ECEN-524 Robot Learning: Human Demonstrations

Trajectory Learning and Adaptation from Human Demonstration

Anthony Angeles
Santa Clara University

February 2, 2026

Abstract

In this report, I detail the implementation of robot learning from human demonstration for a pick-and-place task. Using a dataset of eleven demonstrated trajectories, I explore methods for trajectory segmentation, motion learning, and generalization. I utilize Dynamic Movement Primitives (DMPs) to encode motion segments and demonstrate their adaptability to new start and goal configurations. I also analyze trajectory similarity using Dynamic Time Warping (DTW) and implement probabilistic modeling via Gaussian Mixture Models (GMM) and Gaussian Mixture Regression (GMR). Finally, I demonstrate online adaptation to novel environments by integrating potential field-based obstacle avoidance into the DMP framework.

1 Introduction

Throughout the first portion of ECEN 524 - Robot Learning, I have explored topics and methods for robot learning. Most recently, the class has focused on learning from human demonstration to help a robot manipulator learn trajectories for a pick-and-place task. This approach is invaluable because it allows robots to replicate complicated tasks without requiring large quantities of data to learn them from scratch.

For this project, the class demonstrated the same pick-and-place task eleven times while recording the robot movement data. One challenge we encountered was that the data does not include end-effector actuation data; the robot model and end-effector do not communicate well over ROS2 (Robot Operating System 2). Despite this limitation, our demonstrations captured XYZ position and quaternion orientation data, which proved sufficient for the learning methods that was detailed to be applied.

2 Data Visualization and Preprocessing

Before implementing any learning algorithms, it was important to plot the raw data and understand what it shows in relation to the demonstrations the class recorded. In Figure 1, I present the complete trajectories from both a 3D perspective and a top-down view. The data shows that while there is variety in the trajectories themselves, they still share the same core pick and placement positions relative to the task demonstrated.

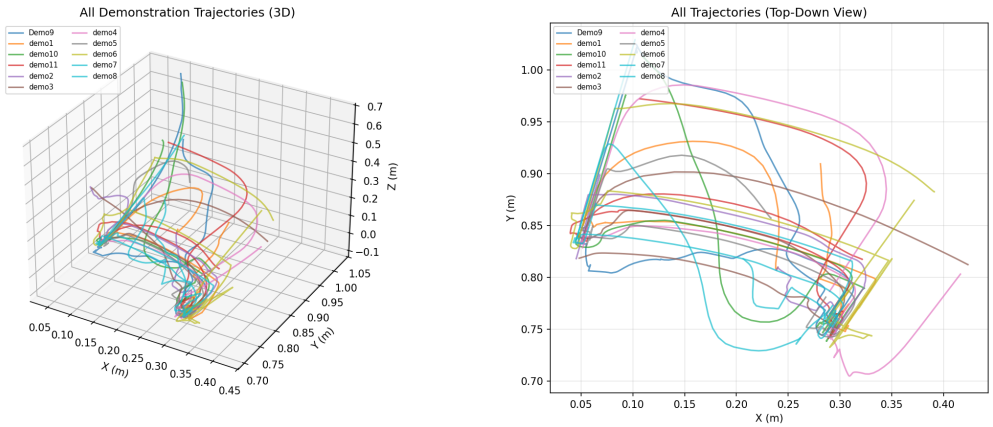


Figure 1: All trajectories plotted in 3D and top-down.

When viewing the raw data from a 3D perspective or top-down view, I realized it does not clearly reveal the smaller differences between demonstrations. Since the data points are closely clumped together, I decided to separate out the specific data I wanted to analyze. I prioritized the positional data (XYZ). In Figure 2, I am better able to see the differences between demonstrations and can more clearly pinpoint the moments of actuation and overall movement taken in the robot demonstration.

