Autonomous Robotics: EKF and SLAM

Arwed Paul Meinert

December 10, 2024

1. Theory

The first part of the report is a theoretical explanation of the Extended Kalman Filter (EKF) and its applications.

1.1 Extended Kalman Filter

To use an EKF, the state of the system must be defined. When the goal is only to locate the object in a solution space, the state usually consists of the robot configuration. When using a mobile robot, that means the position as well as the angle. When the goal is also to map the environment using a Simultaneous Localization and Mapping (SLAM) algorithm, in the state variable are also the locations of known position as can be seen in (1.1). In this case, it is important to know the correct amount of anchor locations that are known to be stationary.

$$\mathbf{x} = \begin{bmatrix} x_r \\ y_r \\ \theta_r \\ x_1 \\ y_1 \\ n_1 \\ x_2 \\ y_2 \\ n2 \\ \vdots \\ x_n \\ y_n \\ n_n \end{bmatrix} \tag{1.1}$$

When using EKF, the initial position and orientation of the robot need to be known with some uncertainty. This is not necessary for the anchor positions since their position is being set the first time the robot detects the object. The uncertainty of the robot position as well as the uncertainty of the anchor positions is saved in the covariance matrix. This defines the standard deviation from the actual position to the expected position and is updated in the EKF algorithm to increase the certainty of the current position or to decrease it.

The initial state vector as well as the covariance matrix are passed to the EKF filter. There, a new prediction of the state vector is made that is based on the motion model of the robot. After that, the measurement function calculates the expected sensor data of the robot based on the new position. It then compares the expected sensor data with the actual data and a kalman gain is being calculated. When the deviation of the expected measurement and the actual measurement is large, the kalman gain is low resulting in a greater uncertainty in the covariance matrix. if the expected measurement and the real measurements match up, the uncertainty decreases and the pose of the robot as well as the location of the anchors can be determined more accurately.

When implementing a SLAM algorithm, the initial position of the anchor positions can be set to an arbitrary value. It is however important to keep track if each anchor already has been encountered. In this case, the initial position needs to be calculated using the first measurement and written to the correct position of the state vector. This is important because in the kalman filter, an initial position needs to be known. If the anchor positions are not initialized with the first best guess, they need to move towards their actual position during several iterations and severely decrease the accuracy of the robots position.

In the motion model, new expected values of the state vector (1.1) need to be calculated based on the control inputs of the robot. For linear systems, this is relatively simple. If the motion model contains non-linear equations, a linearization using the Jacobians need to be performed such as the example in (2.1). The Jacobian matrix maps the non-linear equation to linear equations using Taylor polynomials (1.3). This works well for short time steps since then the error is minimal. When no motion is expected (e.G. when having stationary anchors) the position of those elements stays the same in the prediction step.

$$\mathbf{x}_{t+1} = \begin{bmatrix} x_r + v\Delta t\cos(\theta_r) \\ y_r + v\Delta t\sin(\theta_r) \\ \theta_r + \omega\Delta t \\ x_a \\ y_a \end{bmatrix} \tag{1.2}$$

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & -v\Delta t\sin(\theta_r) & 0 & 0 \\ 0 & 1 & v\Delta t\cos(\theta_r) & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.3}$$

When dealing with non linear equations in the correction step, the values also need to be linearized after the expected sensor outputs have been computed. This is similar to Eq. (2.1) for the sensor mapping and Eq. (1.3) for the linearization. In addition, the uncertainty of the sensors is taken into account.

Finally, the state equation and the covariance matrix is updated and returned. This describes the current estimate of the robots location and orientation as well as the anchor point positions as well as the uncertainty. The complete structure and the steps of the EKF can be seen in Fig. 1.1.

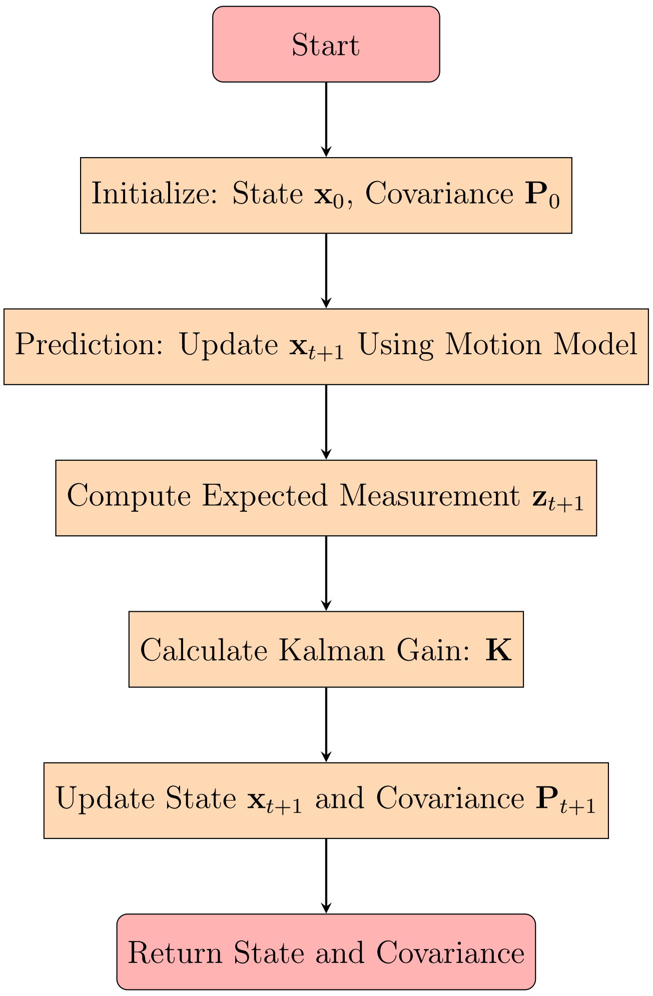
Start
Initialize: State $\mathbf{x}_0$, Covariance $\mathbf{P}_0$
Prediction: Update $\mathbf{x}_{t+1}$ Using Motion Model
Compute Expected Measurement $\mathbf{z}_{t+1}$
Calculate Kalman Gain: $\mathbf{K}$
Update State $\mathbf{x}_{t+1}$ and Covariance $\mathbf{P}_{t+1}$
Return State and Covariance

Figure 1.1: Structure of an EKF filter.

