Assignment 3 Report

Leonardo Celi & Oways Jaffer
lgc59-omj9

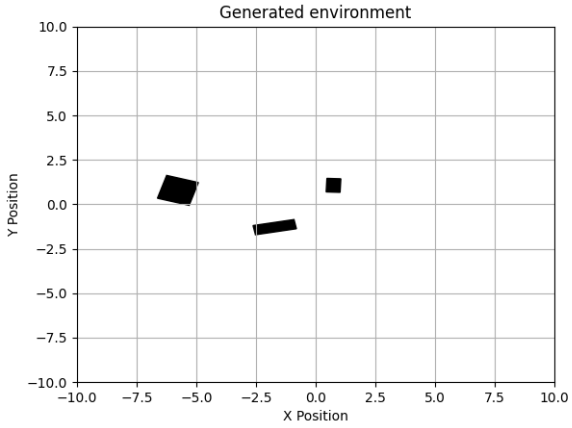
Project details

Language: Python
Libraries used: numpy, matplotlib.pyplot, functools, argparse, gtsam, scipy.optimize, networkx, matplotlib.animation.

Testing and Visualizations

1 Potential Function (Testing Report)

0.0.1 Trial 1

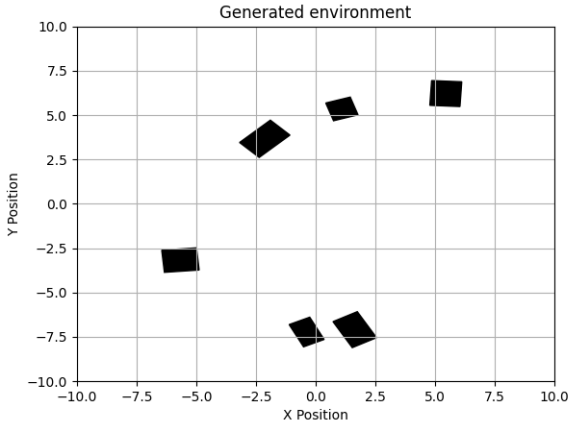


- **Environment:**
- **Number of obstacles**: 3
- **Start/goal configurations**:
  1. (0.0, 0.0) → (-5.0, 6.0)
  2. (4.0, 6.0) → (-7.0, -8.0)
  3. (-2.5, 4.0) → (-9.0, 6.0)
  4. (-1.0, 8.5) → (3.0, -4.0)

5. (-8.0, 8.5) → (8.5, -8.0)

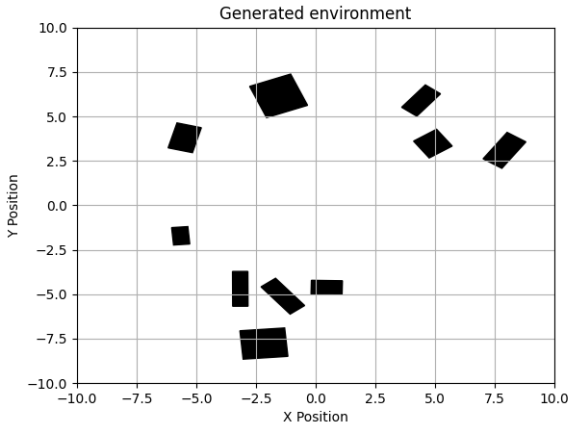
- **Success rate**: 100% (5/5)
- **Average duration of path search**: 0.003841 seconds

