EKF Slam 1.0

Introduction

This document's purpose is to document in a detailed manner our first and current EKF Slam implementation, it does not dive deeply into mathematical details so it is not a substitution for literature or Cyrill lectures (https://www.youtube.com/watch?v=DE6Jn2cB4J4&list=PLgnQpQtFTOGQrZ4O5QzbIHgl3b1JHimN_&index=7).
Since our system runs with ROS2, the code and the EKF SLAM were designed and written with that in mind.
The algorithm runs inside a ROS node more specifically LMNode:

```
LMNode::LMNode(ExtendedKalmanFilter *ekf, ConeMap *perception_map, MotionUpd
               ConeMap *track_map, VehicleState *vehicle_state, bool use_odo
    : Node("loc_map"),
      _ekf(ekf),
      _perception_map(perception_map),
      _motion_update(imu_update),
      _track_map(track_map),
      _vehicle_state(vehicle_state),
      _use_odometry(use_odometry) {
  this->_perception_subscription = this->create_subscription<custom_interfac
      "perception/cone_coordinates", 10,
      std::bind(&LMNode::_perception_subscription_callback, this, std::place
  this->_localization_publisher =
      this->create_publisher<custom_interfaces::msg::Pose>("vehicle_localiza
  this->_map_publisher = this->create_publisher<custom_interfaces::msg::Cone

  adapter = adapter_map[mode](this);
}
```

The constructor above helps us understand the purpose. Receive perception and localization sensors information and publish a pose and a track map. This is obviously not an easy task with a moving robot/car and therefore we need an algorithm to guide this proccess (localization and mapping algorithm) this is where the EKF SLAM comes in.

Implementation Details

As the name implies the EKF SLAM is based on an Extended Kalman Filter and it is common for these filters to be divided in 2 steps, prediction and correction step.

Prediction Step:

```
void ExtendedKalmanFilter::prediction_step(const MotionUpdate &motion_update
  std::chrono::time_point<std::chrono::high_resolution_clock> now =
      std::chrono::high_resolution_clock::now();
  double delta =
      std::chrono::duration_cast<std::chrono::microseconds>(now - this->_las
  this->_last_motion_update = motion_update;
  Eigen::VectorXf tempX = this->X;
  this->X = this->_motion_model.predict_expected_state(tempX, motion_update,
  Eigen::MatrixXf G =
      this->_motion_model.get_motion_to_state_matrix(tempX, motion_update, d
  Eigen::MatrixXf R = this->_motion_model.get_process_noise_covariance_matri
      this->X.size());  // Process Noise Matrix
  // this->P = G * this->P * G.transpose() + R;
  Eigen::MatrixXf P_dense = G * Eigen::MatrixXf(this->P) * G.transpose() + R
  this->P = P_dense.sparseView();
  this->_last_update = now;
}
```

Above is the prediction step code currently implemented. First we compute the predicted state based on the current state and the measurements stored in MotionUpdate:

```
struct MotionUpdate {
  double translational_velocity = 0.0;   /**< Meters per sec */
  double translational_velocity_x = 0.0; /**< Meters per sec */
  double translational_velocity_y = 0.0; /**< Meters per sec */
  double rotational_velocity = 0.0;      /**< Degrees per sec */
  double steering_angle = 0.0;           /**< Degrees */
  std::chrono::time_point<std::chrono::high_resolution_clock>
      last_update; /**< Timestamp of last update */
};
```

The new state state is predicted following the equations below:

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v_t}{w_t}\sin\theta + \frac{v_t}{w_t}\cos\theta \\ \frac{v_t}{w_t}\cos\theta - \frac{v_t}{w_t}\sin\theta \\ w_t\Delta t \end{pmatrix}$$

