Autonomous Robotics: ROS Project 2

Arwed Paul Meinert

December 26, 2024

1. Task 1

1.1 AMCL Connection to ROS

The task is to explain the communication between Robot Operating System (ROS) and the navigation system running the Adaptive Monte-Carlo Localizer (AMCL) algorithm. The primary communication occurs in the function `processBag` when live data is not being used. However, the structure is similar to the live implementation. This function is called within the main loop. A partial implementation is provided in Lst. 1.1. While some functional aspects are omitted, all parts relevant to the communication are included.

In line 4, the log of previous messages is read. Subsequently, the important topics are defined in lines 7 and 8. In this case, the relevant topics are `/tf` and `basescan`. The previous messages are then filtered to identify those matching these topics.

The communication follows the Publisher-Subscriber (Pub/Sub) model. It establishes publishers to publish data to specific topics and subscribes to other topics to receive information as soon as it is published. This process is demonstrated in Lst. 1.1 in lines 10 and 11. The `advertise` function is used to create ROS publishers for the laser scan messages and the Transformation (TF) tree.

Afterward, the messages are iterated through. If a message belongs to one of the two topics, it is republished for other subscribers. If the connection to ROS fails, the program terminates (line 13). Additionally, if an unexpected message type is encountered, an error is raised (line 29). Once all messages have been processed, the log file is closed. The function does not return any value. The full function flow is illustrated in Fig. 1.1.

```
1 // Sample code of the cummunication was extracted using AI
2 void processBag(const std::string &bag_file, ros::NodeHandle
      &nh) {
3     // Open the bag file
4     rosbag::Bag bag;
5     bag.open(bag_file, rosbag::bagmode::Read);
6     // Topics to read from the bag file
7     std::vector<std::string> topics = {"/tf", "base_scan"};
8     rosbag::View view(bag, rosbag::TopicQuery(topics));
9     // Advertise publishers
10    ros::Publisher laser_pub = nh.advertise<sensor_msgs::
          LaserScan>("base_scan", 100);
11    ros::Publisher tf_pub = nh.advertise<tf2_msgs::TFMessage
          >("/tf", 100);
12    // Main loop to process messages
13    BOOST_FOREACH(rosbag::MessageInstance const msg, view) {
14        if (!ros::ok()) {
15            break;
```

```
        }
        // Handle TF messages
        tf2_msgs::TFMessage::ConstPtr tf_msg = msg.instantiate<tf2_msgs::TFMessage>();
        if (tf_msg) {
            tf_pub.publish(tf_msg);  // Republish TF message
            continue;
        }
        // Handle LaserScan messages
        sensor_msgs::LaserScan::ConstPtr scan_msg = msg.instantiate<sensor_msgs::LaserScan>();
        if (scan_msg) {
            laser_pub.publish(scan_msg);  // Republish LaserScan message
            continue;
        }
        ROS_WARN_STREAM("Unsupported message type: " << msg.getTopic());
    }
    // Close the bag file
    bag.close();
}
```

Listing 1.1: Communication between ROS and the AMCL program.

1.2 Particle Filter Code

The particle filter code is included via the header file in the AMCL implementation. The actual implementation resides in the `pf` file. Since this file only contains the function and object definitions utilized in the AMCL code, all necessary parameters for the particle filter are defined in the code shown in Lst. 1.2.

```
private_nh_.param("laser_min_range", laser_min_range_, -1.0);
  private_nh_.param("laser_max_range", laser_max_range_, -1.0);
  private_nh_.param("laser_max_beams", max_beams_, 30);
  private_nh_.param("min_particles", min_particles_, 100);
  private_nh_.param("max_particles", max_particles_, 5000);
  private_nh_.param("kld_err", pf_err_, 0.01);
  private_nh_.param("kld_z", pf_z_, 0.99);
  private_nh_.param("odom_alpha1", alpha1_, 0.2);
  private_nh_.param("odom_alpha2", alpha2_, 0.2);
  private_nh_.param("odom_alpha3", alpha3_, 0.2);
  private_nh_.param("odom_alpha4", alpha4_, 0.2);
  private_nh_.param("odom_alpha5", alpha5_, 0.2);
```

Listing 1.2: Parameters used in the AMCL filter.