While the EKF is effective for estimating robot position using various sensor signals, combining the robot's position with anchor locations in the SLAM algorithm can lead to errors, particularly when the uncertainties in the motion model and sensors are large. The motion model's inaccuracies can propagate through the filter, affecting both the robot's pose and the map accuracy.

The EKF requires an initial estimate of the robot's position. Without this, the deviation would be infinite. Additionally, the EKF is best suited for small changes in the state vector. If there are large changes, such as in the kidnapped robot problem, the EKF struggles to accurately update the new position.

Another challenge arises with the known correspondences of anchor points. Since anchor positions are predefined in the state vector, any discrepancy in the order of measurements can cause data association issues. If an anchor is incorrectly matched to a sensor measurement, the update step can introduce significant errors.

Additionally, as the number of anchors increases, the state vector grows, leading to higher computational costs. The EKF also relies on linearization, which can introduce errors when dealing with highly nonlinear models, especially over long prediction intervals.

In summary, while the EKF works well for integrating motion and sensor data, its performance in SLAM depends on the accuracy of the models, proper data association, and computational efficiency.

1.2 Application

The EKF SLAM algorithm works without satellite-based systems like The Global Positioning System (GPS) or The Globalnaja Nawigazionnaja Sputnikowaja Sistema (GLONASS), making it ideal for indoor localization and fast position estimation. In [1], this approach was used to map and localize a drone in indoor environments with various sensors.

The researchers in [1] followed the outlined structure shown in Fig. 1.1. For the state equations describing the drone's position, they used the center position of the quadcopter ${}^{N}\rho$ relative to an inertial frame fixed to the Earth's surface, the velocity ${}^{B}v$, the angles of rotation around all three axes ${}^{B}\Lambda$, and the angular velocity ${}^{B}\omega$. Each of these states has three components. The motion model describes the new state based on velocities and forces acting on the robot, including gravitational and thrust forces. The latter depends on the drone's mass, moment of inertia, and the intensity of thrust generated by the rotors.

Since the equations in the motion model are nonlinear, the researchers applied Differential Flatness (DF) for linearization, including second derivatives to minimize errors. The motion model was then used to compute both expected and actual velocities based on the flight controller's output.

For the actual EKF SLAM, the drone is equipped with a 3D Light Detection and Ranging (LiDAR) sensor to provide the distances and angles to different landmarks. It also has an Inertia Measurement Unit (IMU) to provide the angles of the drone in all three axis and the angular accelerations.

Because the SLAM algorithm estimates both the drone's pose and the

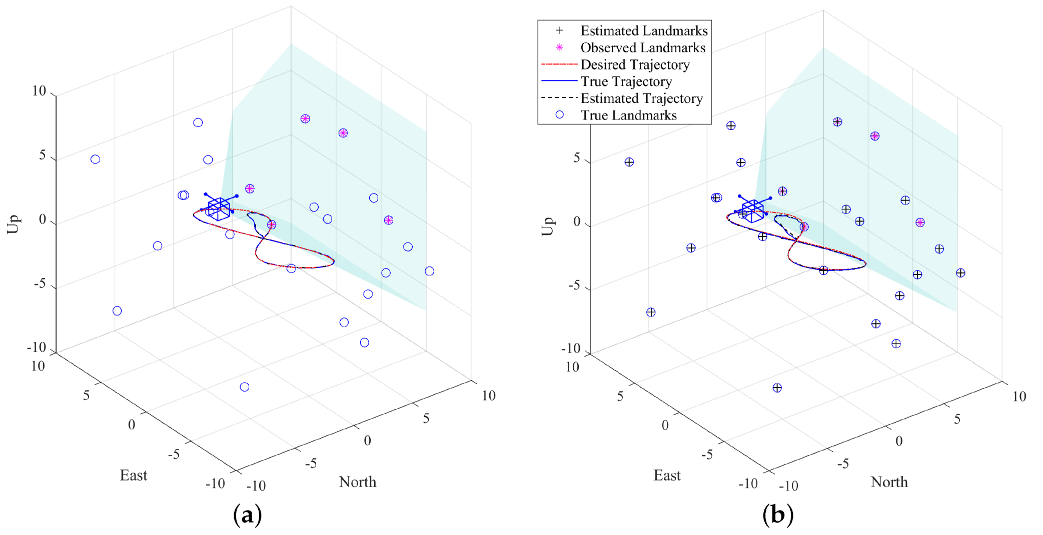

Figure 1.2: Drone flight path and known (a) and unknown (b) landmarks used for testing. [1]

environment map, the state vector was extended to include landmark coordinates. New landmarks were added to the state vector upon detection, initialized using an inverse measurement model. The covariance matrix was updated accordingly as the state vector changed.

The EKF SLAM model was tested in a simulation with 40 randomly generated landmarks. The drone followed a figure-eight trajectory for 40 seconds. Fig. 1.2 shows the environment and desired path. The drone successfully followed the trajectory with both known and unknown landmark positions. However, the Root Mean Square Error (RMSE) was lower with known landmarks (0.012 m) compared to unknown landmarks (0.04 m). With SLAM, the initial landmark error was within 3σ, decreasing over time to a final RMSE of 0.03 m.

The researchers also evaluated the computational cost of their implementation. The primary driver was the size of the state vector and the estimation step, with computational cost rising quadratically with the number of landmarks. This poses a challenge in maintaining a runtime under 0.1 s, required for the drone to autonomously follow its trajectory. Excessive computation time can result in overcorrection and inaccuracies due to model nonlinearities.

