Turtlebot Seeker–Target Interception in Unknown Cluttered Environments
with Monte Carlo Localization and Model Predictive Control

Tim, Gabe, Thomas, Ryan, Kush

Contents

1 Problem Formulation

We consider two differential-drive robots (Turtlebots) in a cluttered, initially unknown environment:

- A *seeker* robot that attempts to intercept the target.
- A *target* robot that moves through the environment.

The environment is partly unknown and is sensed using 2D LIDAR. The seeker builds a map and estimates its own pose using Monte Carlo Localization (MCL). The target pose is estimated via a separate estimator (e.g. Kalman filter) based on relative measurements. A receding-horizon Model Predictive Controller (MPC) continuously recomputes a trajectory to intercept the target in approximately minimum time while avoiding obstacles and respecting dynamics.

1.1 State, Control, and Map

Let the true system state at discrete time t be

$$X_t = \big(x_t^{\text{seek}}, x_t^{\text{tgt}}, M_t\big),$$

where

- x_t^{seek} is the seeker state (pose and velocity).
- x_t^{tgt} is the target state (position and velocity).
- M_t is the map / occupancy grid (environment).

The control input applied to the seeker is

$$u_t \in \mathcal{U},$$

typically linear acceleration and angular velocity, or directly wheel commands.

The observations are

$$z_t = \big(z_t^{\text{LIDAR}}, z_t^{\text{tgt}}\big),$$

where z_t^{LIDAR} is a LIDAR scan and z_t^{tgt} is a measurement of the target position (e.g. via vision).

The (stochastic) dynamics and observation models are

$$X_{t+1} \sim p(X_{t+1} \mid X_t, u_t), \tag{1}$$
$$z_t \sim p(z_t \mid X_t). \tag{2}$$

The optimal control problem in its most general form is a partially observable stochastic control problem (POMDP). Instead of solving the full POMDP, we:

1. Maintain a belief over seeker pose via MCL.
2. Maintain a belief over target state via (E)KF.
3. Use a point estimate plus uncertainty summaries (covariances) inside a receding-horizon MPC.

1.2 Objective

Define an interception condition at some time T:

$$x_T^{\text{seek}} \in \mathcal{X}_{\text{catch}}(x_T^{\text{tgt}}) := \left\{ x : \|p_T^{\text{seek}} - p_T^{\text{tgt}}\| \leq r_{\text{catch}} \right\},$$

where $p_T^{\text{seek}} \in \mathbb{R}^2$ is the seeker position and p_T^{tgt} is the target position.

The ideal objective is to minimize T (time to intercept) subject to dynamics, actuator limits, and obstacle avoidance. In practice we approximate this using a fixed-horizon MPC with a cost that strongly rewards rapid reduction of distance to the target.

2 System Modeling

2.1 Seeker Dynamics (Unicycle Model)

We adopt a unicycle model for the seeker. The state is

$$x_t^{\text{seek}} = \begin{bmatrix} p_{x,t} \\ p_{y,t} \\ \theta_t \\ v_t \end{bmatrix},$$

where $(p_{x,t}, p_{y,t})$ is position in the map frame, θ_t is heading, and v_t is linear speed.

The control is

$$u_t = \begin{bmatrix} a_t \\ \omega_t \end{bmatrix},$$

where a_t is linear acceleration and ω_t is angular velocity.

Continuous-time dynamics:

$$\begin{aligned} \dot{p}_x &= v \cos\theta, && (3) \\ \dot{p}_y &= v \sin\theta, && (4) \\ \dot{\theta} &= \omega, && (5) \\ \dot{v} &= a. && (6) \end{aligned}$$

With sampling time Δt, the noise-free discrete-time dynamics are

$$\begin{aligned} p_{x,t+1} &= p_{x,t} + \Delta t\, v_t \cos\theta_t, && (7) \\ p_{y,t+1} &= p_{y,t} + \Delta t\, v_t \sin\theta_t, && (8) \\ \theta_{t+1} &= \theta_t + \Delta t\, \omega_t, && (9) \\ v_{t+1} &= v_t + \Delta t\, a_t. && (10) \end{aligned}$$

Including process noise w_t:

$$x_{t+1}^{\text{seek}} = f(x_t^{\text{seek}}, u_t) + w_t.$$

2.2 Target Dynamics (Constant Velocity Model)

For the target, we use a simple nearly-constant-velocity model. The state is

$$x_t^{\text{tgt}} = \begin{bmatrix} p_{x,t}^{\text{tgt}} \\ p_{y,t}^{\text{tgt}} \\ v_{x,t}^{\text{tgt}} \\ v_{y,t}^{\text{tgt}} \end{bmatrix}.$$

The discrete-time dynamics are

$$x_{t+1}^{\text{tgt}} = Ax_t^{\text{tgt}} + w_t^{\text{tgt}},$$

where

$$A = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The process noise w_t^{tgt} models random accelerations.