0.0.2 Trial 2



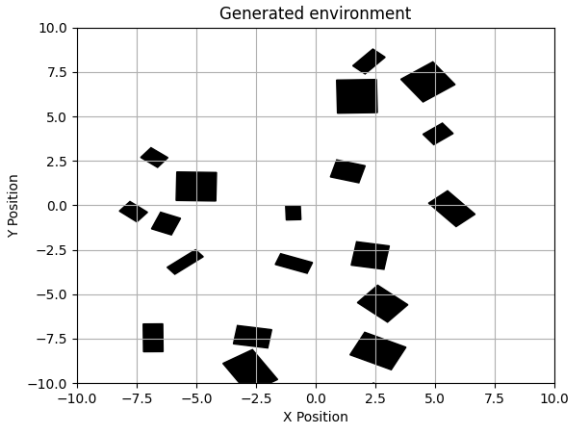
- **Environment:**
- **Number of obstacles**: 6
- **Start/goal configurations**:
  1. (0.0, 0.0) → (-5.0, 6.0)
  2. (4.0, 6.0) → (-7.0, -8.0)
  3. (-7.5, 4.0) → (5.0, -3.0)
  4. (-1.0, 8.5) → (3.0, -4.0)
  5. (0.0, 8.5) → (8.5, -8.0)
- **Success rate**: 80% (4/5) *Failure*: For configuration 2, the algorithm couldn't find a collision-free path as all recognized paths were collision-prone.
- **Average duration of path search**: 0.007825 seconds

0.0.3 Trial 3



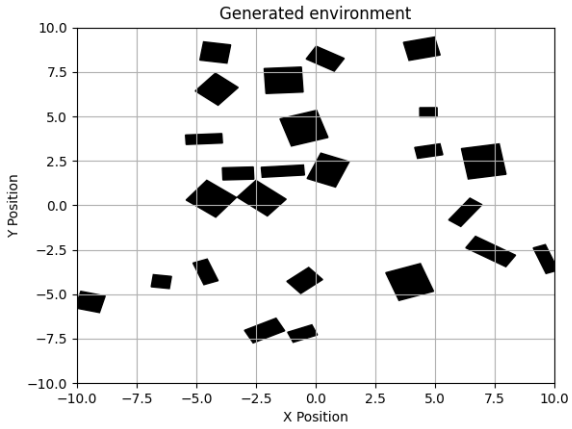
- **Environment:**
- **Number of obstacles**: 10
- **Start/goal configurations**:
  1. (0.0, 0.0) → (-5.0, 6.0)
  2. (4.0, 6.0) → (-7.0, -8.0)
  3. (-7.5, 4.0) → (-5.0, -8.0)
  4. (-1.0, 8.5) → (3.0, -4.0)
  5. (0.0, 8.5) → (8.5, -8.0)
- **Success rate**: 80% (4/5) *Failure*: For configuration 2, the algorithm reached the maximum number of iterations. The start position was likely too far from the goal, requiring more time and data to compute a full path.
- **Average duration of path search**: 0.00181125 seconds

0.0.4 Trial 4



- **Environment:**
- **Number of obstacles**: 20
- **Start/goal configurations**:
  1. (1.5, 0.0) → (-5.0, 6.0)
  2. (4.0, 6.0) → (-7.0, -8.0)
  3. (-7.5, 4.0) → (-5.0, -8.0)
  4. (-1.0, 8.5) → (3.0, -4.0)
  5. (0.0, 8.5) → (8.5, -8.0)
- **Success rate**: 40% (2/5) *Failures*: For configurations 2, 3, and 4, the algorithm reached the maximum number of iterations. Start positions were likely too far from goal positions, necessitating more time and data to compute full paths.
- **Average duration of path search**: 0.0942 seconds

0.0.5 Trial 5

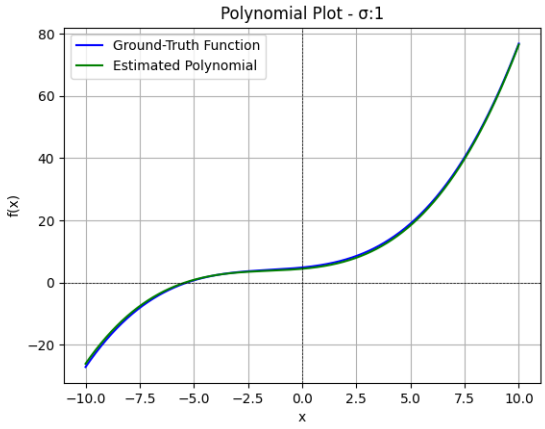


- **Environment:**
- **Number of obstacles**: 25
- **Start/goal configurations**:
  1. (1.5, 0.0) → (-5.0, 6.0)
  2. (2.5, 2.5) → (-2.5, -5.0)
  3. (-7.5, 4.0) → (-5.0, -8.0)
  4. (-1.0, 8.5) → (3.0, -4.0)
  5. (1.0, 0.5) → (-7.5, -8.0)
- **Success rate**: 40% (2/5) *Failures*: For configurations 2, 3, and 4, the algorithm reached the maximum number of iterations. Start positions were likely too far from goal positions, necessitating more time and data to compute full paths.
- **Average duration of path search**: 0.003325 seconds