To address this, they proposed limiting the number of landmarks in the state vector to those currently visible, discarding obstructed ones. While this reduces computational time, it decreases accuracy as landmarks must be reinitialized upon reappearance. However, for large environments, this trade-off can help maintain real-time performance.

2. Implementation

This chapter describes the implementation of the EKF SLAM algorithm, as well as the Sequential Monte Carlo (SMC) method for localization. Both algorithms use the same outputs and work within the same simulation environment. The main program can be found in App. A.

2.1 Extended Kalman Filter

Outside of the EKF, the covariance matrix and the state vector are initialized. The initial uncertainty in the robot's position is very low. Since the initial position of the anchors is calculated the first time an anchor is detected, its accuracy is determined by the sensor errors. It can be assumed that this error is also relatively low.

The state vector combines the robot's position with the anchor positions. It is formatted the same way as in Eq. (1.1). Since the anchor positions are unknown at the beginning, they are initialized to -1 for all three components.

The EKF SLAM function takes as inputs the previous state vector μ, the covariance matrix, the control signals for velocity and rotational speed, as well as the sensor data and time step Δt.

Within the function, the noise covariance of the motion model and the noise matrix of the measurement model are defined. These matrices must match the form of the state vector and the measurement vector. Additionally, the sensor signal matrix is reshaped into a vector that has the same shape as the output of the measurement model. The initialization can be seen in Lst. 2.1.

```matlab
function [mu_new, cov_new] = EKFslam(mu, cov, speed, rotationspeed, signal, dt)

    %Comments and repetitive calculations (eg. the measurement Jacobian matrix) were
    %done by chat GPT
    % Noise matrices
    pNoise = diag([0.0001, 0.0001, 0.08,repmat([0.001],1,15)]);    % Process noise covariance
    distanceNoise = 4; % Noise for distance measurements
    angleNoise = deg2rad(10); % Noise for angle measurements (in radians)
    % Construct the measurement noise covariance matrix for 5 anchors (10 measurements)
    mVar = diag([repmat(distanceNoise, 1, 5), repmat(angleNoise, 1, 5)]);
    z = reshape(signal(:, 1:2)', [], 1);  % Create the vector for the measurements from the signal input
    % Initialize EKF
    filter = trackingEKF( ...
        @(state) myTransitionModel(state, speed,
           rotationspeed, dt,z), ...
        @(state) myMeasurementFunction(state), ...
        'ProcessNoise', pNoise, ...
        'MeasurementNoise', mVar, ...
        'StateTransitionJacobianFcn', @(state)
           myStateJacobian(state, speed, rotationspeed, dt),
           ...
        'MeasurementJacobianFcn', @(state)
           myMeasurementJacobian(state));
    initialize(filter,mu,cov);

    predict(filter); % Prediction

    correct(filter,z);  % Correct the predicted state using
       the actual measurements
    %Update
    mu_new=filter.State;
    cov_new=filter.StateCovariance;
end
```

Listing 2.1: Kalman filter implementation.

After that, the filter object is initialized. For this implementation, the MATLAB trackingEKF toolbox is used. This provides a framework for the implementation and simplifies the understanding of the actual algorithm. During initialization, the transition model, the measurement model, as well as the Jacobians and the noise matrices, are defined. This step can be seen in Lst. 2.1.

```
function statePred = myTransitionModel(mu, speed,
   rotationspeed, dt,z)
    statePred=mu;
    num_anchors=5;
    robot_dim = 3;
    if mu(robot_dim+3)==-1 %Initialize the anchor positions
        for i=1:num_anchors
            statePred(robot_dim+i*3-2)=mu(1)+z(i*2-1)*cos(mu
               (3)+z(i*2));
            statePred(robot_dim+i*3-1)=mu(2)+z(i*2-1)*sin(mu
               (3)+z(i*2));
            statePred(robot_dim+i*3)=i;
        end

    end
    x = mu(1);
    y = mu(2);
    theta = mu(3);

    % Predict new state
    theta_new = wrapToPi(theta + rotationspeed * dt);
    x_new = x + speed * cos(theta) * dt;
    y_new = y + speed * sin(theta) * dt;

    statePred(1:3) = [x_new; y_new; theta_new];
end
```

Listing 2.2: Transition model for the EKF

The object is initialized with the last state vector μ and the previous covariance matrix. After that, the new state vector is predicted using the motion model and the Jacobians defined during the initialization of the filter. The new states are written to the filter itself and can be accessed.

After the prediction step, the filter corrects itself using the correction method. Here, the measurement vector z is passed. Based on the difference between the actual measurements and the expected measurements, the state vector and the covariance matrix are updated.

Lastly, the new state and the covariance matrix are returned. In the calling function, these values are then used to display the estimated position of the robot and the anchors. In the next iteration, the new state vector and new measurements are passed again to this function.

In the initialization of the filter, the transition function is defined. This function is used to predict the next state and include the motion model. The function can be seen in Lst. 2.1.

First, the last state is copied to the predicted state, which will be the output of the function. This ensures that the dimensions stay the same and reallocates memory. In addition, the anchor positions are copied. This is part of the motion model since it is expected that the positions of the anchors remain constant.

After that, the function checks if the iterator for the anchor positions is -1. This would mean that the anchor positions have not yet been initialized. If this is the case, the expected positions are written to the state vector using the measurements. This is only done in the first iteration of the EKF, as in all subsequent iterations, the positions are updated based on the measurement function.

Finally, the new positions of the robot are calculated using the motion model. The predicted positions are written to the correct positions in the state vector. The motion model of the robot can be seen in Eq. (2.1).

$$\mathbf{x}_{\text{pred}} = \begin{bmatrix} x + v\cos(\theta)\Delta t \\ y + v\sin(\theta)\Delta t \\ \text{wrapToPi}(\theta + \omega\Delta t) \end{bmatrix} \tag{2.1}$$

Since the equations shown in Eq. (2.1) are non-linear, they need to be linearized for the EKF to work. This linearization step is defined during the initialization of the filter (Lst. 2.1). The function can be seen in Lst. 2.1.

```matlab
function F = myStateJacobian(mu, speed, rotationspeed, dt)
    theta = mu(3);
    F = eye(length(mu)); % Identity matrix for all state
        variables

    % Update the first three rows for robot motion
    F(1, 3) = -speed * sin(theta) * dt;
    F(2, 3) = speed * cos(theta) * dt;
end
```

Listing 2.3: Jacobian matrix for the motion model of the EKF.