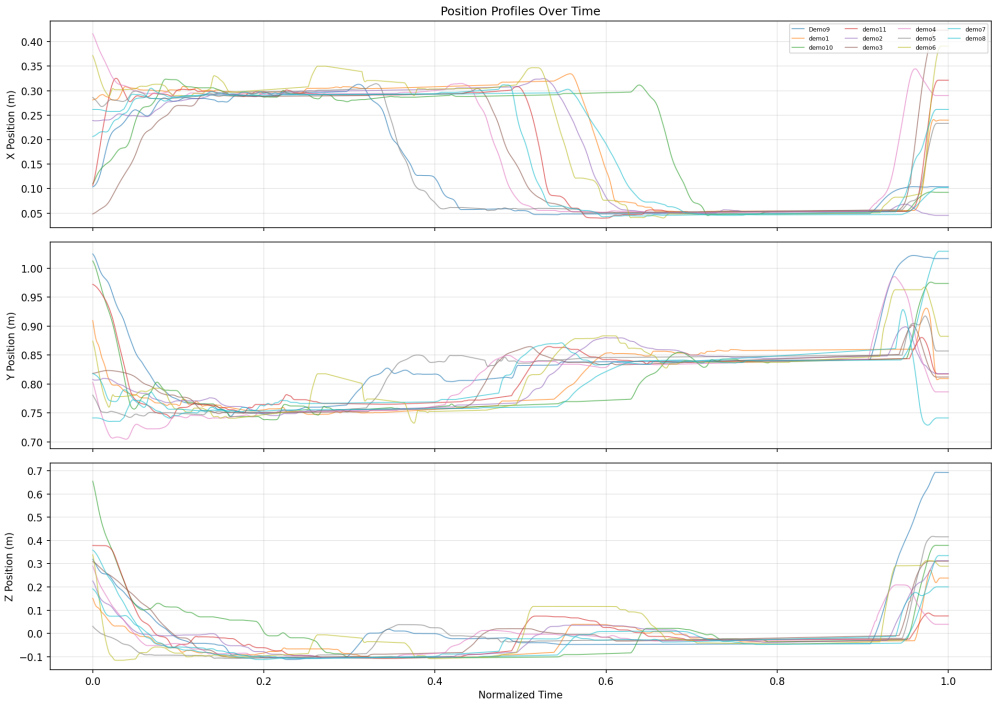


Figure 2: XYZ Position profiles of all trajectories.

After plotting the raw XYZ positional data, I moved forward with identifying points of interest. Ideally, I would have simply gathered the actuation data for when the end-effector opened and closed; yet as I noted earlier, that was not possible with my data. Instead, I determined points of interest by identifying Z-minima and low velocity points. In Figure 3, I display what this analysis

looks like for demonstration nine. The idea behind this is that while we can use pauses in timestamps in our recorded data to identify point of interests, that could cause multiple false positives in actuation detection; so we look for two lowest points where it clearly slowed down to actuate.

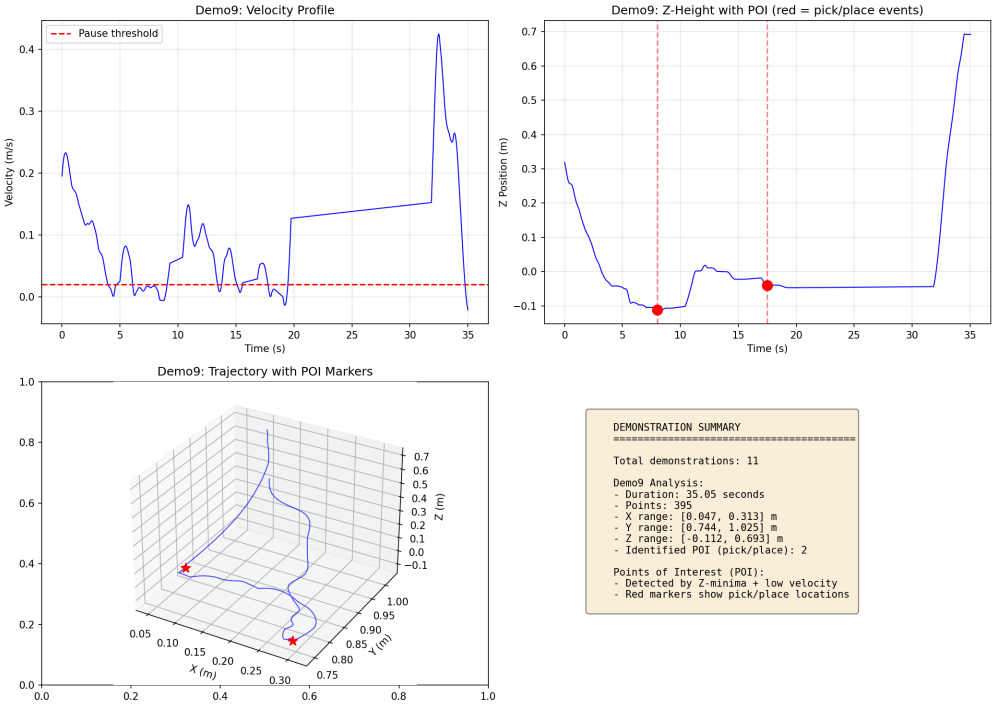


Figure 3: Points of interest (Z-minima and stops) visualized on Demonstration Nine.

3 Dynamic Movement Primitives

3.1 Methodology and Relevance

DMPs encode trajectories as a dynamical system that can be trained from a single demonstration and then generalized to new start and goal positions without retraining. I first segmented each demonstration into individual pick-to-place motions, then trained a separate DMP on each segment, and finally tested generalization by modifying start and goal positions.

I found DMPs highly relevant for robotics because they provide a compact representation of motion that naturally handles timing variations and can adapt to new task configurations. Unlike simple trajectory playback, DMPs guarantee convergence to the goal position and can be executed in real-time with perturbations. In the larger context of robot learning, this means that when a new task is presented to the robot, instead of requiring large amounts of new training data, I can use the DMP methodology to allow the robot to execute on this new task configuration.

3.2 Experimental Results

I extracted 8 valid pick-and-place segments across all demonstrations and trained DMPs on each. The average reproduction error was approximately 147mm, indicating that the DMPs successfully captured the essential shape of the trajectories. My generalization tests demonstrated that shifting the start and goal positions by up to 15cm produced smooth, dynamically consistent trajectories

that maintained the learned motion profile. This confirms that DMPs separate the "what" (motion shape) from the "where" (spatial configuration).

3.3 Implementation Details

Libraries Used:

- `pydmps` (DMPs_discrete): Open-source DMP implementation for discrete movements (`https://github.com/studywolf/pydmps`).
- `SciPy` (interp1d): Linear interpolation for trajectory resampling (`https://scipy.org/`).
- Custom Algorithm: Segment Extraction.