The first three parameters define the validity of the laser sensor readings. If a value obtained from a sensor falls outside the range specified by `laser_min_range` or `laser_max_range`, it is considered invalid. The `max_beams` parameter specifies the number of laser beams used. If the sensor provides more beams, they are sampled.

The next four parameters configure the particle filter. First, the minimum and maximum number of particles are defined. Following this, the maximum allowable estimation errors are specified. Finally, the `odom_alpha` values represent the noise covariance of the motion model.

There are also parameters that define the sensor's measurement model, such as the probability of a detected sensor hit being valid or the likelihood of incorrect sensor readings.

These parameters are defined in a configuration file and are assigned to corresponding variables in the code. To modify these parameters, changes must be made to the configuration file.

Once all parameters are assigned, memory is allocated using the `pf_alloc` function, and the particle filter is initialized via the `pf_init` function. During initialization, all particles are created randomly.

When the first sensor readings are received, the particles are updated based on the sensor data. At this stage, probabilities are calculated for all particles. Afterward, the particles are resampled using the `pf_update_resample` function, which selects particles randomly with a bias toward those with higher probabilities. Finally, the filter object is updated using the resampled particles.

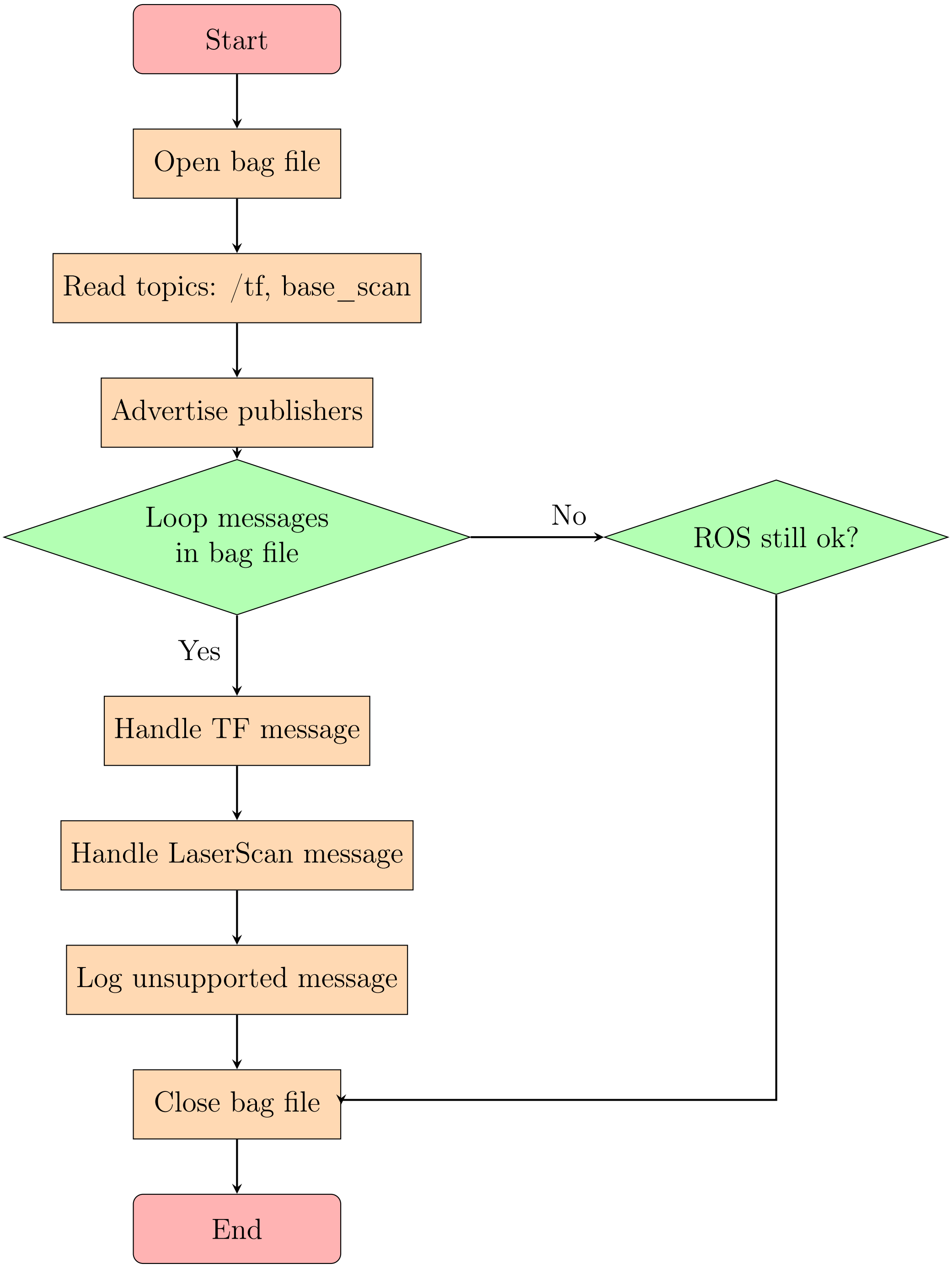

Figure 1.1: Flowchart of the communication program.

2. Motion and Measurement Model

2.1 Motion Model