2 Gtsam: Factor Graphs (Testing Report)

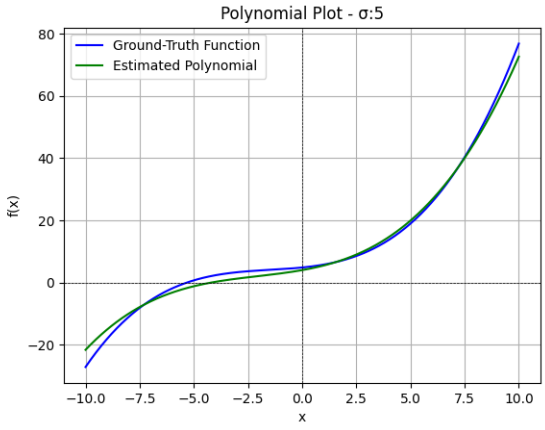
Each trial was run with the following command: python fg_polynomial —initial 1 2 3 4

0.0.6 Trial 1



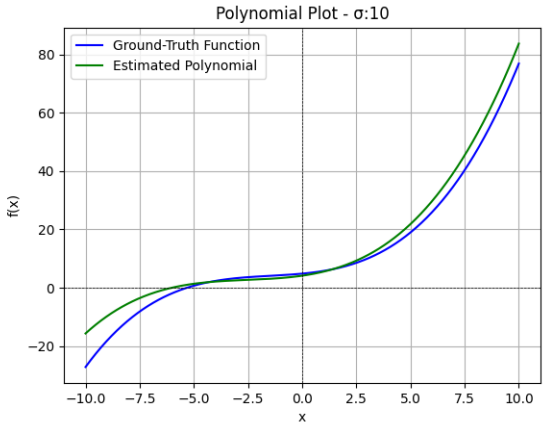
- **Plot Visual:**
- **Ground Truth Polynomial**: $f(x) = 0.045x^3 + 0.2x^2 + 0.7x + 4.86$
- **Noise Level**: $= 1$
- **Estimated Polynomial**: $f(x) = 0.045086x^3 + 0.207x^2 + 0.623x + 4.45$

0.0.7 Trial 2



- **Plot Visual:**
- **Ground Truth Polynomial**: $f(x) = 0.045x^3 + 0.2x^2 + 0.7x + 4.86$
- **Noise Level**: $= 5$
- **Estimated Polynomial**: $f(x) = 0.0346x^3 + 0.215x^2 + 1.24x + 4.02$