With this library, I selected and trained DMPs on the best data I found. Figure 4 shows the training results, and Figure 5 visualizes the generalization capabilities.

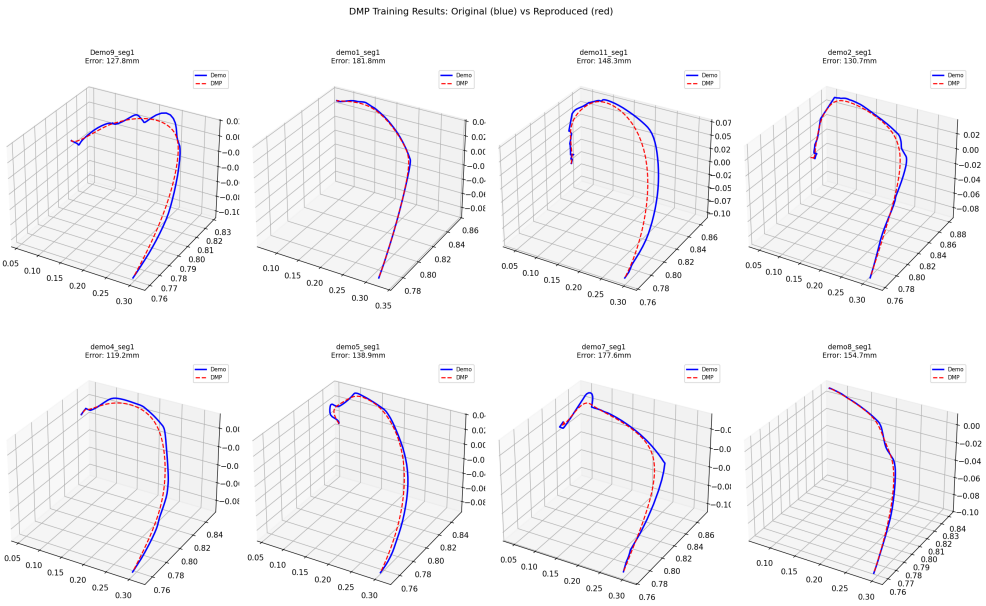
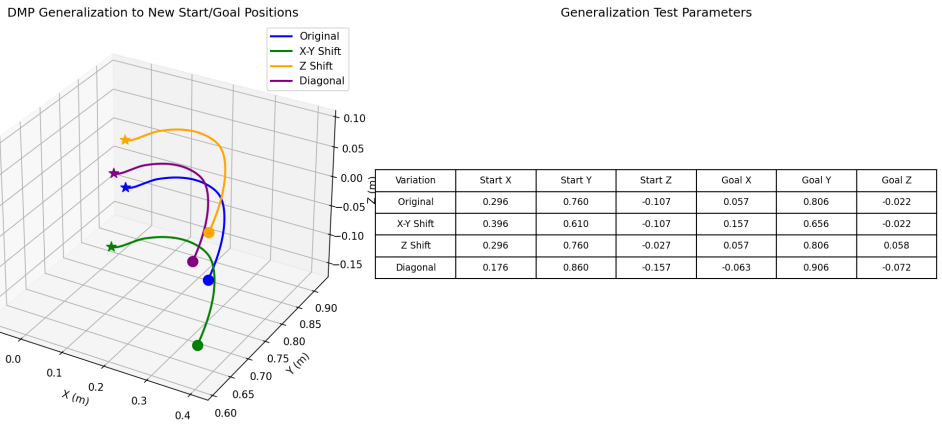


Figure 4: Training results from the DMPs.



Variation	Start X	Start Y	Start Z	Goal X	Goal Y	Goal Z
Original	0.296	0.760	-0.107	0.057	0.806	-0.022
X-Y Shift	0.396	0.610	-0.107	0.157	0.656	-0.022
Z Shift	0.296	0.760	-0.027	0.057	0.806	0.058
Diagonal	0.176	0.860	-0.157	-0.063	0.906	-0.072

Figure 5: Generalization of the DMP to new targets.

3.3.1 Algorithm: Segment Identification

I developed the core idea of the algorithm used, while using Claude Sonnet 4.5 to refine the results of the algorithm that identifies pauses in the demonstration where the robot stopped (time gap > 0.5s) and position change is minimal (< 15mm). These pauses mark the boundaries of pick-place motions. Between consecutive pauses, I extract segments and validate them using three criteria: sufficient travel distance (> 15cm), adequate duration (> 3s), and meaningful Z-height variation (> 3cm, confirming vertical motion for picking or placing).

```
def identify_pick_place_segments(trajectory, times):
    positions = trajectory[:, :3]
    time_diffs = np.diff(times)

    # Find significant pauses (time gaps > 0.5 seconds with minimal movement)
    pause_threshold = 0.5
    position_tolerance = 0.015

    pauses = []
    for i in range(len(time_diffs)):
        if time_diffs[i] > pause_threshold:
            pos_change = np.linalg.norm(positions[i+1] - positions[i])
            if pos_change < position_tolerance:
                pauses.append({'index': i, 'time': times[i], 'position': positions
    [i]})

    # Extract segments between consecutive pauses
    segments = []
    for i in range(len(pauses) - 1):
        start_idx = pauses[i]['index']
        end_idx = pauses[i + 1]['index']

        distance = np.linalg.norm(positions[end_idx] - positions[start_idx])
        duration = times[end_idx] - times[start_idx]
        z_range = np.max(positions[start_idx:end_idx+1, 2]) - np.min(positions[
    start_idx:end_idx+1, 2])

        # Only keep meaningful pick-place segments
        if distance > 0.15 and duration > 3.0 and z_range > 0.03:
            segments.append({'trajectory': trajectory[start_idx:end_idx+1], ...})

    return segments
```