2.3 Map and LIDAR Measurement Model

We represent the map as a 2D occupancy grid:

$$M = \{m_i\}_{i=1}^{N_{\text{cells}}}, \quad m_i \in \{0, 1\},$$

where $m_i = 1$ indicates an occupied cell.

The LIDAR sensor at time t produces a set of range measurements

$$z_t^{\text{LIDAR}} = \{r_t^k\}_{k=1}^{N_{\text{beams}}}$$

at fixed angles $\{\phi^k\}$ in the robot frame. Given a hypothesized pose (x, y, θ) and map M, one can raycast to obtain expected ranges $\hat{r}^k(x, y, \theta, M)$.

A common Gaussian measurement model is:

$$p(z_t^{\text{LIDAR}} \mid x_t^{\text{seek}}, M) \propto \exp\left(-\frac{1}{2}\sum_k \frac{(r_t^k - \hat{r}^k(x_t^{\text{seek}}, M))^2}{\sigma_r^2}\right).$$

2.4 Target Observation Model

Suppose a vision system provides a noisy measurement of the target position in the map frame:

$$z_t^{\text{tgt}} = \begin{bmatrix} z_{x,t}^{\text{tgt}} \\ z_{y,t}^{\text{tgt}} \end{bmatrix} = Hx_t^{\text{tgt}} + v_t^{\text{tgt}},$$

where

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \quad v_t^{\text{tgt}} \sim \mathcal{N}(0, R).$$

3 Monte Carlo Localization (MCL)

3.1 Bayes Filter

The belief over the seeker pose is

$$bel(x_t) = p(x_t^{\text{seek}} \mid z_{0:t}, u_{0:t-1}, M).$$

The Bayes filter recursion consists of:

Prediction:

$$\overline{bel}(x_t) = \int p(x_t \mid x_{t-1}, u_{t-1})\, bel(x_{t-1})\, dx_{t-1}.$$

Update:

$$bel(x_t) = \eta\, p(z_t^{\text{LIDAR}} \mid x_t, M)\, \overline{bel}(x_t),$$

where η is a normalizing constant.

3.2 Particle Filter Approximation

We approximate the belief by a weighted set of particles:

$$bel(x_t) \approx \sum_{i=1}^{N_p} w_t^{(i)}\, \delta(x_t - x_t^{(i)}),$$

where $x_t^{(i)}$ is the i-th particle and $w_t^{(i)}$ its weight.

MCL Algorithm

Algorithm 1 Monte Carlo Localization (one time step)

Require: particles $\{x_{t-1}^{(i)}, w_{t-1}^{(i)}\}_{i=1}^{N_p}$, control u_{t-1}, LIDAR scan z_t^{LIDAR}, map M

Ensure: new particles $\{x_t^{(i)}, w_t^{(i)}\}_{i=1}^{N_p}$

1: **for** $i = 1$ to N_p **do**

2: Sample motion:

$$x_t^{(i)} \sim p(x_t \mid x_{t-1}^{(i)}, u_{t-1})$$

using the unicycle motion model plus noise.

3: Compute importance weight:

$$w_t^{(i)} \leftarrow p(z_t^{\text{LIDAR}} \mid x_t^{(i)}, M)$$

4: **end for**

5: Normalize weights:

$$w_t^{(i)} \leftarrow \frac{w_t^{(i)}}{\sum_j w_t^{(j)}}$$

6: Resample N_p particles from $\{x_t^{(i)}\}$ according to $\{w_t^{(i)}\}$.