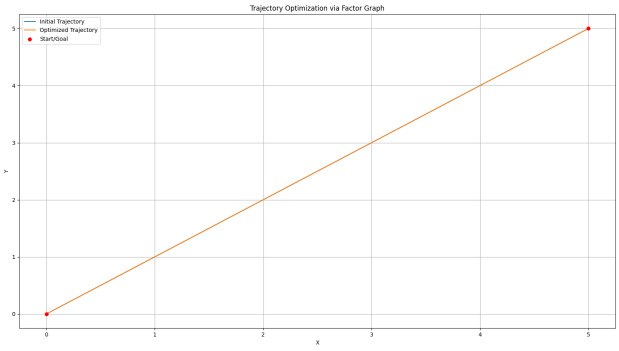
0.0.8 Trial 3

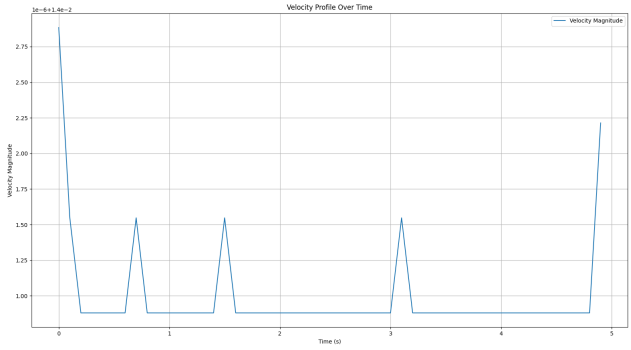


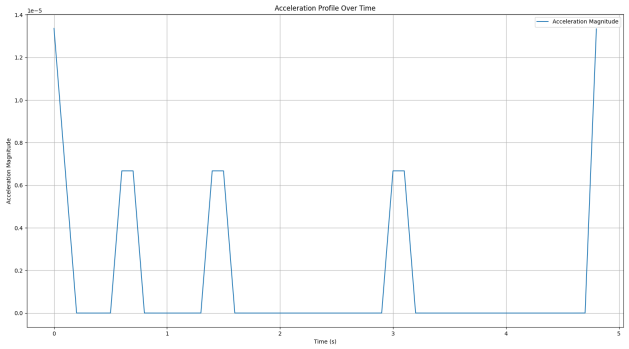
- **Plot Visual:**
- **Ground Truth Polynomial**: $f(x) = 0.045x^3 + 0.2x^2 + 0.7x + 4.86$
- **Noise Level**: $= 10$
- **Estimated Polynomial**: $f(x) = 0.039x^3 + 0.298x^2 + 1.07x + 4.16$

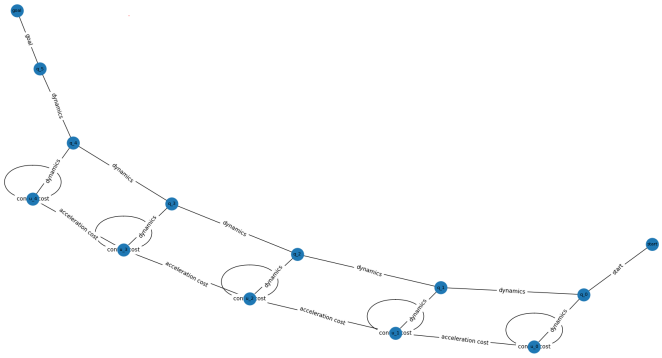
Trajectory Optimization via Factor Graphs