The motion model is utilized in the `pf_update_action` function within the particle filter implementation. However, it is not directly implemented in this function. Instead, it references a function located in the `amcl_odom` file. This file provides implementations for various motion models tailored to different robots. The available models are:

- `ODOM_MODEL_OMNI`
- `ODOM_MODEL_DIFF`
- `ODOM_MODEL_OMNI_CORRECTED`
- `ODOM_MODEL_DIFF_CORRECTED`

The `OMNI` model represents omnidirectional robots, which can move in any direction without needing to turn. The `DIFF` model is designed for differential-drive robots, which must rotate before moving in a specified direction. The `CORRECTED` versions of these models account for potential errors, such as sensor drift or inaccuracies. Since the TurtleBot used in the ROS environment is a differential-drive robot, this section focuses on the measurement model `ODOM_MODEL_DIFF_CORRECTED`.

The implementation of the differential model is shown in Lst. 2.1. Both the sensor data and particle data are passed as pointers, allowing the function to directly modify the pose data. The function's only return value is `true`, which indicates successful execution.

The pose data for each particle and the movement delta (i.e., the distance moved) are stored in the `ndata` variable. This data is used to compute the new position of the robot based on the previous pose and the velocities.

To optimize performance, the function first checks whether the velocity data is below a defined threshold. If it is, the velocity is set to 0 to avoid unnecessary computations. If the robot is in motion, the translational and rotational differences are calculated (see Lst. 2.1, lines 21–23). Following this, the noise covariance is computed to account for potential wheel slippage in the motion model.