Listing 1: Identifying Pick and Place Segments

3.3.2 Algorithm: DMP Training

I found that raw demonstration data often contains irregular timestamps due to sensor delays or pauses. Large gaps would cause artificially high velocities when computing derivatives. To address this, I "clean" the timestamps by replacing gaps larger than 150ms with a nominal 50ms step. Then I resample the trajectory to uniform timesteps (10ms) using linear interpolation, which DMPs require for consistent learning. Finally, I train a DMP with 50 Gaussian basis functions per dimension—this number provides sufficient expressiveness to capture complex motion shapes while avoiding overfitting.

```
def train_dmp_on_segment(segment, n_bfs=50):
    trajectory = segment['trajectory'][:, :3].copy()
```

```
    times = segment['times'].copy()

    # Remove large time gaps that would distort velocity
    gap_threshold = 0.15
    cleaned_times = np.zeros_like(times)
    cleaned_times[0] = 0
    nominal_dt = 0.05

    for i in range(len(np.diff(times))):
        if np.diff(times)[i] > gap_threshold:
            cleaned_times[i + 1] = cleaned_times[i] + nominal_dt
        else:
            cleaned_times[i + 1] = cleaned_times[i] + np.diff(times)[i]

    # Resample to uniform timesteps
    dt = 0.01
    n_timesteps = max(int(cleaned_times[-1] / dt), 100)
    uniform_times = np.linspace(0, cleaned_times[-1], n_timesteps)
    uniform_positions = np.zeros((n_timesteps, 3))

    for i in range(3):
        interp_func = interp1d(cleaned_times, trajectory[:, i], kind='linear')
        uniform_positions[:, i] = interp_func(uniform_times)

    # Train DMP with 50 basis functions per dimension
    dmp = DMPs_discrete(n_dmps=3, n_bfs=n_bfs, dt=dt)
    dmp.imitate_path(y_des=uniform_positions.T)

    return dmp, n_timesteps, cleaned_times[-1]
```

Listing 2: Training DMP on Segment

4 Trajectory Alignment

4.1 Methodology and Relevance

Unlike Euclidean distance, which requires aligned sequences, DTW finds the optimal temporal alignment between two sequences that may differ in speed or duration. I computed DTW distances for all unique pairs of demonstrations and created a similarity matrix to identify which demonstrations are most alike.

From this, I learned that DTW is particularly relevant because human demonstrations naturally vary in execution speed, a demonstrator might move faster or pause at different times across repetitions. DTW accounts for these temporal variations by warping the time axis to find correspondences. Identifying similar demonstrations helps me understand demonstration quality and informs which demonstrations to combine or exclude from learning.

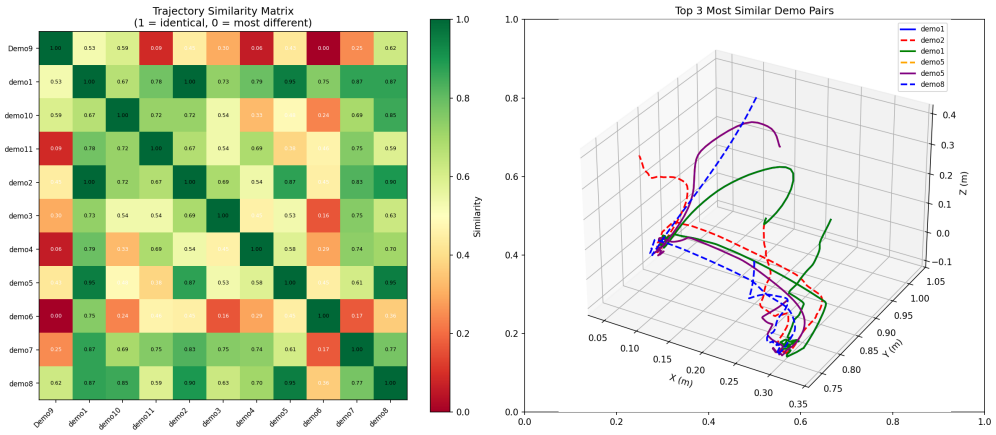


Figure 6: DTW Similarity Matrix.