3.1 Simple Trajectory
fg_traj_opt.py



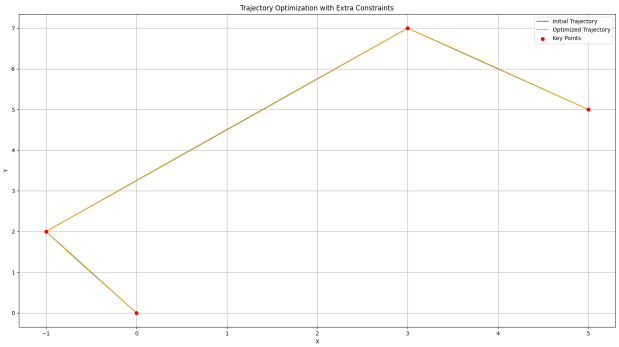


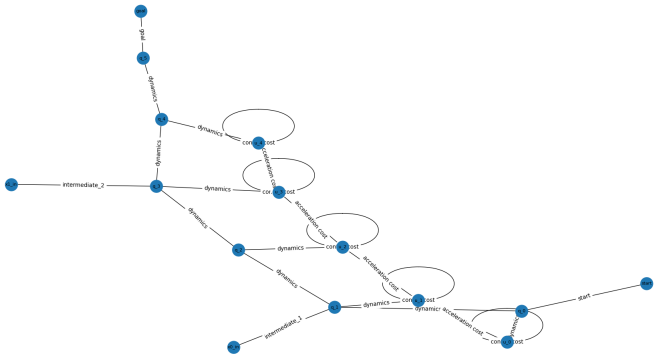


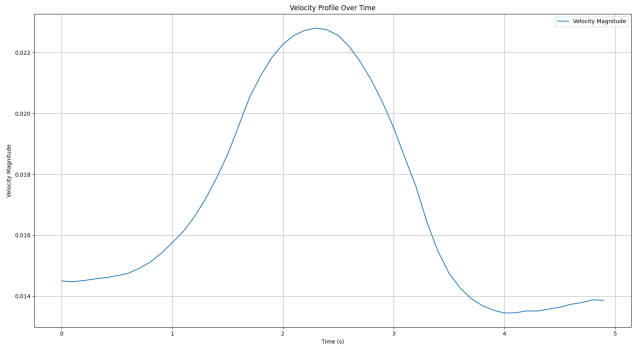


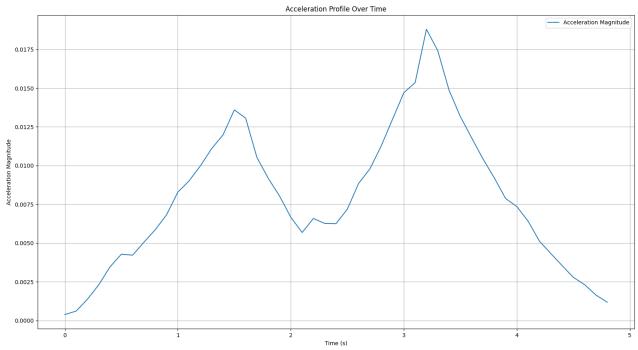
Factor Graph Diagram: A diagram illustrating the factor graph with a few states (e.g., for $T = 5$). Show the states q_t, control inputs u_t, and the factors connecting them (dynamics factors, control cost factors, acceleration cost factors, start and goal factors).
Trajectory Optimization Graph: Please note that the blue line in the first image does not appear until you zoom in. This applies to all other graphs in this report as well.

3.2 Extra Constraints
fg_traj_opt_2.py







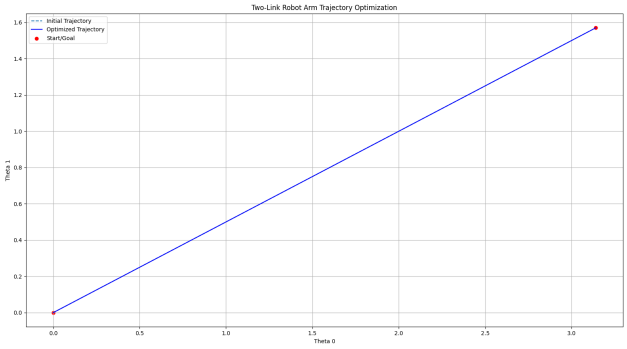


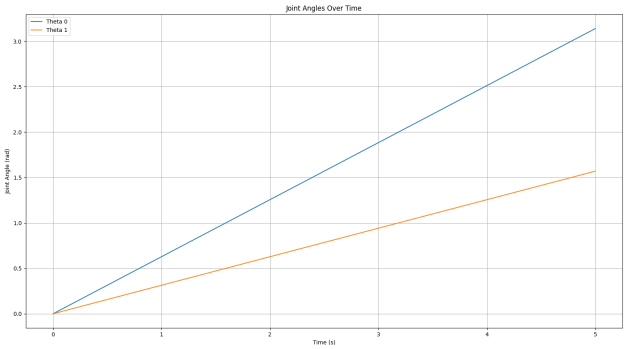
Modified Factor Graph Diagram: A diagram of the factor graph highlighting the new intermediate state factors at $T/3$ and $2T/3$.

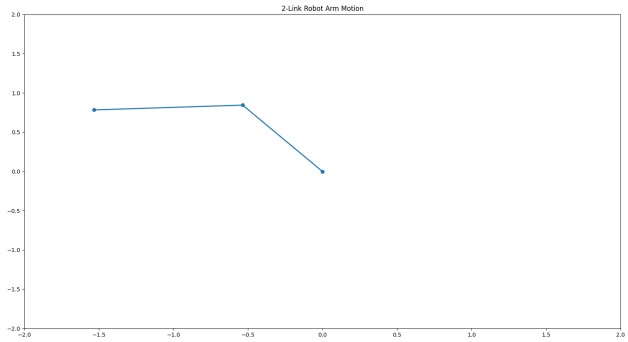
Trajectory Plot with Intermediate States: A plot showing the initial and optimized trajectories in the X-Y plane, including the intermediate states x_{in}^0 and x_{in}^1. Mark the key points (start, intermediate states, goal).