In the loop beginning at line 31, random noise is applied to each particle, and their positions are updated based on the velocities. Each particle is sampled and updated accordingly. Once all particles have been processed, the function returns `true` to signal that the update has been successfully completed.

```
bool AMCLOdom::UpdateAction(pf_t *pf, AMCLSensorData *data)
{
  AMCLOdomData *ndata;
  ndata = (AMCLOdomData*) data;

  // Compute the new sample poses
  pf_sample_set_t *set;

  set = pf->sets + pf->current_set;
  pf_vector_t old_pose = pf_vector_sub(ndata->pose, ndata->delta);
{
    // Implement sample_motion_odometry (Prob Rob p 136)
    double delta_rot1, delta_trans, delta_rot2;
    double delta_rot1_hat, delta_trans_hat, delta_rot2_hat;
    double delta_rot1_noise, delta_rot2_noise;
    // Avoid computing a bearing from two poses that are extremely near each
    // other (happens on in-place rotation).
    if(sqrt(ndata->delta.v[1]*ndata->delta.v[1] + ndata->delta.v[0]*ndata->delta.v[0]) < 0.01)
      delta_rot1 = 0.0;
    else
      delta_rot1 = angle_diff(atan2(ndata->delta.v[1], ndata->delta.v[0]),old_pose.v[2]);
    delta_trans = sqrt(ndata->delta.v[0]*ndata->delta.v[0] + ndata->delta.v[1]*ndata->delta.v[1]);
    delta_rot2 = angle_diff(ndata->delta.v[2], delta_rot1);

    // We want to treat backward and forward motion symmetrically for the
    // noise model to be applied below.  The standard model seems to assume
    // forward motion.
    delta_rot1_noise = std::min(fabs(angle_diff(delta_rot1,0.0)),fabs(angle_diff(delta_rot1,M_PI)));
    delta_rot2_noise = std::min(fabs(angle_diff(delta_rot2,0.0)),fabs(angle_diff(delta_rot2,M_PI)));

    for (int i = 0; i < set->sample_count; i++)
    {
      pf_sample_t* sample = set->samples + i;
      // Sample pose differences
      delta_rot1_hat = angle_diff(delta_rot1,
                                  pf_ran_gaussian(sqrt(this->alpha1*delta_rot1_noise*delta_rot1_noise+this->alpha2*delta_trans*
```

```
                                    delta_trans)));
      delta_trans_hat = delta_trans - pf_ran_gaussian(sqrt(
         this->alpha3*delta_trans*delta_trans +this->alpha4*
         delta_rot1_noise*delta_rot1_noise +this->alpha4*
         delta_rot2_noise*delta_rot2_noise));
      delta_rot2_hat = angle_diff(delta_rot2,pf_ran_gaussian(
         sqrt(this->alpha1*delta_rot2_noise*delta_rot2_noise
         +this->alpha2*delta_trans*delta_trans)));
      // Apply sampled update to particle pose
      sample->pose.v[0] += delta_trans_hat * cos(sample->pose
         .v[2] + delta_rot1_hat);
      sample->pose.v[1] += delta_trans_hat * sin(sample->pose
         .v[2] + delta_rot1_hat);
      sample->pose.v[2] += delta_rot1_hat + delta_rot2_hat;
    }
  }
    return true;
}
```

Listing 2.1: Motion model implementation in the particle filter.

2.2 Measurement Model

The measurement model is implemented in the `amcl_laser` file and referenced in the main function, similar to the motion model. Various models for different sensors are defined in this file. The beam model's measurement implementation is shown in Lst. 2.2. This function takes sensor data as input and returns the probability that the robot's current pose corresponds to the sensor readings. Here, the pose represents a single particle in the particle filter, which indicates a potential position and orientation of the robot.

The function consists of two nested `for` loops. The outer loop iterates through each particle representing a robot pose, while the inner loop processes each laser reading from the sensor. The distance to an object and the laser's bearing are stored in the variables `obs_range` and `obs_bearing`, respectively. The expected range at the specified bearing is then calculated using the `map_calc_range` function (see Lst. 2.2, line 28). This function returns the distance from the current pose to the nearest obstacle in the given direction, based on the map.

The overall probability is accumulated in the `pz` variable. The probability of each laser producing the observed sensor reading is added to this variable. If the actual range is smaller than the expected range, this suggests the presence of a moving object, such as a person. In such cases, the probability is modeled differently. Similarly, adjustments are made when the observed range equals the maximum range or when the range data is noisy.