4.2 Experimental Results

The similarity matrix revealed that demo1 and demo2 were the most similar pair (similarity score of 1.000 after normalization), indicating these were likely performed most consistently. The visualization shows clusters of similar demonstrations and highlights outliers that differ significantly from the majority. These results can guide demonstration selection where highly similar demonstrations reinforce learning, while outliers might indicate errors or intentional variations. Another example of similarity can be seen between demo5 and demo8 (similarity score of 0.95 after normalization)

4.3 Implementation Details

Libraries Used:

- `SciPy` (cdist): Efficient pairwise Euclidean distance computation. (`https://scipy.org/`)
- `dtw` (optional): DTW library (falls back to custom implementation if unavailable). (`https://pypi.org/project/dtw-python/`)

4.3.1 Algorithm: Dynamic Time Warping

With dynamic programming and the help of Claude Sonnet 4.5, I found the minimum-cost alignment between two sequences. I first compute a distance matrix where entry (i, j) is the Euclidean distance between point i of trajectory 1 and point j of trajectory 2. Then I fill a cumulative cost matrix where each cell represents the minimum total cost to align the sequences up to that point. The final cell contains the total alignment cost, which I normalize by path length for fair comparison between sequences of different lengths.

```
def compute_dtw(traj1, traj2):
    # Compute pairwise Euclidean distance matrix
    dist_matrix = cdist(traj1, traj2, metric='euclidean')

    n, m = dist_matrix.shape
    # Initialize DTW accumulation matrix with infinity
    dtw_matrix = np.full((n + 1, m + 1), np.inf)
    dtw_matrix[0, 0] = 0

    # Dynamic programming: fill matrix
```

```
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            cost = dist_matrix[i-1, j-1]
            dtw_matrix[i, j] = cost + min(
                dtw_matrix[i-1, j],      # Insertion (advance traj1 only)
                dtw_matrix[i, j-1],      # Deletion (advance traj2 only)
                dtw_matrix[i-1, j-1]     # Match (advance both)
            )

    # Final DTW distance
    distance = dtw_matrix[n, m]
    path_length = n + m  # Normalization factor

    return distance / path_length  # Normalized distance
```

Listing 3: DTW Calculation

5 Gaussian Mixture Model/Gaussian Mixture Regression

5.1 Methodology and Relevance

Using Gaussian Mixture Models (GMM) to model the joint distribution of time and position, then applying Gaussian Mixture Regression (GMR) to predict positions conditioned on time. I determined the optimal number of Gaussian components using the Bayesian Information Criterion (BIC); which using BIC was the recommended method by our professor.

GMM/GMR is highly relevant for learning from multiple demonstrations, not only does it naturally handle variability rather than memorizing a single trajectory, the GMM captures where demonstrations agree (low variance) and where they differ (high variance). GMR produces a trajectory that respects this variance, following the consensus where demonstrations align and smoothly interpolating through variable regions.

5.2 Experimental Results

The BIC analysis determined that 14 Gaussian components optimally balance model complexity and fit quality. The GMR-generated trajectory achieved an average reproduction error of approximately 63mm across all demonstrations which is significantly better than individual DMPs (147mm). This improvement occurs because GMR implicitly averages multiple demonstrations, canceling out individual noise and variations. The GMM component visualization (Figure 7) shows how Gaussians are distributed along the trajectory, with higher density in regions where all demonstrations pass.

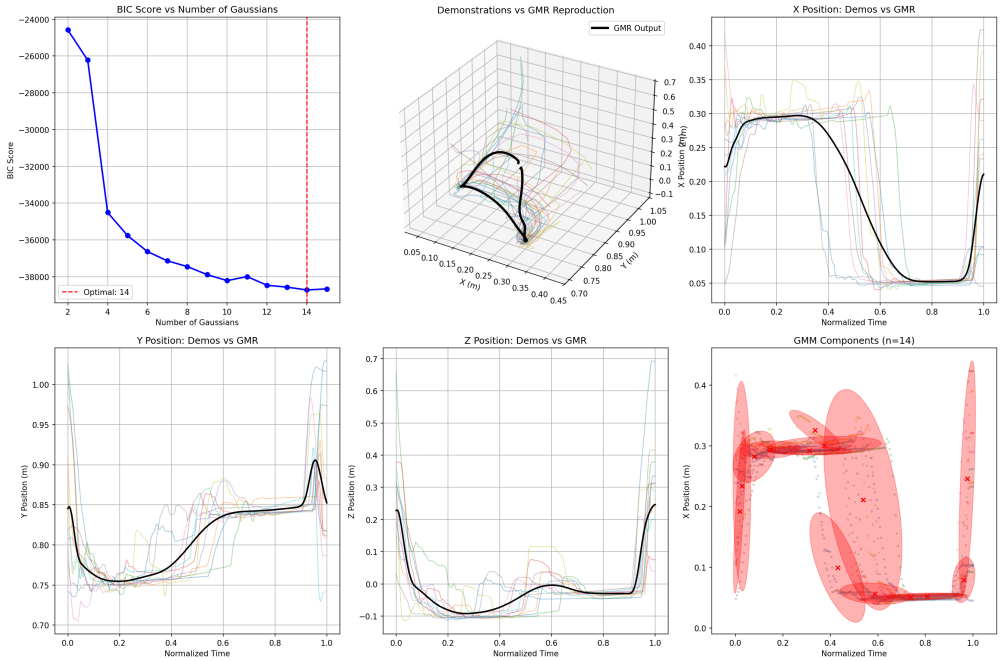


Figure 7: GMM and GMR Results.

5.3 Implementation Details

Libraries Used:

- `scikit-learn` (GaussianMixture): GMM fitting with expectation-maximization. (`https://scikit-learn.org/`)
- `NumPy`: Matrix operations for GMR computation.