3.3 Pose Estimate and Covariance

From the particle set, approximate the mean pose as:

$$\hat{x}_t^{\text{seek}} = \sum_{i=1}^{N_p} w_t^{(i)} x_t^{(i)}.$$

The covariance is

$$P_t^{\text{seek}} = \sum_{i=1}^{N_p} w_t^{(i)} (x_t^{(i)} - \hat{x}_t)(x_t^{(i)} - \hat{x}_t)^\top.$$

In practice one often only uses the 2×2 position covariance.

4 Target State Estimation via Kalman Filter

Because the target model is linear with Gaussian noise, we use a Kalman Filter.

4.1 Dynamics and Observation

Dynamics:

$$x_{t+1}^{\text{tgt}} = A x_t^{\text{tgt}} + w_t^{\text{tgt}}, \quad w_t^{\text{tgt}} \sim \mathcal{N}(0, Q).$$

Observation:

$$z_t^{\text{tgt}} = H x_t^{\text{tgt}} + v_t^{\text{tgt}}, \quad v_t^{\text{tgt}} \sim \mathcal{N}(0, R).$$

4.2 Kalman Filter Equations

Let $\hat{x}_{t|t}$ be the filtered estimate and $P_{t|t}$ the corresponding covariance.

Prediction

$$\hat{x}_{t|t-1} = A \hat{x}_{t-1|t-1}, \tag{11}$$

$$P_{t|t-1} = A P_{t-1|t-1} A^\top + Q. \tag{12}$$

Update

$$S_t = H P_{t|t-1} H^\top + R, \tag{13}$$

$$K_t = P_{t|t-1} H^\top S_t^{-1}, \tag{14}$$

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t (z_t^{\text{tgt}} - H \hat{x}_{t|t-1}), \tag{15}$$

$$P_{t|t} = (I - K_t H) P_{t|t-1}. \tag{16}$$

The KF provides $\hat{x}_t^{\text{tgt}}$ and P_t^{tgt}.

5 Model Predictive Control (MPC)

MPC repeatedly solves a finite-horizon optimal control problem using current state estimates and predicted target motion.

5.1 Finite-Horizon Optimal Control Problem

At time t, let $\hat{x}_t^{\text{seek}}$ be the seeker pose estimate and $\hat{x}_t^{\text{tgt}}$ the target state estimate. We define a prediction horizon of length N. For prediction step $k = 0, \dots, N$ we have:

- seeker state: x_k,
- seeker control: u_k,
- predicted target state: x_k^{tgt}.

The predicted target states are obtained by propagating the KF estimate:

$$x_{k+1}^{\text{tgt}} = A x_k^{\text{tgt}}.$$

The seeker dynamics are:

$$x_{k+1} = f(x_k, u_k), \quad k = 0, \dots, N-1,$$

with $x_0 = \hat{x}_t^{\text{seek}}$.

5.2 Constraints

We impose:

- **Actuator limits:**
$$v_{\min} \le v_k \le v_{\max}, \quad a_{\min} \le a_k \le a_{\max}, \quad |\omega_k| \le \omega_{\max}.$$
- **Obstacle avoidance:** for a set of obstacles approximated as circles with centers c^o and radii r^o,
$$\|p_k - c^o\|_2 \ge r_{\text{eff}}^o,$$
where r_{eff}^o is an inflated radius (see Section 6).

5.3 Cost Function

Define the seeker and target positions:

$$p_k = \begin{bmatrix} p_{x,k} \\ p_{y,k} \end{bmatrix}, \quad p_k^{\text{tgt}} = \begin{bmatrix} p_{x,k}^{\text{tgt}} \\ p_{y,k}^{\text{tgt}} \end{bmatrix}.$$

A typical quadratic stage cost is:

$$\begin{aligned} \ell(x_k, u_k, x_k^{\text{tgt}}) = (p_k - p_k^{\text{tgt}})^\top Q (p_k - p_k^{\text{tgt}}) + q_\theta (\theta_k - \theta_k^{\text{tgt}})^2 \\ + u_k^\top R u_k + (u_k - u_{k-1})^\top R_\Delta (u_k - u_{k-1}), \end{aligned} \tag{17}$$

with u_{-1} taken as the previously applied control. The terminal cost is:

$$\ell_f(x_N, x_N^{\text{tgt}}) = (p_N - p_N^{\text{tgt}})^\top Q_f (p_N - p_N^{\text{tgt}}).$$

The total horizon cost is

$$J = \sum_{k=0}^{N-1} \ell(x_k, u_k, x_k^{\text{tgt}}) + \ell_f(x_N, x_N^{\text{tgt}}).$$

5.4 Approximate Minimum-Time Behavior

Strict time-optimal control would minimize the interception time T explicitly. Instead, we approximate minimum time by the choice of cost and constraints.