The Jacobian matrix is obtained by forming the partial derivative of every combination of inputs, as can be seen in Eq. (1.3). Since the position of the anchors remains constant, this part of the Jacobian is the identity matrix. The only positions that are affected are $\frac{\partial x}{\partial \theta_{\text{pred}}}$ and $\frac{\partial y}{\partial \theta_{\text{pred}}}$. Therefore, in Lst. 2.1, the Jacobian is initialized with the identity matrix, and only the two relevant rows are updated.

$$\mathbf{F} = \frac{\partial \mathbf{x}_{\text{pred}}}{\partial \mu} = \begin{bmatrix} \frac{\partial x}{\partial x_{\text{pred}}} & \frac{\partial x}{\partial y_{\text{pred}}} & \frac{\partial x}{\partial \theta_{\text{pred}}} & 0 & \cdots & 0 \\ \frac{\partial y}{\partial x_{\text{pred}}} & \frac{\partial y}{\partial y_{\text{pred}}} & \frac{\partial y}{\partial \theta_{\text{pred}}} & 0 & \cdots & 0 \\ \frac{\partial \theta}{\partial x_{\text{pred}}} & \frac{\partial \theta}{\partial y_{\text{pred}}} & \frac{\partial \theta}{\partial \theta_{\text{pred}}} & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -v\sin(\mu_3)\Delta t & 0 & \cdots & 0 \\ 0 & 1 & v\cos(\mu_3)\Delta t & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \tag{2.2}$$

After predicting the robot's position for the defined time step, the sensors are used to correct the prediction. To do this, the measurement function is defined during the initialization of the EKF in Lst. 2.1. The measurement function uses the predicted state to generate the expected sensor data.

```matlab
function z_pred = myMeasurementFunction(state)
    num_anchors=5;
    robot_dim = 3;
    anchor_dim = 3;

    % Extract robot position and orientation
    robot_x = state(1);
    robot_y = state(2);
    robot_theta = state(3);

    % Preallocate measurement vector
    z_pred = zeros(num_anchors * 2, 1);

    % Compute relative measurements for anchors
    for i = 1:num_anchors
```

```
        anchor_idx = robot_dim + (i - 1) * anchor_dim + 1;
        anchor_x = state(anchor_idx);
        anchor_y = state(anchor_idx + 1);

        % Relative distance and orientation
        dx = anchor_x - robot_x;
        dy = anchor_y - robot_y;
        distance = sqrt(dx^2 + dy^2);
        orientation = wrapToPi(atan2(dy, dx) - robot_theta);

        % Add to measurement vector
        z_pred((i-1)*2+1:i*2) = [distance; orientation];
    end
end
```

Listing 2.4: Measurement function to map the predicted state to the sensor signals.

The implementation of the measurement function can be seen in Lst. 2.1. It takes only the state as input, since the actual comparison between the predicted sensor signals and the real ones is performed by the correction function in the EKF function, as shown in Lst. 2.1. It is important that the predicted measurement has the same form as the actual measurements:

$$\mathbf{z}_{\text{pred}} = \begin{bmatrix} d_1 \\ \theta_1 \\ \vdots \\ d_n \\ \theta_n \end{bmatrix} = \begin{bmatrix} \sqrt{(x_1 - x)^2 + (y_1 - y)^2} \\ \text{wrapToPi}\left(\arctan 2(y_1 - y, x_1 - x) - \theta\right) \\ \vdots \\ \sqrt{(x_n - x)^2 + (y_n - y)^2} \\ \text{wrapToPi}\left(\arctan 2(y_n - y, x_n - x) - \theta\right) \end{bmatrix} \tag{2.3}$$

Using the equations in Eq. (2.3), the expected signals can be generated. However, this function also involves non-linear operations. Therefore, during the EKF initialization, a function for the linearization of the measurements is provided. The function used for this can be seen in Lst. 2.1.

```
function H = myMeasurementJacobian(state)
    num_anchors=5;
    robot_dim = 3;
    anchor_dim = 2;

    % Extract robot position and orientation
    robot_x = state(1);
    robot_y = state(2);
    robot_theta = state(3);

    % Initialize measurement Jacobian
    H = zeros(num_anchors * 2, length(state));

    for i = 1:num_anchors
        anchor_idx = robot_dim + (i - 1) * anchor_dim + 1;
        anchor_x = state(anchor_idx);
        anchor_y = state(anchor_idx + 1);

        % Relative position
        dx = anchor_x - robot_x;
        dy = anchor_y - robot_y;
        q = dx^2 + dy^2;
        sqrt_q = sqrt(q);

        % Partial derivatives for distance
        H((i-1)*2+1, 1) = -dx / sqrt_q; % ddistance/drobot_x
        H((i-1)*2+1, 2) = -dy / sqrt_q; % ddistance/drobot_y
        H((i-1)*2+1, anchor_idx) = dx / sqrt_q; % ddistance/
            danchor_x
        H((i-1)*2+1, anchor_idx+1) = dy / sqrt_q; % ddistance
            /danchor_y

        % Partial derivatives for orientation
        H((i-1)*2+2, 1) = dy / q; % dorientation/drobot_x
        H((i-1)*2+2, 2) = -dx / q; % dorientation/drobot_y
        H((i-1)*2+2, 3) = -1; % dorientation/drobot_theta
        H((i-1)*2+2, anchor_idx) = -dy / q; % dorientation/
            danchor_x
        H((i-1)*2+2, anchor_idx+1) = dx / q; % dorientation/
            danchor_y
    end
end
```

Listing 2.5: Jacobian matrix of the measurement function.