Since the particle data passed to this function is provided via a pointer, the calculated probability for each particle is written directly into the particle's data. After all probabilities have been computed, they are added to a cumulative sum, which is returned by the function (see Lst. 2.2, line 52). This sum is later used in the calling function to normalize the probabilities.

```
1  double AMCLLaser::BeamModel(AMCLLaserData *data,
     pf_sample_set_t* set)
2  {
3    AMCLLaser *self;
4    int i, j, step;
5    double z, pz;
6    double p;
7    double map_range;
8    double obs_range, obs_bearing;
9    double total_weight;
10   pf_sample_t *sample;
11   pf_vector_t pose;
12   self = (AMCLLaser*) data->sensor;
13   total_weight = 0.0;
14   // Compute the sample weights
15   for (j = 0; j < set->sample_count; j++)
16   {
17     sample = set->samples + j;
18     pose = sample->pose;
19     // Take account of the laser pose relative to the robot
20     pose = pf_vector_coord_add(self->laser_pose, pose);
21     p = 1.0;
22     step = (data->range_count - 1) / (self->max_beams - 1);
23     for (i = 0; i < data->range_count; i += step)
24     {
25       obs_range = data->ranges[i][0];
26       obs_bearing = data->ranges[i][1];
27       // Compute the range according to the map
28       map_range = map_calc_range(self->map, pose.v[0], pose.v
           [1],
29                                  pose.v[2] + obs_bearing,
                                     data->range_max);
30       pz = 0.0;
31       // Part 1: good, but noisy, hit
32       z = obs_range - map_range;
33       pz += self->z_hit * exp(-(z * z) / (2 * self->sigma_hit
           * self->sigma_hit));
34       // Part 2: short reading from unexpected obstacle (e.g
           ., a person)
35       if(z < 0)
36         pz += self->z_short * self->lambda_short * exp(-self
```

```
            ->lambda_short*obs_range);
      // Part 3: Failure to detect obstacle, reported as max-
         range
      if(obs_range == data->range_max)
        pz += self->z_max * 1.0;
      // Part 4: Random measurements
      if(obs_range < data->range_max)
        pz += self->z_rand * 1.0/data->range_max;
      // TODO: outlier rejection for short readings
      assert(pz <= 1.0);
      assert(pz >= 0.0);
      //      p *= pz;
      // here we have an ad-hoc weighting scheme for
         combining beam probs
      // works well, though...
      p += pz*pz*pz;
    }
    sample->weight *= p;
    total_weight += sample->weight;
  }
  return(total_weight);
}
```

Listing 2.2: Measurement model implementation in the particle filter.

3. Comparison Between EKF and AMCL Implementation

The implementations of the Extended Kalman Filter (EKF) and AMCL share several similarities. Both programs use the same methods to interface with ROS, utilizing the Pub/Sub communication model. Additionally, both implementations employ a motion model to predict the robot's pose and a measurement model to verify whether the actual position corresponds to the estimated position.

The AMCL implementation appears to be more flexible, offering a wide range of configurable parameters and various motion models tailored for different types of robots. In contrast, the EKF implementation is limited to differential-drive robots, as reflected in its linearized motion model, shown in Lst. 3.

```
// Linearize control noise
    BFL::Matrix J(3,3);
    J(1,1)=-sin(filter->PostGet()->ExpectedValueGet()(3)+
        delta_rot1)*delta_trans;
    J(1,2)=cos(filter->PostGet()->ExpectedValueGet()(3)+
        delta_rot1);
    J(2,2)=0;

    J(2,1)=cos(filter->PostGet()->ExpectedValueGet()(3)+
        delta_rot1)*delta_trans;
    J(2,2)=sin(filter->PostGet()->ExpectedValueGet()(3)+
        delta_rot1);
    J(2,3)=0;

    J(3,1)=1;
    J(3,2)=0;
    J(3,3)=1;
```

Listing 3.1: Measurement model implementation in the ekf.

The same applies to the measurement model, implemented in the EKF as `LinearAnalytic-MeasurementModel-GaussianUncertainty`. In the AMCL implementation, the laser properties are highly configurable, allowing parameters such as the laser range and the number of beams to be adjusted. In contrast, the EKF implementation is designed for a specific model and sensor setup. While this limits flexibility, it simplifies the code, making it easier to manage and understand.

4. Mapping Implementation in MatLab

The implementation of the EKF closely resembles that of previous projects. In the prediction step, the motion model estimates the robot's new position. Subsequently, the map is generated by iterating through all cells and calculating the laser beam range to the nearest object. If a laser beam detects an object, the map is updated with a positive value. If no object is detected, a negative value is assigned. Cells not intersected by any laser beams remain unchanged.