Let

$$d_k = \|p_k - p_k^{\text{tgt}}\|$$

be the distance between seeker and target. One can add a "progress" term:

$$\ell_{\text{progress}} = -\alpha(d_{k-1} - d_k),$$

which rewards large reductions in distance per step.

An approximate time-to-go can be defined as

$$T_{\text{guess}} = \frac{\|\hat{p}_t^{\text{seek}} - \hat{p}_t^{\text{tgt}}\|}{v_{\max}}.$$

The weighting matrices Q, Q_f, and α can be adapted as functions of T_{guess} to make the controller more aggressive when close to interception.

6 Uncertainty-Aware Constraints

The MCL and KF provide covariances P_t^{seek} and P_t^{tgt} that quantify uncertainty. We use them to adapt constraints.

6.1 Obstacle Inflation via Pose Uncertainty

Let P_t^{seek} be the 2×2 position covariance of the seeker. Let $\lambda_{\max}(P_t^{\text{seek}})$ be its largest eigenvalue. Define a scalar uncertainty measure:

$$\sigma_t^{\text{seek}} = \sqrt{\lambda_{\max}(P_t^{\text{seek}})}.$$

For an obstacle with base safe radius r^o, define the inflated radius:

$$r_{\text{eff}}^o = r^o + k_\sigma \sigma_t^{\text{seek}},$$

where $k_\sigma > 0$ is a tuning parameter.

The deterministic collision avoidance constraint at prediction step k becomes:

$$\|p_k - c^o\|_2 \geq r_{\text{eff}}^o.$$

As localization improves (smaller σ_t^{seek}), r_{eff}^o shrinks and the robot can drive closer to obstacles.

6.2 Interception Region with Uncertainty

Let the seeker and target positions at step k be independent Gaussians:

$$p_k^{\text{seek}} \sim \mathcal{N}(\mu_k^{\text{seek}}, \Sigma_k^{\text{seek}}), \tag{18}$$

$$p_k^{\text{tgt}} \sim \mathcal{N}(\mu_k^{\text{tgt}}, \Sigma_k^{\text{tgt}}). \tag{19}$$

The difference

$$d_k = p_k^{\text{seek}} - p_k^{\text{tgt}}$$

is Gaussian with

$$d_k \sim \mathcal{N}(\mu_k^d, \Sigma_k^d),$$

where

$$\mu_k^d = \mu_k^{\text{seek}} - \mu_k^{\text{tgt}}, \tag{20}$$

$$\Sigma_k^d = \Sigma_k^{\text{seek}} + \Sigma_k^{\text{tgt}}. \tag{21}$$

A chance constraint for interception could be

$$\mathbb{P}\left(\|d_k\| \leq r_{\text{catch}}\right) \geq 1 - \delta.$$

An approximate deterministic surrogate is

$$\|\mu_k^d\| \leq r_{\text{catch}} - \beta_\delta \sqrt{\lambda_{\max}(\Sigma_k^d)},$$

where β_δ is chosen from Gaussian tail bounds.

In practice, we may simply set

$$r_{\text{catch,eff}} = r_{\text{catch,base}} + k_{\sigma,\text{catch}} \sqrt{\lambda_{\max}(\Sigma_k^d)}$$

and require $\|\mu_k^d\| \leq r_{\text{catch,eff}}$.

6.3 Speed Limits Based on Uncertainty

One can also modulate speed based on seeker uncertainty:

$$s_t = \sqrt{\lambda_{\max}(P_t^{\text{seek}})},$$

$$v_{\max}(t) = v_{\max}^{\text{base}}\, e^{-\alpha s_t},$$

for some $\alpha > 0$. The speed constraint becomes

$$|v_k| \leq v_{\max}(t).$$

As MCL converges and uncertainty shrinks, $v_{\max}(t)$ approaches $v_{\max}^{\text{base}}$ and the robot can move faster.