This function creates the partial derivatives based on this Equations:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial z_{1,\mathrm{d}}}{\partial x} & \frac{\partial z_{1,\mathrm{d}}}{\partial y} & \frac{\partial z_{1,\mathrm{d}}}{\partial \theta} & \cdots & \frac{\partial z_{1,\mathrm{d}}}{\partial x_1} & \frac{\partial z_{1,\mathrm{d}}}{\partial y_1} & \cdots & 0 \\ \frac{\partial z_{1,\mathrm{o}}}{\partial x} & \frac{\partial z_{1,\mathrm{o}}}{\partial y} & \frac{\partial z_{1,\mathrm{o}}}{\partial \theta} & \cdots & \frac{\partial z_{1,\mathrm{o}}}{\partial x_1} & \frac{\partial z_{1,\mathrm{o}}}{\partial y_1} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_{n,\mathrm{d}}}{\partial x} & \frac{\partial z_{n,\mathrm{d}}}{\partial y} & \frac{\partial z_{n,\mathrm{d}}}{\partial \theta} & \cdots & \frac{\partial z_{n,\mathrm{d}}}{\partial x_n} & \frac{\partial z_{n,\mathrm{d}}}{\partial y_n} & \cdots & 0 \\ \frac{\partial z_{n,\mathrm{o}}}{\partial x} & \frac{\partial z_{n,\mathrm{o}}}{\partial y} & \frac{\partial z_{n,\mathrm{o}}}{\partial \theta} & \cdots & \frac{\partial z_{n,\mathrm{o}}}{\partial x_n} & \frac{\partial z_{n,\mathrm{o}}}{\partial y_n} & \cdots & 0 \end{bmatrix}. \tag{2.4}$$

The robot's position, the anchor positions, and the covariance matrix are updated using the measurement model and its Jacobians. The new state vector and covariance matrix are then used to display the robot's position in the simulation environment, as shown in Fig. 2.1. The complete implementation of the EKF SLAM algorithm can be seen in App. B.

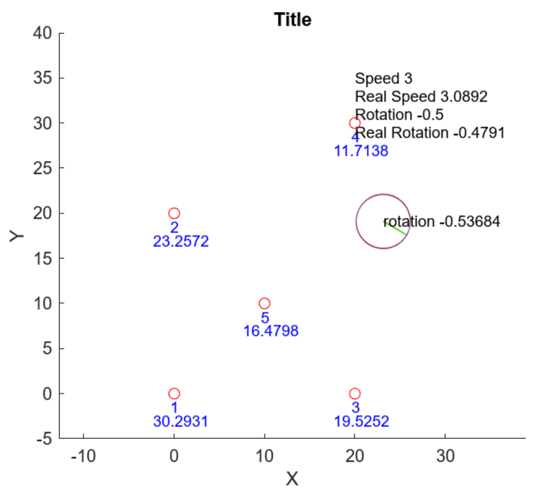

Figure 2.1: EKF SLAM algorithm using the simulated environment.

2.2 Sequential Monte Carlo Localization

The SMC method employs a randomly generated particle swarm to represent the state across all dimensions. This allows the robot's configuration and its environment to be described in multiple dimensions, similar to the EKF. The key difference, however, is that the EKF requires an accurate initial position estimate with known uncertainty, while the SMC method does not. Instead, it generates a large number of particles within the solution space, eliminating the need for a precise starting position. Nevertheless, providing an initial estimate can enhance the method's accuracy.

The SMC method is implemented using the same inputs and outputs as the EKF described in Sec. 2.1. In this approach, the particle swarm for the current iteration is initialized using the covariance matrix from the previous iteration. The output state corresponds to the particle that best matches the measurements. Other potential configurations are retained and utilized in subsequent iterations.

```
function [state,particles] = MCLlocalization(state_prev,
    particles_prev,speed,rotationspeed,signal,anchors,dt)
n=1000;
if size(particles_prev,1)<100 %no particles are initialised
     particles_prev=createRandom_Particles(n);
end

particles=ones(n,3);

deviation=0.5;
```

```
z = reshape(signal(:, 1:2)', [], 1);

state_pred=zeros(n,3);
prob=zeros(n,1);
for m=1:size(particles_prev,1)
    state_pred(m,:)=sample_motion_model(particles_prev(m,:),
        speed,rotationspeed,dt);
    prob(m)=measurement_model(state_pred(m,:),deviation,
        anchors,z);

end
%prob_norm=ones(1000,1);
prob_norm=prob/sum(prob);

for m=1:size(particles,1)
    [particles(m,:),prob(m)]=select_particle(prob_norm,
        state_pred);
end
[val, max_idx] = max(prob); % Get the index of the highest
   probability
state = state_pred(max_idx, :)'; % Return the particle with
   the highest probability
end
```

Listing 2.6: SMC method implementation.

The main function of the SMC is presented in Lst. 2.2. Initially, the number of particles, n, is defined. Next, the size of the particle swarm is verified. If the covariance matrix has been initialized by the main program (see App. A) at the start, it is overwritten, and random particles are generated within the solution space. Similar to the EKF implementation, the format of the measurement signal must align with that of the predicted measurements. Therefore, the simulated measurements are reshaped into a vector.

Each particle's new position is then estimated using the motion model and control inputs for speed and rotational speed. Additionally, the probability of each particle generating the observed sensor signals based on its predicted position is calculated and stored in a vector. Once all particles have been processed, the probabilities are normalized.

Using the normalized probabilities, a new particle swarm is created. The likelihood of a particle being selected corresponds to its normalized probability of representing the correct configuration based on the match between expected and actual sensor signals. This produces a new swarm with a higher density of particles near the actual position, while still retaining some particles in configurations that do not align with the sensor data. Retaining such particles is beneficial in scenarios involving significant inaccuracies, such as the "kidnapped robot problem," or when the expected position is incorrect.