5.3.1 Algorithm: GMR Prediction

For each query time point: (1) I compute the "responsibility" of each Gaussian—how likely that Gaussian generated this time value, weighted by mixture weights. (2) For each Gaussian, I compute the conditional mean of the output (position) given the input (time) using the standard formula for Gaussian conditioning. (3) The final prediction is a weighted average of these conditional means, where weights are the responsibilities.

```
def gmr_predict(gmm, input_data, input_dims, output_dims):
    """Predict output dimensions conditioned on input dimensions."""
    n_samples = input_data.shape[0]
    n_output = len(output_dims)
    predicted_mean = np.zeros((n_samples, n_output))

    means = gmm.means_
    covars = gmm.covariances_
    weights = gmm.weights_
    n_components = len(weights)

    for i in range(n_samples):
        x_in = input_data[i]  # Query time point

```

```
        # Step 1: Compute responsibility of each Gaussian for this input
        h = np.zeros(n_components)
        for k in range(n_components):
            mu_in = means[k, input_dims]  # Mean of input dimension
            sigma_in = covars[k][np.ix_(input_dims, input_dims)]  # Input
    covariance

            # Gaussian probability density
            diff = x_in - mu_in
            sigma_in_inv = np.linalg.inv(sigma_in)
            exp_term = -0.5 * diff @ sigma_in_inv @ diff
            det = np.linalg.det(sigma_in)
            h[k] = weights[k] * np.exp(exp_term) / np.sqrt((2*np.pi)**len(
    input_dims) * det)

        h = h / np.sum(h)  # Normalize to get responsibilities

        # Step 2: Compute conditional mean from each Gaussian
        mu_out = np.zeros(n_output)
        for k in range(n_components):
            mu_in_k = means[k, input_dims]
            mu_out_k = means[k, output_dims]
            sigma_in_k = covars[k][np.ix_(input_dims, input_dims)]
            sigma_oi_k = covars[k][np.ix_(output_dims, input_dims)]  # Cross-
    covariance

            # Conditional Gaussian mean formula
            sigma_in_inv = np.linalg.inv(sigma_in_k)
            cond_mean = mu_out_k + sigma_oi_k @ sigma_in_inv @ (x_in - mu_in_k)

            mu_out += h[k] * cond_mean  # Weighted sum

        predicted_mean[i] = mu_out

    return predicted_mean
```

Listing 4: GMR Prediction

5.4 Piecewise vs Complete Trajectory Comparison

5.4.1 Methodology

Should I preserve the full motion structure when training GMM/GMR, or break it into smaller segments? This was a question presented to us by our professor in this project.

To find out, we were guided to test two approaches. For the **complete trajectory approach**, I resampled all 11 demonstrations to 200 points each, giving me 2,200 samples across 4 dimensions (time + XYZ). For the **piecewise approach**, I segmented the demonstrations into pick-and-place phases (8 segments total), resampled each to 100 points, and ended up with 800 samples. I trained a single GMM on each dataset and used BIC to select the optimal number of Gaussian components for both.

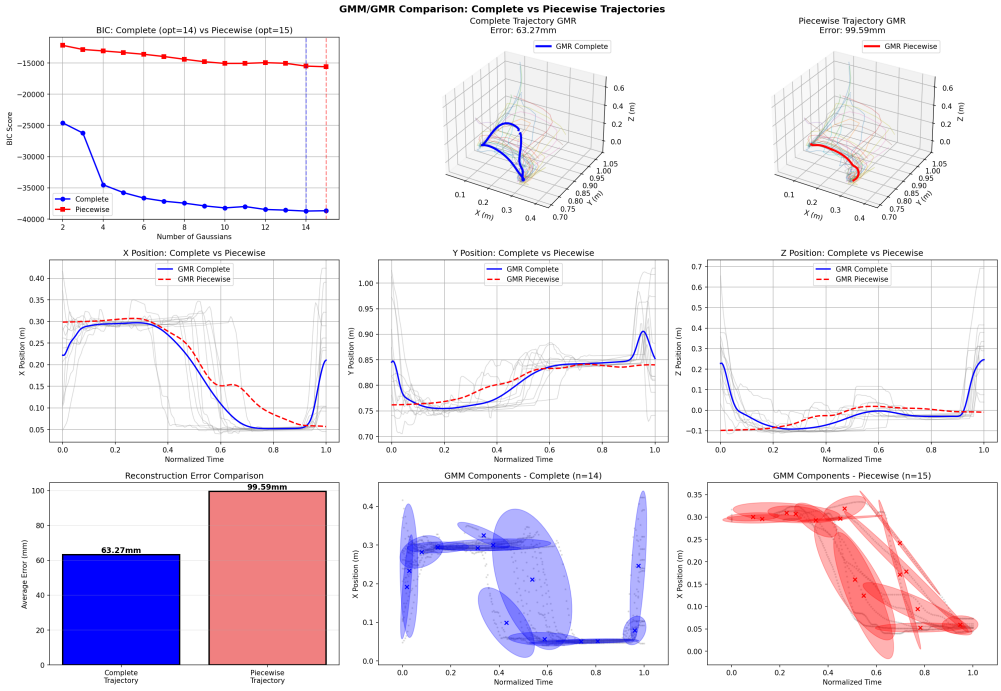


Figure 8: GMM and GMR Comparisons.