7 Closed-Loop Algorithm

We now summarize the overall closed-loop algorithm.

Algorithm 2 Closed-Loop MCL + KF + MPC

Initialize MCL particles for seeker pose
Initialize KF for target state
while experiment running **do**
    Receive control u_{t-1} that was sent at previous step
    Receive new LIDAR scan z_t^{LIDAR} and target measurement z_t^{tgt}
    MCL update:
        $bel(x_t^{\text{seek}}) \leftarrow \text{MCL}(bel(x_{t-1}^{\text{seek}}), u_{t-1}, z_t^{\text{LIDAR}}, M_t)$
        Extract $\hat{x}_t^{\text{seek}}$, P_t^{seek}
    KF update:
        $(\hat{x}_t^{\text{tgt}}, P_t^{\text{tgt}}) \leftarrow \text{KF}(\hat{x}_{t-1}^{\text{tgt}}, P_{t-1}^{\text{tgt}}, z_t^{\text{tgt}})$
    Prediction for horizon:
        Predict target states x_k^{tgt} for $k = 0, \ldots, N$
        Set initial seeker state $x_0 = \hat{x}_t^{\text{seek}}$
        Compute inflated obstacle radii and interception radius based on P_t^{seek}, P_t^{tgt}
    MPC:
        Solve finite-horizon OCP for x_k, u_k to minimize cost subject to dynamics and constraints
        Extract $u_t = u_0^*$
    Apply control:
        Send u_t to seeker robot
    $t \leftarrow t + 1$
end while

8 Offline Probabilistic Trajectory Generation

To initialize the real-time interception system, we compute an *offline probabilistic trajectory* from the current seeker pose to a desired goal location. This trajectory is generated using a stochastic motion model together with noise propagation, producing a sequence of waypoints

$$\big\{(\bar{x}_k, \Sigma_k)\big\}_{k=0}^{N_{\text{off}}},$$

where $\bar{x}_k$ is the nominal waypoint pose and Σ_k is the predicted covariance of pose uncertainty at that point. These covariance matrices define confidence ellipses along the trajectory, giving a spatial representation of uncertainty (Figure 1).

8.1 Stochastic Motion Model

We assume the seeker obeys the unicycle dynamics

$$x_{k+1} = f(x_k, u_k) + w_k,$$

with process noise

$$w_k \sim \mathcal{N}(0, Q_k).$$

Given a nominal control sequence $\{u_k\}$ produced by a deterministic planner (e.g. straight-line interpolation, A* path, RRT), we propagate uncertainty using the first-order linearization

$$A_k = \frac{\partial f}{\partial x}\bigg|_{\bar{x}_k, u_k}, \qquad B_k = \frac{\partial f}{\partial u}\bigg|_{\bar{x}_k, u_k}.$$

The covariance recursion becomes

$$\Sigma_{k+1} = A_k \Sigma_k A_k^\top + Q_k.$$

The result is a *belief trajectory*:

$$\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_{N_{\text{off}}}, \qquad \Sigma_0, \Sigma_1, \ldots, \Sigma_{N_{\text{off}}}.$$

8.2 Sampling-Based Trajectory Visualization

In addition to covariance propagation, Monte Carlo realizations

$$x_k^{(i)} = f(x_{k-1}^{(i)}, u_{k-1}) + w_{k-1}^{(i)}, \qquad i = 1, \ldots, N_s,$$

are generated to visualize dispersion around the nominal path. Plotting these noisy sampled trajectories produces the red ensemble shown in Figure 1. Confidence ellipses, computed from Σ_k, illustrate the predicted uncertainty at each waypoint.