Finally, the particle that best matches the expected and actual sensor data is returned as the robot's most probable position. This position can then be displayed in the simulation to represent the robot's location.

```matlab
function [predicted] = sample_motion_model(state,speed,
   rotationspeed,dt)
    predicted=state;
    x = state(1);
    y = state(2);
    theta = state(3);

    % Predict new state
    theta_new = wrapToPi(theta + rotationspeed * dt+randn*pi
        *0.03);
    x_new = x + speed * cos(theta) * dt+randn*0.3;
    y_new = y + speed * sin(theta) * dt+randn*0.3;

    predicted(:) = [x_new; y_new; theta_new];
end
```

Listing 2.7: SMC motion model.

The motion model used in the SMC method closely resembles the implementation described in Sec. 2.1. However, since the SMC method incorporates elements of a genetic algorithm, it includes a random component, as shown in Lst. 2.2. This random element introduces variability to the predicted values, simulating inaccuracies or deviations in the motion model compared to real-world behavior. Unlike the EKF, the SMC motion model does not require linearization.

```matlab
function [prob] = measurement_model(state,deviation,anchors,z
   )
    num_anchors=5;
    % Extract robot position and orientation
    robot_x = state(1);
    robot_y = state(2);
    robot_theta = state(3);

    % Preallocate measurement vector
    z_pred = zeros(num_anchors * 2, 1);

    % Compute relative measurements for anchors
    for i = 1:num_anchors
        % Index for anchor positions (x and y) in the anchors
            matrix
        anchor_x = anchors(i, 1);  % x-coordinate of the ith
            anchor
        anchor_y = anchors(i, 2);  % y-coordinate of the ith
            anchor

        % Calculate relative distance and orientation
        dx = anchor_x - robot_x;
        dy = anchor_y - robot_y;
        distance = sqrt(dx^2 + dy^2);
        orientation = wrapToPi(atan2(dy, dx) - robot_theta);

        % Add to measurement vector (z_pred)
        z_pred((i-1)*2+1:i*2) = [distance; orientation];
    end
    prob=prod((1/sqrt(2*pi*deviation^2)).*exp(-((z' - z_pred
       ').^2) / (2 * deviation^2)));
end
```

Listing 2.8: SMC measurement model.

The method for determining the probability that the predicted position produces the same sensor values as the actual sensor values is similar to the implementation in the EKF, as shown in Lst. 2.2. The key difference is that the function returns a probability indicating how likely it is that the predicted position corresponds to the measured sensor values. This probability is computed in the last line of Lst. 2.2 using Eq. (2.5):

$$p(z_i|\hat{z}_i) = \prod_{i=1}^{n} \left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_i - \hat{z}_i)^2}{2\sigma^2} \right) \right) \tag{2.5}$$

The closer the predicted sensor values match the actual sensor values, the higher the probability that a particle accurately represents the robot's pose. This increases its likelihood of being selected during the next iteration's selection phase.

The algorithm performs well for localization tasks. However, its effectiveness in a SLAM context is limited. This is due to the vast solution space created by the combination of the robot's position and anchor positions, which results in generally low probabilities. The likelihood of generating a particle that matches the real configuration is very small. Consequently, the simulated robot position tends to be highly inaccurate, as the most probable estimation fluctuates significantly.

The complete implementation of the SMC method for localization is provided in App. C, while the SLAM implementation can be found in App. D.

3. Acronyms

DF Differential Flatness. 4

EKF Extended Kalman Filter. 1, 3–5, 7, 9–16

GLONASS The Globalnaja Nawigazionnaja Sputnikowaja Sistema. 4

GPS The Global Positioning System. 4

IMU Inertia Measurement Unit. 4

LiDAR Light Detection and Ranging. 4

RMSE Root Mean Square Error. 5

SLAM Simultaneous Localization and Mapping. 1–5, 7, 12, 13, 16

SMC Sequential Monte Carlo. 7, 13–16

4. Bibliography

[1] S. Rauniyar, S. Bhalla, D. Choi, and D. Kim, "Ekf-slam for quadcopter using differential flatness-based lqr control," *Electronics*, vol. 12, no. 5, 2023, ISSN: 2079-9292. DOI: `10.3390/electronics12051113`. [Online]. Available: `https://www.mdpi.com/2079-9292/12/5/1113`.

A. Main Program

```matlab
clc
clear all
myapp = SPRob;
x = 20;
y = 20;
theta = 0;
speed = 3;
rotationspeed = -0.5;
cov_robot=0.0001;
cov_anchors=0.01;
cov = diag([repmat(cov_robot,1,3),repmat(cov_anchors,1,15)]);
myanchorsunit = [-1,-1,-1]; %x,y,number
mu = [x,y,theta,myanchorsunit,myanchorsunit,myanchorsunit,
    myanchorsunit,myanchorsunit]';

dt = 0.1;
myapp.dt=dt;
while(true)
%speed = 3.0;
myapp.setspeed(speed);
myapp.setrotationspeed(rotationspeed);
signal = myapp.getanchorssignal();
anchors=[0,0;0,20;20,0;20,30;10,10];
[mu, cov] = EKFslam(mu,cov,speed,rotationspeed,signal,myapp.
    dt);%Develop this function
%[mu, cov] = MCL(mu,cov,speed,rotationspeed,signal,myapp.dt)
    ;%Develop this function
%[mu, cov] = MCLlocalization(mu,cov,speed,rotationspeed,
    signal,anchors,myapp.dt);%Develop this function
myapp.updateyourAGV(mu(1,1),mu(2,1),mu(3,1));
%myanchors=anchors;
myanchors=[mu(4),mu(5);mu(7),mu(8);mu(10),mu(11);mu(13),mu
    (14);mu(16),mu(17)];

myapp.updateyouranchors(myanchors);

myapp.update;
pause(dt);
end
```

