian in one shot, diffusion dissolves the pretzel into pure noise by gradually adding noise, then learns to sculpt noise back into the pretzel step-by-step.

6.7. **Invertible neural networks vs. normalizing flow.** Invertible neural networks are a class of deep networks that approximate bijective functions and are characterized by a forward mapping that can be inverted. All normalizing flow networks are invertible neural networks.

6.7.1. *INNs as generative models.* Because INNs learn a transformation between the input data distribution and a simple prior distribution (typically a normal distribution), the inverse mapping automatically acts as a generator, converting the simple distribution into samples resembling the input data.

6.7.2. *INNs that are not normalizing flow models.* Invertible neural networks that are not flow models exist (TODO: Read into this paper [1]).

References

[1] Lynton Ardizzone, Jakob Kruse, Sebastian J. Wirkert, Daniel Rahner, Eric W. Pellegrini, Ralf S. Klessen, Lena Maier-Hein, Carsten Rother, and Ullrich Köthe. Analyzing inverse problems with invertible neural networks. *CoRR*, abs/1808.04730, 2018.