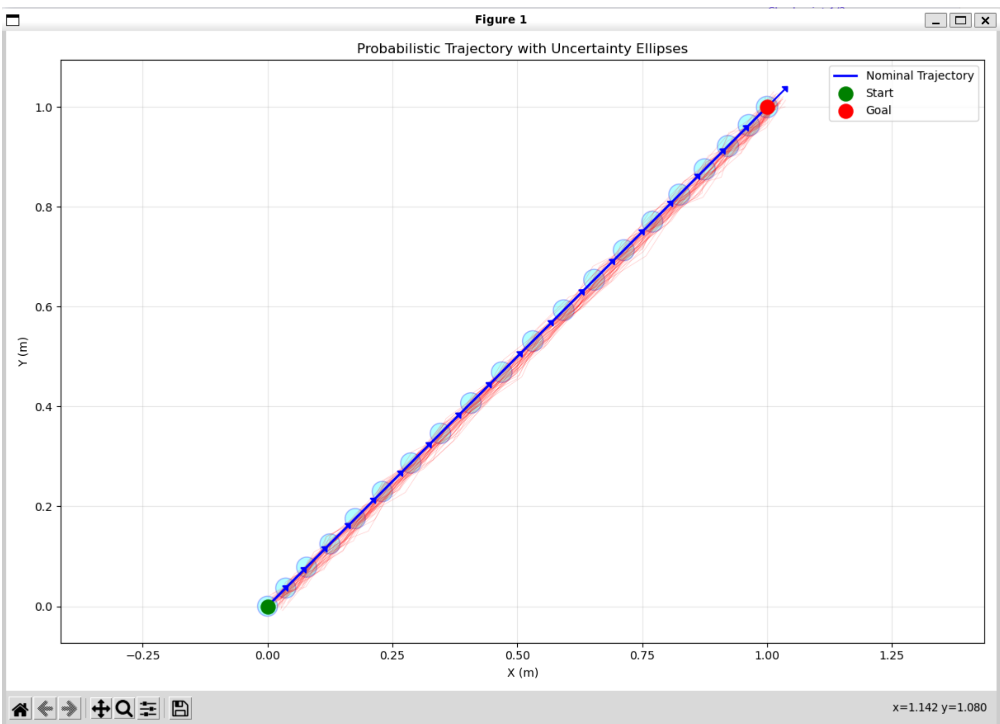

Figure 1: Offline probabilistic trajectory with covariance ellipses and sampled paths.

8.3 Role of Offline Trajectory in the Real-Time System

The offline trajectory serves three purposes:

1. It provides an initial **belief-state path** for the MPC to track.
2. It provides **uncertainty priors** Σ_k used to shape:
   - obstacle inflation radii,

  - speed limits,
  - catching-radius uncertainty bounds during interception.

3. It initializes the controller before MCL converges, avoiding erratic early-stage control.

During real-time operation, these offline waypoints are *not* followed blindly. Instead, they act as a *warm start* for the adaptive MPC and belief updates, which then continuously re-plan based on real measurements.

9 Integration of Offline Belief Trajectory with Real-Time MPC

Once the experiment begins, the belief about the seeker pose is maintained via MCL:

$$b_t(x) = p(x_t^{\text{seek}} \mid z_{0:t}, u_{0:t-1}).$$

Simultaneously, the target is estimated using a Kalman filter.

Let $\hat{x}_t^{\text{seek}}$ and P_t^{seek} denote the filtered mean and covariance of the seeker at time t. The offline trajectory provides:

$$(\bar{x}_k, \Sigma_k) \quad \text{for } k = 0, \ldots, N_{\text{off}}.$$

At each MPC cycle:

1. The controller compares the online belief $(\hat{x}_t, P_t)$ to the offline predicted belief $(\bar{x}_{k(t)}, \Sigma_{k(t)})$.
2. The MPC cost includes a belief-tracking term:

$$\ell_{\text{belief}} = (\hat{x}_t - \bar{x}_{k(t)})^\top Q_b (\hat{x}_t - \bar{x}_{k(t)}).$$

3. The uncertainty Σ_k shapes the online MPC constraints:

$$r_{\text{inflated}} = r_0 + k_\sigma \sqrt{\lambda_{\max}(\Sigma_k)},$$

   where r_{inflated} is the current safe obstacle buffer.

4. If MCL yields lower covariance ($P_t \ll \Sigma_k$), the MPC automatically becomes more aggressive.
5. If MCL uncertainty grows, the MPC reverts to safer velocities and larger buffers.

Thus, the offline trajectory acts as a prior over feasible robot motion, while the online MPC adaptively re-optimizes the control inputs in real time based on updated belief estimates.

10 Algorithm: Offline-to-Online Hybrid Planning

11 ROS2 Integration with Probabilistic Planner

The provided ROS2 node integrates the offline trajectory through:

- calls to `plan_probabilistic_trajectory()` which returns waypoints with their covariances,
- publishing noisy (simulated) pose estimates,
- a waypoint-level controller that moves the robot along the nominal offline path while respecting predicted uncertainty.