B. Extended Kalman Filter

```matlab
function [mu_new, cov_new] = EKFslam(mu, cov, speed,
   rotationspeed, signal, dt)

    %Comments and repetitive calculations (eg. the
       measurement Jacobian matrix) were
    %done by chat GPT
    % Noise matrices
    pNoise = diag([0.0001, 0.0001, 0.08,repmat([0.001],1,15)
       ]);    % Process noise covariance
    distanceNoise = 4; % Noise for distance measurements
    angleNoise = deg2rad(10); % Noise for angle measurements
       (in radians)
    % Construct the measurement noise covariance matrix for 5
        anchors (10 measurements)
    mVar = diag([repmat(distanceNoise, 1, 5), repmat(
       angleNoise, 1, 5)]);
    z = reshape(signal(:, 1:2)', [], 1);  % Create the vector
        for the measurements from the signal input
    % Initialize EKF
    filter = trackingEKF( ...
        @(state) myTransitionModel(state, speed,
           rotationspeed, dt,z), ...
        @(state) myMeasurementFunction(state), ...
        'ProcessNoise', pNoise, ...
        'MeasurementNoise', mVar, ...
        'StateTransitionJacobianFcn', @(state)
           myStateJacobian(state, speed, rotationspeed, dt),
           ...
        'MeasurementJacobianFcn', @(state)
           myMeasurementJacobian(state));
    initialize(filter,mu,cov);

    predict(filter); % Prediction

    correct(filter,z);  % Correct the predicted state using
       the actual measurements
    %Update
    mu_new=filter.State;
    cov_new=filter.StateCovariance;
end




function statePred = myTransitionModel(mu, speed,
   rotationspeed, dt,z)
    statePred=mu;
    num_anchors=5;
    robot_dim = 3;
    if mu(robot_dim+3)==-1 %Initialize the anchor positions
        for i=1:num_anchors
            statePred(robot_dim+i*3-2)=mu(1)+z(i*2-1)*cos(mu
                (3)+z(i*2));
            statePred(robot_dim+i*3-1)=mu(2)+z(i*2-1)*sin(mu
                (3)+z(i*2));
            statePred(robot_dim+i*3)=i;
        end

    end
    x = mu(1);
    y = mu(2);
    theta = mu(3);

    % Predict new state
    theta_new = wrapToPi(theta + rotationspeed * dt);
    x_new = x + speed * cos(theta) * dt;
    y_new = y + speed * sin(theta) * dt;

    statePred(1:3) = [x_new; y_new; theta_new];
end


function z_pred = myMeasurementFunction(state)
    num_anchors=5;
    robot_dim = 3;
    anchor_dim = 3;

    % Extract robot position and orientation
    robot_x = state(1);
    robot_y = state(2);
    robot_theta = state(3);

    % Preallocate measurement vector
    z_pred = zeros(num_anchors * 2, 1);

    % Compute relative measurements for anchors
    for i = 1:num_anchors
        anchor_idx = robot_dim + (i - 1) * anchor_dim + 1;
        anchor_x = state(anchor_idx);
        anchor_y = state(anchor_idx + 1);

        % Relative distance and orientation
        dx = anchor_x - robot_x;
        dy = anchor_y - robot_y;
        distance = sqrt(dx^2 + dy^2);
        orientation = wrapToPi(atan2(dy, dx) - robot_theta);

        % Add to measurement vector
        z_pred((i-1)*2+1:i*2) = [distance; orientation];
    end
    %z_pred
end

function F = myStateJacobian(mu, speed, rotationspeed, dt)
    theta = mu(3);
    F = eye(length(mu)); % Identity matrix for all state
        variables

    % Update the first three rows for robot motion
    F(1, 3) = -speed * sin(theta) * dt;
    F(2, 3) = speed * cos(theta) * dt;
end

function H = myMeasurementJacobian(state)
    num_anchors=5;
    robot_dim = 3;
    anchor_dim = 2;

    % Extract robot position and orientation
    robot_x = state(1);
    robot_y = state(2);
    robot_theta = state(3);

    % Initialize measurement Jacobian
    H = zeros(num_anchors * 2, length(state));

    for i = 1:num_anchors
        anchor_idx = robot_dim + (i - 1) * anchor_dim + 1;
        anchor_x = state(anchor_idx);
        anchor_y = state(anchor_idx + 1);

        % Relative position
        dx = anchor_x - robot_x;
        dy = anchor_y - robot_y;
        q = dx^2 + dy^2;
        sqrt_q = sqrt(q);

        % Partial derivatives for distance
        H((i-1)*2+1, 1) = -dx / sqrt_q; % distance/robot_x
        H((i-1)*2+1, 2) = -dy / sqrt_q; % distance/robot_y
        H((i-1)*2+1, anchor_idx) = dx / sqrt_q; %distance/
            anchor_x
        H((i-1)*2+1, anchor_idx+1) = dy / sqrt_q; % distance/
            anchor_y

        % Partial derivatives for orientation
        H((i-1)*2+2, 1) = dy / q; % orientation/robot_x
        H((i-1)*2+2, 2) = -dx / q; % orientation/robot_y
        H((i-1)*2+2, 3) = -1; % orientation/robot_theta
        H((i-1)*2+2, anchor_idx) = -dy / q; % orientation/anchor_x
        H((i-1)*2+2, anchor_idx+1) = dx / q; % orientation/anchor_y
    end
end
```