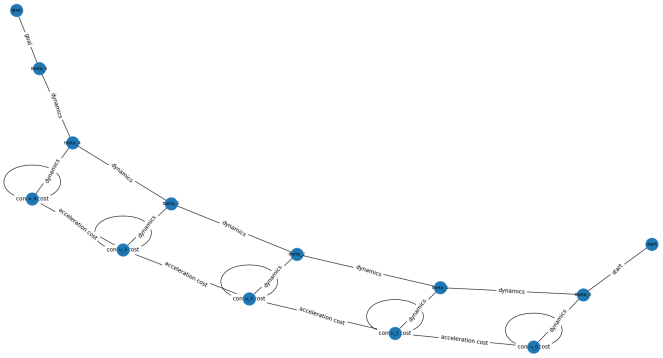
Animation of the Trajectory (Not Shown): An animation showing the point moving along the optimized trajectory, visibly passing through the intermediate states.

3.3 2-Link Robot Arm
fg_traj_opt_arm.py









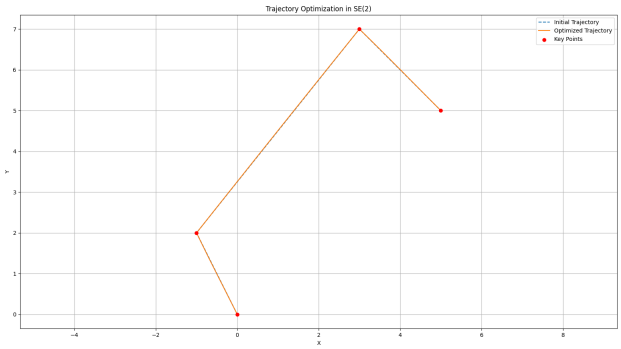
Factor Graph Diagram for Robot Arm: A diagram illustrating the factor graph for the 2-link robot arm, showing joint angles θ_t, control inputs u_t, and the associated factors.

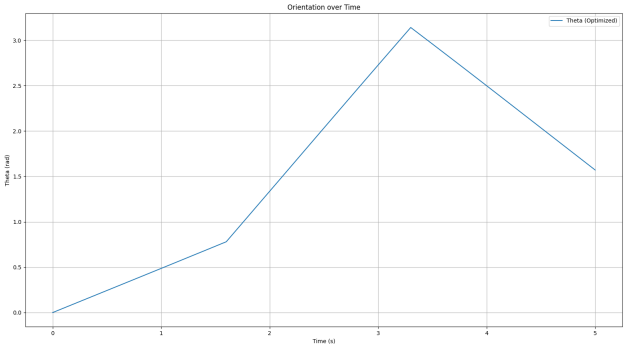
Joint Angle Trajectory Plot: A plot of the initial and optimized trajectories in joint angle space (θ_0 vs. θ_1).

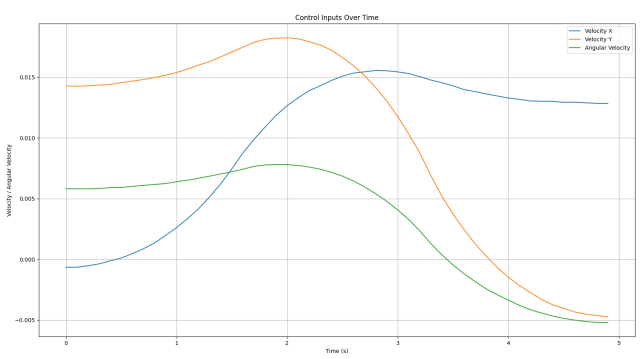
Animation of the Robot Arm: An animation showing the movement of the 2-link robot arm along the optimized trajectory.