Algorithm 3 Hybrid Offline Probabilistic Planning + Online MPC

1: **Offline Phase:**
2: Generate nominal controls $\{u_k\}_{k=0}^{N_{\text{off}}}$
3: Propagate states $\bar{x}_{k+1} = f(\bar{x}_k, u_k)$
4: Propagate covariance $\Sigma_{k+1} = A_k \Sigma_k A_k^\top + Q_k$
5: Store $(\bar{x}_k, \Sigma_k)$

6: **Online Phase (Real-Time Loop):**
7: **while** robot active **do**
8: MCL update $\Rightarrow (\hat{x}_t^{\text{seek}}, P_t^{\text{seek}})$
9: KF update $\Rightarrow (\hat{x}_t^{\text{tgt}}, P_t^{\text{tgt}})$
10: Determine nearest offline waypoint $(\bar{x}_{k(t)}, \Sigma_{k(t)})$
11: Inflate obstacles using $\Sigma_{k(t)}$ or P_t^{seek}
12: Formulate MPC with:

- belief-tracking cost,
- target interception cost,
- uncertainty-dependent constraints.

13: Solve MPC $\Rightarrow u_t^*$
14: Apply u_t^* to seeker
15: **end while**

In the full system, this node is replaced or augmented by:

1. an MCL node producing online beliefs,
2. a target-tracker node producing target beliefs,
3. an MPC node that re-plans dynamically,
4. but retains the offline waypoint covariance sequence as an MPC warm start.

12 ROS2 Implementation Sketch (Python)

This section provides minimal ROS2 node skeletons in Python using `rclpy`. They are intended as starting points only.

12.1 Seeker State Subscriber (using AMCL)

We assume the standard `amcl` node publishes `/amcl_pose` of type `geometry_msgs/msg/PoseWithCovarianceStamped`.

Listing 1: seeker_state_node.py

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import PoseWithCovarianceStamped
from geometry_msgs.msg import Twist

import numpy as np
```

```python
class SeekerStateNode(Node):
    def __init__(self):
        super().__init__('seeker_state_node')
        self.pose_sub = self.create_subscription(
            PoseWithCovarianceStamped,
            '/amcl_pose',
            self.pose_callback,
            10
        )
        self.cmd_pub = self.create_publisher(Twist, '/cmd_vel', 10)
        self.latest_pose = None
        self.latest_cov = None

    def pose_callback(self, msg: PoseWithCovarianceStamped):
        self.latest_pose = msg.pose.pose
        self.latest_cov = np.array(msg.pose.covariance).reshape((6, 6))
        # For debugging:
        # self.get_logger().info(f"Pose: {self.latest_pose.position.x}, {self.
            latest_pose.position.y}")

def main(args=None):
    rclpy.init(args=args)
    node = SeekerStateNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