Next, the measurement vector, consisting of the distances from 36 laser beams, is passed to the measurement function along with the updated map. The laser readings from the new position are simulated to generate a predicted measurement vector. The filter is then corrected based on the difference between the actual sensor readings and the predicted values. The updated state vector and map are returned and can be visualized.

The map produced by the script is shown in Fig. 4.1. Yellow squares represent positive values (detected objects), dark blue squares indicate negative values (no objects), and light blue squares correspond to areas not yet detected by the laser. The full EKF implementation code is provided in App. A.

The mapping function is called after the prediction step because the EKF only knows the estimated position at that point. Although the position is not as accurate as after the correction step, performing the mapping after the correction would lead to incorrect results in the first iteration, as the map has not been initialized. Additionally, the map's resolution is not high enough for the initial error to significantly impact the results.

```
function [map] = mapping(state, map, sensor, block_size, zmax
    )
    % Robot's state
    x = state(1);
    y = state(2);
    theta = state(3);
        for ll = 1:size(map, 1)
            for kk = 1:size(map, 2)
                % Compute likelihood for the cell
                %map(ll,kk) = inverse_sense_model(ll, kk,
                    block_size, [x; y; theta], zmax, sensor)
                map(ll, kk) = map(ll, kk) +
                    inverse_sense_model(ll, kk, block_size, [x
                    ; y; theta], zmax, sensor);
            end
        end
end
```

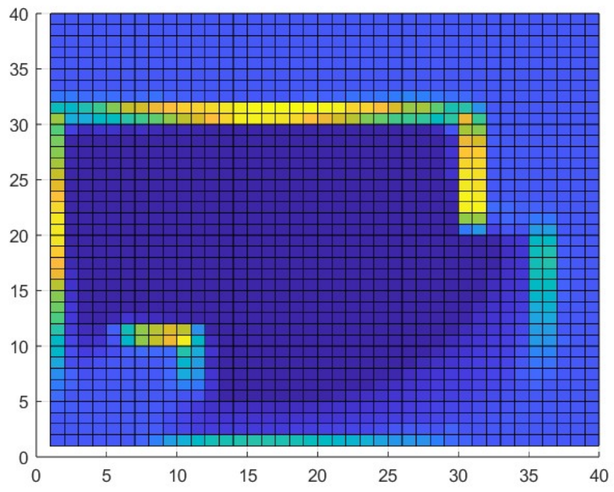

Figure 4.1: Map created by the Matlab script

Listing 4.1: Mapping implementation and calling of the `inverse_sense_model` function.

The mapping function is shown in Lst. 4. The function iterates through each cell in the map and calls the `inverse_sense_model`, which is provided. For each cell, the function determines the closest laser beam. It then checks whether the laser beam hits the cell with a distance less than the `zmax` value, indicating that the cell is occupied by an object. In this case, the function returns `1`, which is added to the corresponding cell in the map. If the laser beam passes through the cell or reaches the maximum distance, the cell is considered free, and `-0.2` is returned. If no laser beam passes through the cell, `0` is returned, indicating an unobserved cell.

This approach results in a high probability of detecting objects in cells near actual objects. Furthermore, if the sensor data is noisy, its impact on the final map is minimized, as multiple readings help to provide a more accurate result.

```
function [z_pred] = measurementFunction(state, sensor, map)
    num_beams = length(sensor); % Number of laser beams
    max_range = 20;             % Maximum sensor range
    beam_angle = 2 * pi / num_beams; % Angle between beams
    x = state(1);
    y = state(2);
    theta = state(3);
    z_pred = zeros(num_beams, 1);
```

```
9       % Iterate over each beam
10      for i = 1:num_beams
11          beam_theta = wrapToPi(theta + (i - 1) * beam_angle);
12          range = 0; % Start at the robot's position
13          while range < max_range
14              % Calculate the beam's endpoint in map coordinates
15              beam_x = x + range * cos(beam_theta);
16              beam_y = y + range * sin(beam_theta);
17              % Convert to map indices
18              map_x = round(beam_x);
19              map_y = round(beam_y);
20              % Check if the beam is out of bounds
21              if map_x < 1 || map_x > size(map, 1) || map_y < 1 || map_y > size(map, 2)
22                  range = max_range; % Beam reaches maximum range
23                  break;
24              end
25              % Check if the cell is occupied
26              if map(map_x, map_y) > 0.5 % Threshold for occupied cells
27                  break; % Obstacle detected
28              end
29              % Increment range
30              range = range + 0.1; % Step size for beam tracing
31          end
32          % Store the predicted range for this beam
33          z_pred(i) = min(range, max_range);
34      end
35  end
```