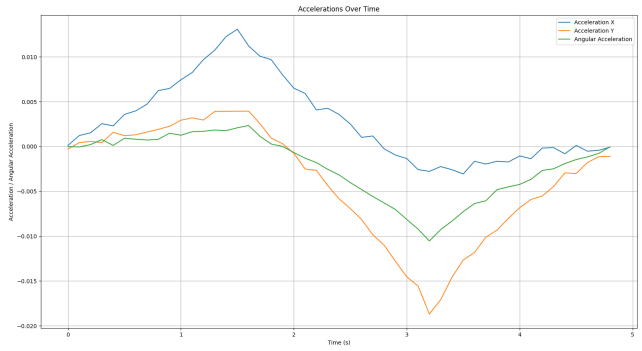
Trajectory Optimization in SE(2)

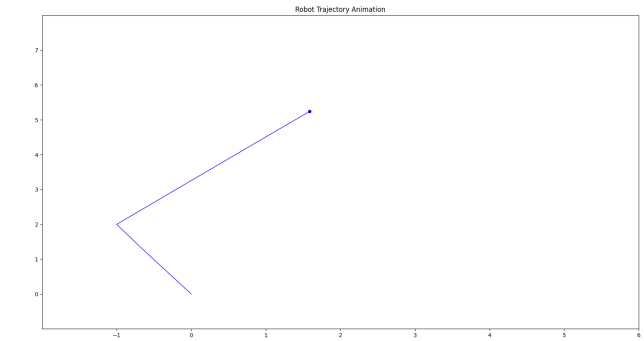
fg_traj_opt_se2.py

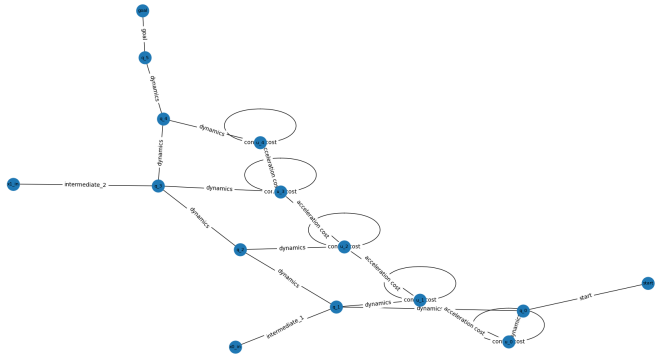












Factor Graph Diagram in SE(2): A diagram of the factor graph representing poses $q_t = (x_t, y_t, \theta_t)$, control inputs u_t, and factors, including dynamics, control costs, acceleration costs, and constraints (start, goal, intermediate poses).

Trajectory Plot in X-Y Plane: A plot showing the initial and optimized trajectories, including the start, goal, and intermediate poses. Mark key points for clarity.

Orientation Over Time Plot: A plot of the robot's orientation θ_t over time.

Control Inputs and Acceleration Profiles: Plots of the control inputs (velocities) and accelerations over time.

Animation of the Robot Trajectory: An animation showing the robot moving along the optimized trajectory in SE(2), passing through intermediate poses.

Implementation Specifications

potential.py

Potential Functions

- scene_from_file(filename)
- gradient_repulsive_potential(q, polygon)
- gradient_potential(q, goal, obstacles)
- collisionCheckFreeBody(newPolygon, existingPolygon)
- pathCollisionCheck(p1, p2, polygon)
- line_segment_intersection(p1, p2, q1, q2)
- visualize_robot_path(q, robotCorners, obstacles, goal, path)

Description: This code plans a robot's path in a 2D space using potential fields, where the robot is "pulled" toward a goal and "repelled" from obstacles. It loads an environment with obstacles from a file, calculates the attractive and repulsive forces, and updates the robot's position using gradient descent. The robot stops when it reaches the goal or detects a collision-free path isn't possible. In each step, the robot checks for collisions with obstacles using geometry and adjusts its movement by reducing the step size if necessary. The path is visualized in real-time, showing the robot navigating toward the goal while avoiding obstacles.