C. Sequential Monte Carlo Method Localization

```
function [state,particles] = MCLlocalization(state_prev,particles_prev,speed,rotationspeed,signal,anchors,dt)
n=1000;
if size(particles_prev,1)<100 %no particles are initialised
    particles_prev=createRandom_Particles(n);
end

particles=ones(n,3);

deviation=0.5;
z = reshape(signal(:, 1:2)', [], 1);

state_pred=zeros(n,3);
prob=zeros(n,1);
for m=1:size(particles_prev,1)
    state_pred(m,:)=sample_motion_model(particles_prev(m,:),speed,rotationspeed,dt);
    prob(m)=measurement_model(state_pred(m,:),deviation,anchors,z);

end
%prob_norm=ones(1000,1);
prob_norm=prob/sum(prob);

for m=1:size(particles,1)
    [particles(m,:),prob(m)]=select_particle(prob_norm,state_pred);
end
[val, max_idx] = max(prob); % Get the index of the highest probability
state = state_pred(max_idx, :)'; % Return the particle with the highest probability
end

function [selected_particle,prob] = select_particle(probabilities, particles)

    % Create the cumulative distribution function (CDF)
    cdf = cumsum(probabilities);

    % Generate a random number in the range [0, 1]
    r = rand();

    % Find the first particle whose CDF value exceeds the random number
    idx = find(cdf >= r, 1);

    % Select the corresponding particle
    selected_particle = particles(idx, :);
    prob=probabilities(idx);
end

function [particles]=createRandom_Particles(n)
    particles=ones(n,3);
    particles(:, 1:2) = rand(n, 2) * 35;
    particles(:, 3) = wrapToPi(rand(n, 1) * 2 * pi);
end

function [predicted] = sample_motion_model(state,speed,
   rotationspeed,dt)
    predicted=state;
    x = state(1);
    y = state(2);
    theta = state(3);

    % Predict new state
    theta_new = wrapToPi(theta + rotationspeed * dt+randn*pi
       *0.03);
    x_new = x + speed * cos(theta) * dt+randn*0.3;
    y_new = y + speed * sin(theta) * dt+randn*0.3;

    predicted(:) = [x_new; y_new; theta_new];
end

function [prob] = measurement_model(state,deviation,anchors,z
   )

    num_anchors=5;


    % Extract robot position and orientation
    robot_x = state(1);
    robot_y = state(2);
    robot_theta = state(3);

    % Preallocate measurement vector
    z_pred = zeros(num_anchors * 2, 1);

    % Compute relative measurements for anchors
    for i = 1:num_anchors
        % Index for anchor positions (x and y) in the anchors
           matrix
        anchor_x = anchors(i, 1);  % x-coordinate of the ith
           anchor
        anchor_y = anchors(i, 2);  % y-coordinate of the ith
            anchor

        % Calculate relative distance and orientation
        dx = anchor_x - robot_x;
        dy = anchor_y - robot_y;
        distance = sqrt(dx^2 + dy^2);
        orientation = wrapToPi(atan2(dy, dx) - robot_theta);

        % Add to measurement vector (z_pred)
        z_pred((i-1)*2+1:i*2) = [distance; orientation];
    end
    prob=prod((1/sqrt(2*pi*deviation^2)).*exp(-((z' - z_pred
        ').^2) / (2 * deviation^2)));
end
```

D. Sequential Monte Carlo Method SLAM

```
function [state,particles] = MCLlocalization(state_prev,
    particles_prev,speed,rotationspeed,signal,anchors,dt)
n=1000;
if size(particles_prev,1)<100 %no particles are initialised
     particles_prev=createRandom_Particles(n);
end

particles=ones(n,3);

deviation=0.5;
z = reshape(signal(:, 1:2)', [], 1);

state_pred=zeros(n,3);
prob=zeros(n,1);
for m=1:size(particles_prev,1)
     state_pred(m,:)=sample_motion_model(particles_prev(m,:),
         speed,rotationspeed,dt);
     prob(m)=measurement_model(state_pred(m,:),deviation,
         anchors,z);

end
%prob_norm=ones(1000,1);
prob_norm=prob/sum(prob);

for m=1:size(particles,1)
     [particles(m,:),prob(m)]=select_particle(prob_norm,
         state_pred);
end
[val, max_idx] = max(prob); % Get the index of the highest
    probability
state = state_pred(max_idx, :)'; % Return the particle with
    the highest probability
end

function [selected_particle,prob] = select_particle(
    probabilities, particles)

     % Create the cumulative distribution function (CDF)
     cdf = cumsum(probabilities);

     % Generate a random number in the range [0, 1]
     r = rand();

     % Find the first particle whose CDF value exceeds the
         random number
     idx = find(cdf >= r, 1);
```

```matlab

    % Select the corresponding particle
    selected_particle = particles(idx, :);
    prob=probabilities(idx);
end

function [particles]=createRandom_Particles(n)
    particles=ones(n,3);
    particles(:, 1:2) = rand(n, 2) * 35;
    particles(:, 3) = wrapToPi(rand(n, 1) * 2 * pi);
end

function [predicted] = sample_motion_model(state,speed,rotationspeed,dt)
    predicted=state;
    x = state(1);
    y = state(2);
    theta = state(3);

    % Predict new state
    theta_new = wrapToPi(theta + rotationspeed * dt+randn*pi*0.03);
    x_new = x + speed * cos(theta) * dt+randn*0.3;
    y_new = y + speed * sin(theta) * dt+randn*0.3;

    predicted(:) = [x_new; y_new; theta_new];
end

function [prob] = measurement_model(state,deviation,anchors,z)

    num_anchors=5;


    % Extract robot position and orientation
    robot_x = state(1);
    robot_y = state(2);
    robot_theta = state(3);

    % Preallocate measurement vector
    z_pred = zeros(num_anchors * 2, 1);

    % Compute relative measurements for anchors
    for i = 1:num_anchors
        % Index for anchor positions (x and y) in the anchors matrix
        anchor_x = anchors(i, 1);  % x-coordinate of the ith anchor
        anchor_y = anchors(i, 2);  % y-coordinate of the ith
            anchor

        % Calculate relative distance and orientation
        dx = anchor_x - robot_x;
        dy = anchor_y - robot_y;
        distance = sqrt(dx^2 + dy^2);
        orientation = wrapToPi(atan2(dy, dx) - robot_theta);

        % Add to measurement vector (z_pred)
        z_pred((i-1)*2+1:i*2) = [distance; orientation];
    end
    prob=prod((1/sqrt(2*pi*deviation^2)).*exp(-((z' - z_pred
        ').^2) / (2 * deviation^2)));
end
```