Listing 4.2: Measurement function for the correction step.

Since the new sensor input differs from the landmarks used in the previous project, the measurement function needs to be modified. The measurement vector consists of 36 distance readings, taken at 10° intervals. The implementation of the measurement function is shown in Lst. 4. This function generates the expected measurement vector for the new position.

For each laser beam, the function simulates a beam starting from the robot's position with an initial length of 0. It then calculates the map coordinates for the endpoint of the laser (Lst. 4, lines 15-19). The map is checked to see if the coordinates are occupied or if they fall outside the map bounds. If they are occupied or out of bounds, the next beam is simulated. If no occupied cell is found, the length of the laser beam is increased, and the check is repeated. This process continues until the maximum sensor range is reached. If the maximum range is reached, the sensor reading corresponds to the maximum laser range, and the next beam is processed.

After all laser beams have been checked and the results have been recorded in the expected measurement vector `z_pred`, it is compared to the actual measurement vector using the appropriate function in the main EKF script (App. A).

The robot's position remains accurate even after extended runtime. While the resulting map may be slightly offset, it effectively represents the portion of the environment visible to the robot.

5. Acronyms

AMCL Adaptive Monte-Carlo Localizer. 1, 2, 10

EKF Extended Kalman Filter. 10, 11, 14

Pub/Sub Publisher-Subscriber. 1, 10

ROS Robot Operating System. 1, 5, 10

TF Transformation. 1

5.1 Use of Generative AI

AI was used for spelling and grammar checks as well as for the creation of tables and debugging in the LaTeXsyntax. In the Matlab code, it was used for debugging and comments in the code.