Gtsam: Factor Graphs

fg_polynomial.py

- f(x, a=0.045, b=0.2, c=0.7, d=4.86)
- error_func(y: np.ndarray, x: np.ndarray, this: gtsam.CustomFactor, v: gtsam.Values, H: List[np.ndarray])
- main()

Description:
The modified code extends the original linear regression implementation to fit a third-order polynomial by incorporating additional coefficients for higher-order terms (a, b, c, d) and their respective Jacobians. Key changes include updating the "true" function ‘f(x)‘ to a third-order polynomial $f(x) = ax^3 + bx^2 + cx + d$, modifying ‘errorfunc‘ to handle four parameters and compute the partial derivatives ($\frac{\partial \text{error}}{\partial a}, \frac{\partial \text{error}}{\partial b}, \frac{\partial \text{error}}{\partial c}, \frac{\partial \text{error}}{\partial d}$) with respect to these parameters. In the main function, new keys (‘ka‘, ‘kb‘, ‘kc‘, ‘kd‘) and initial guesses for these coefficients are defined, and the factor graph is built to model the polynomial relationship with added Gaussian noise. The implementation effectively uses GTSAM's ‘CustomFactor‘ and the Levenberg-Marquardt optimizer to estimate the coefficients, which are validated by comparing the noisy data to the estimated polynomial and the ground-truth function through visualization and printed results.

Trajectory Optimization via Factor Graphs

3.1 Simple Trajectory
fg_traj_opt.py

- `dynamics_factor(dt, t)`
- `start_factor(start)`
- `goal_factor(goal)`
- `control_cost_factor(t)`
- `acceleration_cost_factor(dt, t)`
- `visualize_factor_graph(T)`

We implemented trajectory optimization for a point mass moving in 2D space using a factor graph. The variables are positions q_t and control inputs u_t. The factors include dynamics factors enforcing $q_{t+1} = q_t + u_t \Delta t$, start and goal factors enforcing the initial and final positions, and cost factors penalizing control efforts and accelerations to promote smoothness. The initial guess was a straight line from the start to the goal, with controls computed accordingly.

3.2 Extra Constraints
fg_traj_opt_2.py

- `dynamics_factor(dt, t)`
- `start_factor(start)`
- `goal_factor(goal)`
- `intermediate_factor(intermediate_state, t)`
- `control_cost_factor(t)`
- `acceleration_cost_factor(dt, t)`
- `visualize_factor_graph(T)`
- `init()`
- `animate(i)`

We extended the previous trajectory optimization by adding intermediate state constraints at times $T/3$ and $2T/3$, ensuring the trajectory passes through specified points x^0_{in} and x^1_{in}. New factors were added to the factor graph to enforce these constraints with large weights. The initial guess was constructed by piecewise linear interpolation through the start, intermediate states, and goal, providing a feasible starting point for optimization.

2-Link Robot Arm
fg_traj_opt_se2.py

- `wrap_angles(angle)`
- `dynamics_factor(dt, t)`
- `start_factor(start)`
- `goal_factor(goal)`
- `control_cost_factor(t)`
- `acceleration_cost_factor(dt, t)`
- `forward_kinematics(theta)`
- `init()`
- `animate(i)`
- `visualize_factor_graph(T)`

We implemented trajectory optimization for a 2-link robot arm, where the state is defined by joint angles $q_t = (\theta_0, \theta_1)$. Factors in the graph include dynamics factors enforcing $\theta_{t+1} = \theta_t + u_t \Delta t$, start and goal factors for initial and final joint angles, and cost factors penalizing control inputs and accelerations for smoothness. We handled angle wrapping manually to account for the periodic nature of joint angles. The initial guess was a linear interpolation between the start and goal angles.

Trajectory Optimization in SE(2)

fg_traj_opt_se2.py

- `wrap_angle(angle)`
- `dynamics_factor(dt, t)`
- `start_factor(start)`
- `goal_factor(goal)`
- `intermediate_factor(intermediate_pose, t)`
- `control_cost_factor(t)`
- `acceleration_cost_factor(dt, t)`
- `init()`
- `animate(i)`

- visualize_factor_graph(T)

We extended trajectory optimization to $SE(2)$ by optimizing over poses $q_t = (x_t, y_t, \theta_t)$ and control inputs u_t. The factor graph includes dynamics factors enforcing $q_{t+1} = q_t + u_t \Delta t$ with angle wrapping for θ_t, start and goal factors, intermediate pose constraints, and cost factors penalizing control inputs and accelerations. The initial guess was generated by interpolating between waypoints (start, intermediate poses, goal) with proper angle wrapping to ensure continuity.