12.2 Target Kalman Filter Node (Skeleton)

Listing 2: target_kf_node.py

```python
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import PoseWithCovarianceStamped
from geometry_msgs.msg import PoseStamped
import numpy as np

class TargetKFNode(Node):
    def __init__(self):
        super().__init__('target_kf_node')

        # state: [px, py, vx, vy]^T
        self.x = np.zeros((4, 1))
        self.P = np.eye(4) * 1.0

        self.dt = 0.1

        self.A = np.array([
```

```
            [1.0, 0.0, self.dt, 0.0],
            [0.0, 1.0, 0.0, self.dt],
            [0.0, 0.0, 1.0, 0.0],
            [0.0, 0.0, 0.0, 1.0],
        ])

        self.Q = np.eye(4) * 0.01

        self.H = np.array([
            [1.0, 0.0, 0.0, 0.0],
            [0.0, 1.0, 0.0, 0.0],
        ])

        self.R = np.eye(2) * 0.05

        self.meas_sub = self.create_subscription(
            PoseStamped,
            '/target_pose_measurement',
            self.measurement_callback,
            10
        )

        self.est_pub = self.create_publisher(
            PoseWithCovarianceStamped,
            '/target_estimate',
            10
        )

        self.timer = self.create_timer(self.dt, self.timer_callback)

    def measurement_callback(self, msg: PoseStamped):
        z = np.array([[msg.pose.position.x],
                      [msg.pose.position.y]])
        # Update step
        x_pred = self.A @ self.x
        P_pred = self.A @ self.P @ self.A.T + self.Q

        S = self.H @ P_pred @ self.H.T + self.R
        K = P_pred @ self.H.T @ np.linalg.inv(S)

        self.x = x_pred + K @ (z - self.H @ x_pred)
        self.P = (np.eye(4) - K @ self.H) @ P_pred

    def timer_callback(self):
        # Prediction-only if no new measurement
        self.x = self.A @ self.x
        self.P = self.A @ self.P @ self.A.T + self.Q

        msg = PoseWithCovarianceStamped()
        msg.header.stamp = self.get_clock().now().to_msg()
        msg.header.frame_id = 'map'
        msg.pose.pose.position.x = float(self.x[0])
        msg.pose.pose.position.y = float(self.x[1])
        # Orientation unused here
```

```
        cov = np.zeros((6, 6))
        cov[0, 0] = self.P[0, 0]
        cov[1, 1] = self.P[1, 1]
        msg.pose.covariance = cov.flatten().tolist()
        self.est_pub.publish(msg)

def main(args=None):
    rclpy.init(args=args)
    node = TargetKFNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

12.3 MPC Node Skeleton

This node subscribes to seeker and target estimates and publishes a velocity command. The actual optimization (e.g. using `cvxpy`) is represented as a placeholder function.

Listing 3: mpc_node.py (skeleton)

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import PoseWithCovarianceStamped, Twist
import numpy as np

class MPCNode(Node):
    def __init__(self):
        super().__init__('mpc_node')

        self.seeker_sub = self.create_subscription(
            PoseWithCovarianceStamped,
            '/amcl_pose',
            self.seeker_callback,
            10
        )

        self.target_sub = self.create_subscription(
            PoseWithCovarianceStamped,
            '/target_estimate',
            self.target_callback,
            10
        )

        self.cmd_pub = self.create_publisher(Twist, '/cmd_vel', 10)

        self.seeker_pose = None
        self.seeker_cov = None
        self.target_pose = None
        self.target_cov = None
```

```python
        self.dt = 0.1
        self.N = 15  # MPC horizon

        self.timer = self.create_timer(self.dt, self.timer_callback)

    def seeker_callback(self, msg: PoseWithCovarianceStamped):
        self.seeker_pose = msg.pose.pose
        self.seeker_cov = np.array(msg.pose.covariance).reshape((6, 6))

    def target_callback(self, msg: PoseWithCovarianceStamped):
        self.target_pose = msg.pose.pose
        self.target_cov = np.array(msg.pose.covariance).reshape((6, 6))

    def timer_callback(self):
        if self.seeker_pose is None or self.target_pose is None:
            return

        # Build initial seeker state x0
        x0 = np.zeros((4, 1))
        x0[0, 0] = self.seeker_pose.position.x
        x0[1, 0] = self.seeker_pose.position.y
        # TODO: extract yaw from quaternion
        # x0[2, 0] = theta
        # x0[3, 0] = v

        # Build target prediction sequence (constant position for now)
        target_seq = []
        px_tgt = self.target_pose.position.x
        py_tgt = self.target_pose.position.y
        for k in range(self.N + 1):
            target_seq.append(np.array([px_tgt, py_tgt]))

        # TODO: compute inflated obstacle radii from seeker_cov, target_cov
        # obstacles = [...]

        # Solve MPC optimization (placeholder)
        v_cmd, w_cmd = self.solve_mpc(x0, target_seq)

        twist = Twist()
        twist.linear.x = float(v_cmd)
        twist.angular.z = float(w_cmd)
        self.cmd_pub.publish(twist)

    def solve_mpc(self, x0, target_seq):
        # Placeholder MPC solver:
        # For now, simply drive straight towards the target.
        px = x0[0, 0]
        py = x0[1, 0]
        tgt = target_seq[0]
        dx = tgt[0] - px
        dy = tgt[1] - py
        dist = np.hypot(dx, dy)
        v_cmd = min(0.5, dist)  # saturate speed
        w_cmd = 0.0
```

```
        return v_cmd, w_cmd

def main(args=None):
    rclpy.init(args=args)
    node = MPCNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```