5.4.2 Results

The complete trajectory approach performed significantly better:

Metric	Complete	Piecewise
Optimal Gaussians	14	15
Average Error	63.27 mm	99.59 mm
Training Samples	2,200	800
BIC Score	-38,717.20	-15,600.02

Table 1: Complete versus piecewise trajectory GMM/GMR performance.

The complete approach reduced error by 36.32mm (a 57% improvement).

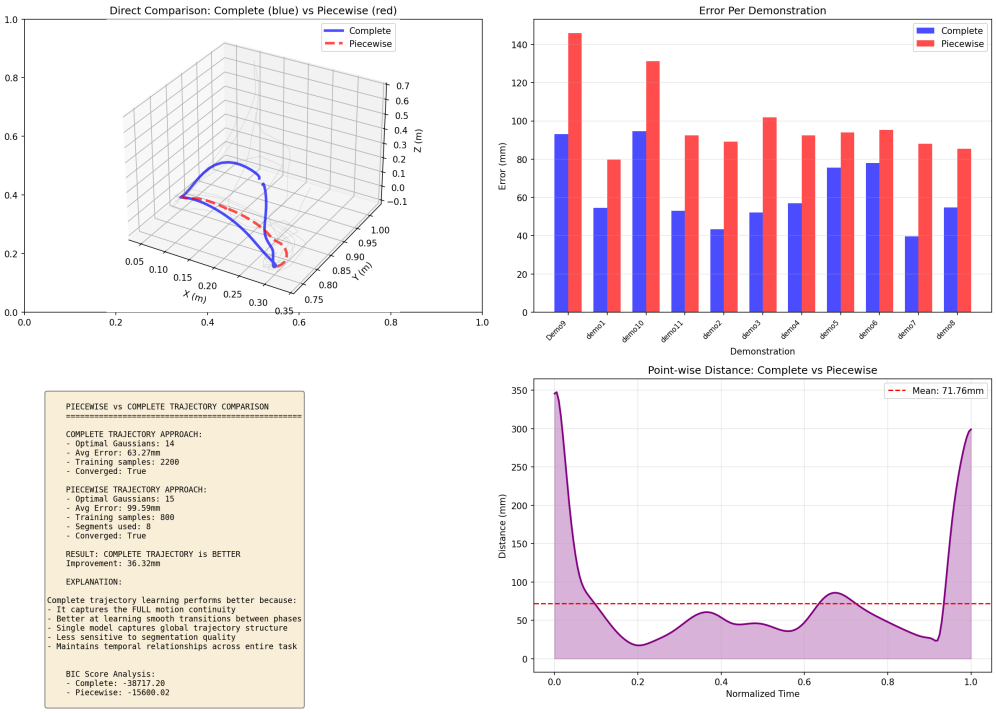


Figure 9: Per-demonstration error comparison between complete and piecewise approaches.

5.4.3 Takeaway

For pick-and-place tasks, it seems like the recommend method is using complete trajectories. The 57% error reduction shows that maintaining motion continuity is critical for accurate reproduction. That said, piecewise approaches might still make sense for tasks with truly independent phases or when you need to adapt individual segments separately.

6 Adaptation to New Environments

6.1 Methodology and Relevance

In this section Claude Sonnet 4.5 aided me the most in helping put the theory I had in my head into application, I implemented a potential field approach where obstacles exert virtual repulsive forces that push the trajectory away. I trained a DMP on a sample trajectory, placed three spherical obstacles along its path, and compared the original trajectory with the obstacle-avoiding version. I found that obstacle avoidance is critical for deploying learned motions in real environments where conditions change. Potential fields provide a reactive, computationally lightweight solution that integrates naturally with the DMP's dynamical system formulation where I simply add the repulsive force as a perturbation to the system state.

6.2 Experimental Results

The obstacle avoidance system I used improved minimum clearance by approximately 89mm compared to the unmodified trajectory. The trajectory smoothly curves around obstacles while still reaching the goal position with minimal final error. The clearance-over-time plot (Figure 10) shows that the modified trajectory maintains greater distance from obstacles throughout execution.