A. EKF and Mapping Implementation

```
function [x,y,theta,cov,map] = GOMapping(x,y,theta,cov,speed,
    rotationspeed,scan,map,dt)
    % EKF Localization without landmarks

    % Noise matrices
    pNoise = diag([0.0001, 0.0001, 0.08]); % Process noise
        covariance
    state=[x,y,theta];
    % Initialize EKF
    filter = trackingEKF( ...
        @(state) transitionModel(state, speed, rotationspeed,
            dt), ...
        @(state) measurementFunction(state, scan, map), ...
        'ProcessNoise', pNoise, ...
        'MeasurementNoise', eye(length(scan)), ... %
            Placeholder for measurement noise
        'StateTransitionJacobianFcn', @(state) stateJacobian(
            state, speed, rotationspeed, dt));

    initialize(filter, state, cov);
    predict(filter); % Prediction
    map=mapping(state,map,scan,1,20);
    correct(filter, scan);
    % Since there are no landmarks, no correction step is
        applied.
    % Update
    filter.State
    x = filter.State(1);
    y = filter.State(2);
    theta = filter.State(3);
    cov = filter.StateCovariance;
end

function statePred = transitionModel(state, speed,
    rotationspeed, dt)
    % Transition model for robot motion
    % Predict new state
    x=state(1);
    y=state(2);
    theta=state(3);
    theta_new = wrapToPi(theta + rotationspeed * dt);
    x_new = x + speed * cos(theta) * dt;
    y_new = y + speed * sin(theta) * dt;

    statePred = [x_new; y_new; theta_new];
end
```

```matlab

function [z_pred] = measurementFunction(state, sensor, map)
    % Predicts sensor measurements based on the robot's state and the map.
    % Inputs:
    %   state - [x; y; theta], the robot's position and orientation.
    %   sensor - Array of sensor measurements (not used in prediction directly).
    %   map - 2D occupancy grid (values between 0 and 1).
    % Output:
    %   z_pred - Predicted sensor readings (same size as sensor).

    % Parameters
    num_beams = length(sensor); % Number of laser beams
    max_range = 20;             % Maximum sensor range
    beam_angle = 2 * pi / num_beams; % Angle between beams

    % Extract robot's state
    x = state(1);
    y = state(2);
    theta = state(3);

    % Initialize predicted sensor readings
    z_pred = zeros(num_beams, 1);

    % Iterate over each beam
    for i = 1:num_beams
        % Calculate angle of the beam relative to the robot
        beam_theta = wrapToPi(theta + (i - 1) * beam_angle);

        % Cast the beam in the map to find the first obstacle
        range = 0; % Start at the robot's position
        while range < max_range
            % Calculate the beam's endpoint in map coordinates
            beam_x = x + range * cos(beam_theta);
            beam_y = y + range * sin(beam_theta);

            % Convert to map indices
            map_x = round(beam_x);
            map_y = round(beam_y);

            % Check if the beam is out of bounds
            if map_x < 1 || map_x > size(map, 1) || map_y < 1 || map_y > size(map, 2)
                range = max_range; % Beam reaches maximum range
```

```
                break;
            end

            % Check if the cell is occupied
            if map(map_x, map_y) > 0.5 % Threshold for occupied cells
                break; % Obstacle detected
            end

            % Increment range
            range = range + 0.1; % Step size for beam tracing
        end

        % Store the predicted range for this beam
        z_pred(i) = min(range, max_range);
    end
end


function [map] = mapping(state, map, sensor, block_size, zmax)
    % Robot's state
    x = state(1);
    y = state(2);
    theta = state(3);
        for ll = 1:size(map, 1)
            for kk = 1:size(map, 2)
                % Compute likelihood for the cell
                %map(ll,kk) = inverse_sense_model(ll, kk, block_size, [x; y; theta], zmax, sensor)
                map(ll, kk) = map(ll, kk) + inverse_sense_model(ll, kk, block_size, [x; y; theta], zmax, sensor);
            end
        end
end

function [map] = mapping_no_inverse_model(state, map, sensor)
% Using a own implementation of the sensor module by exponantially adding
% probability to a cell if it was observed more often to be occupied
    % Update occupancy map using laser scanner data
    % Laser data contains 36 lines, each at a 10-degree offset

    % Parameters
    laser_range = 20; % Maximum range of the laser scanner
    map_size = 40; % Map dimensions (40x40)
```

```
    map_resolution = 1; % Resolution (1 unit per cell)
    x = state(1);
    y = state(2);
    theta = state(3);

    for i = 1:36
        % Compute angle for this sensor reading
        angle = wrapToPi(theta + deg2rad(10) * i);

        % Compute map indices for the detected obstacle
        x_map = ceil(x + cos(angle) * sensor(i)) + 1;
        y_map = ceil(y + sin(angle) * sensor(i)) + 1;

        % Check if indices are within map boundaries
        if x_map > 0 && x_map <= map_size && y_map > 0 && y_map <= map_size
            if sensor(i) < laser_range
                % Update for detected obstacle
                map(x_map, y_map) = map(x_map, y_map) + (1 - map(x_map, y_map)) * 0.1;
            end

            % Update for empty spaces along the laser path
            j = 1; % Start just before the detected obstacle
            while sensor(i) - j > 1
                x_empty = ceil(x + cos(angle) * (sensor(i) - j)) + 1;
                y_empty = ceil(y + sin(angle) * (sensor(i) - j)) + 1;

                if x_empty > 0 && x_empty <= map_size && y_empty > 0 && y_empty <= map_size
                    map(x_empty, y_empty) = map(x_empty, y_empty) - map(x_empty, y_empty) * 0.1;
                end

                j = j + 1;
            end
        end
    end
end


function F = stateJacobian(state, speed, rotationspeed, dt)
    % Jacobian of the state transition model
    theta = state(3);
    F = eye(3); % Identity matrix for the 3-state variables

```

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