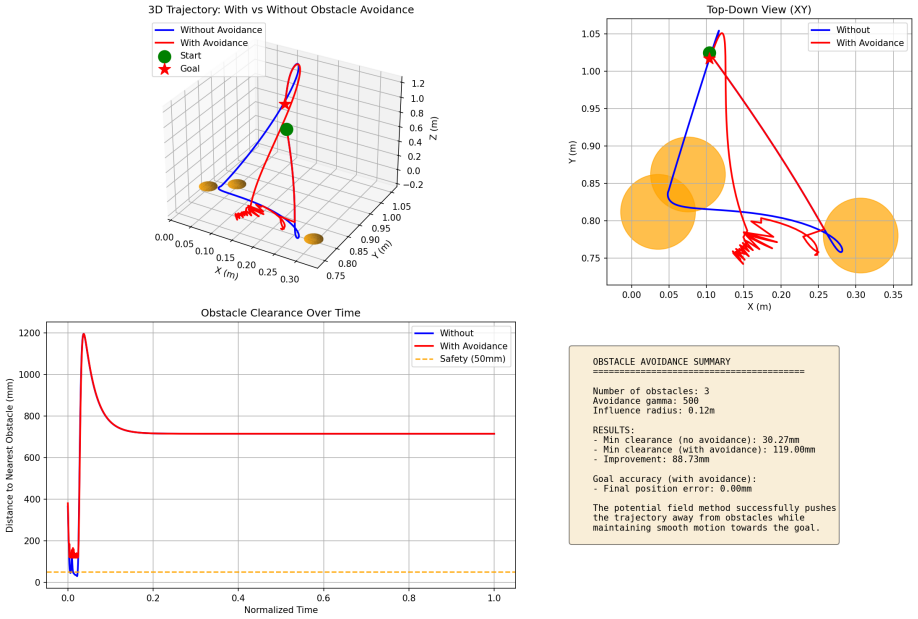


Figure 10: Obstacle Avoidance Results.

6.3 Implementation Details

Libraries Used:

- `pydmps` (DMPs_discrete): DMP implementation for trajectory generation.
- `NumPy`: Vector operations for force computation.

6.3.1 Algorithm: Potential Field Force

How it works: Each obstacle generates a repulsive force that pushes the robot away. I designed the force with two components:

1. **Direction:** A unit vector pointing from the obstacle toward the robot position, ensuring the force pushes "away."
2. **Magnitude:** I compute the magnitude of the repulsive force F as:

$$F = \gamma \cdot \frac{1}{d^2} \cdot \left(1 - \frac{d}{r}\right)^2 \tag{1}$$

where d is the distance to the obstacle, r is the influence radius, and γ is a tuning parameter (e.g., $\gamma = 500$).

The term $\frac{1}{d^2}$ creates strong repulsion when the robot is very close. The term $\left(1 - \frac{d}{r}\right)^2$ ensures the force smoothly decays to zero at the influence boundary. I sum forces from multiple obstacles. During DMP execution, I add this force as a position perturbation at each timestep:

$$y_{\text{dmp}} = y_{\text{dmp}} + F \cdot \Delta t^2$$

```python
class ObstacleAvoidance:
    """Potential field obstacle avoidance."""

    def __init__(self, obstacles, gamma=500, influence_radius=0.12):
        self.obstacles = np.array(obstacles)
        self.gamma = gamma                # Force strength multiplier
        self.influence_radius = influence_radius  # Range of influence (0.12m)

    def compute_force(self, position, velocity):
        force = np.zeros(3)

        for obs in self.obstacles:
            diff = position - obs           # Vector pointing away from obstacle
            dist = np.linalg.norm(diff)     # Distance to obstacle

            if dist < self.influence_radius and dist > 0.001:
                direction = diff / dist     # Unit vector away from obstacle
                normalized_dist = dist / self.influence_radius

                # Repulsive force magnitude: strong when close, zero at boundary
                magnitude = self.gamma * (1.0 / dist**2) * (1 - normalized_dist)
    **2

                force += magnitude * direction

        return force
```

Listing 5: Potential Field Calculation

7 Conclusion

In this project, I explored how a robot can learn pick-and-place trajectories from human demonstrations. Starting with eleven recorded demonstrations, I applied several learning methods and answered key questions about their effectiveness. The code realted to this project and the plots associated can be found in my Github, Located here: `https://github.com/aneangel/robot-learn`

7.1 Summary of Methods and Results

Dynamic Movement Primitives allowed me to encode motion from a single demonstration and generalize to new start and goal positions. With an average reproduction error of 147mm and successful generalization to targets shifted by up to 15cm, DMPs proved effective for adapting learned motions to new task configurations without retraining.

Dynamic Time Warping quantified similarity across all demonstration pairs, revealing that demo1 and demo2 were the most consistent (similarity score of 1.000) while identifying outliers that differed from the majority. This analysis can guide demonstration selection for future learning tasks.

Gaussian Mixture Models with Gaussian Mixture Regression achieved the best trajectory reproduction, with an average error of 63mm (a 57% improvement over DMPs). By modeling the distribution of all demonstrations, GMM/GMR produces trajectories that follow the consensus where demonstrations agree and smoothly interpolates through regions of higher variance.

Complete vs. Piecewise Trajectory Learning showed that preserving full motion structure matters. The complete trajectory approach outperformed piecewise segmentation by 36.32mm (57%

error reduction), demonstrating that GMR benefits from learning temporal dependencies across the entire task rather than isolated segments.

Obstacle Avoidance using potential fields successfully adapted learned DMPs to environments with obstacles not present during training. The system improved minimum clearance by 89mm while maintaining smooth trajectories that still reached the goal position.

7.2 Key Takeaways

With this, we can tell that first, **learning from demonstration is practical**. Even with imperfect data like missing end-effector actuation, irregular timestamps, and natural human variability. The methods I applied successfully captured the essential structure of the pick-and-place task.

Second, **method selection depends on the goal**. DMPs excel when generalization to new configurations is needed from limited demonstrations. GMM/GMR is better suited when multiple demonstrations are available and accurate reproduction of an average trajectory is the priority.

Third, **preserving motion structure improves learning**. Whether through DMPs that encode complete motion primitives or GMM/GMR trained on full trajectories, methods that maintain temporal continuity consistently outperformed approaches that fragment the motion into pieces.