Our code follows those equations and adds a base case for extra porecision when we have no rotational velocity:

```
Eigen::VectorXf NormalVelocityModel::predict_expected_state(
    const Eigen::VectorXf &expected_state, const MotionUpdate &motion_predic
    const double time_interval) const {
  Eigen::VectorXf next_state = expected_state;
  if (motion_prediction_data.rotational_velocity == 0.0) {  // Rectilinear m
    next_state(0) +=
        motion_prediction_data.translational_velocity * cos(expected_state(2
    next_state(1) +=
        motion_prediction_data.translational_velocity * sin(expected_state(2
  } else {  // Curvilinear movement
    next_state(0) +=
        -(motion_prediction_data.translational_velocity /
          motion_prediction_data.rotational_velocity) *
            sin(expected_state(2)) +
        (motion_prediction_data.translational_velocity /
         motion_prediction_data.rotational_velocity) *
            sin(expected_state(2) + motion_prediction_data.rotational_velocit
    next_state(1) +=
        (motion_prediction_data.translational_velocity /
         motion_prediction_data.rotational_velocity) *
            cos(expected_state(2)) -
        (motion_prediction_data.translational_velocity /
         motion_prediction_data.rotational_velocity) *
            cos(expected_state(2) + motion_prediction_data.rotational_velocit
  }
  next_state(2) = normalize_angle(expected_state(2) +
                                  motion_prediction_data.rotational_velocity
  return next_state;
}
```

After that we just update the Kalman Filter matrices following the known EKF formula directly applied in code.

$$2: \quad \bar{\mu}_t = g(u_t, \mu_{t-1})$$
$$3: \quad \bar{\Sigma}_t = G_t \, \Sigma_{t-1} \, G_t^T + R_t$$

Correction Step:

```
void ExtendedKalmanFilter::correction_step(const ConeMap &perception_map) {
  for (auto cone : perception_map.map) {
    ObservationData observation_data = ObservationData(cone.first.x, cone.fi
    int landmark_index = this->discovery(observation_data);
    if (landmark_index == -1) {  // Too far away landmark
      continue;
    }
    Eigen::MatrixXf H = this->_observation_model.get_state_to_observation_ma
        this->X, landmark_index, this->X.size());
    Eigen::MatrixXf Q =
        this->_observation_model.get_observation_noise_covariance_matrix();


    Eigen::MatrixXf K = this->get_kalman_gain(H, this->P, Q);
    Eigen::Vector2f z_hat = this->_observation_model.observation_model(
        this->X, landmark_index);  // expected observation
    Eigen::Vector2f z =
        Eigen::Vector2f(observation_data.position.x,
                        observation_data.position.y);  // position of the la
    this->X = this->X + K * (z - z_hat);
    // TODO(PedroRomao3): divide into multiple calculation and measure time
    // TODO(PedroRomao3): Diagram.
    this->P =
        (Eigen::MatrixXf::Identity(this->P.rows(), this->P.cols()) - K * H)
  }
}
```

The **correction step** happens when we get information from our perception. The information we get from perception is basically the coordinates of cones seen in the car's referential. What our code does is for each cone seen it will first run a discovery step where it will match the cone to the most likely already mapped cone:

Discovery:

```
int ExtendedKalmanFilter::discovery(const ObservationData &observation_data)
  Eigen::Vector2f landmark_absolute =
      this->_observation_model.inverse_observation_model(this->X, observatio
  double distance =
      std::sqrt(pow(observation_data.position.x, 2) + pow(observation_data.p
  if (distance > ExtendedKalmanFilter::max_landmark_distance) {
    return -1;
  }
  double best_delta = 1000000000.0;
  int best_index = -1;
  for (int i = 3; i < this->X.size() - 1; i += 2) {
    double delta_x = this->X(i) - landmark_absolute(0);
    double delta_y = this->X(i + 1) - landmark_absolute(1);
    double delta = std::sqrt(std::pow(delta_x, 2) + std::pow(delta_y, 2));
    if (delta < best_delta && this->_colors[(i - 3) / 2] == observation_data
      best_index = i;
      best_delta = delta;
    }
  }
  if (best_index != -1) {
    double score =
        ExtendedKalmanFilter::cone_match(this->X(best_index), this->X(best_i
                                         landmark_absolute(0), landmark_absc
    if (score > 0) {
      return best_index;
    }
  }

  // If not found, add to the map
  this->X.conservativeResizeLike(Eigen::VectorXf::Zero(this->X.size() + 2));
  this->X(this->X.size() - 2) = landmark_absolute(0);
  this->X(this->X.size() - 1) = landmark_absolute(1);
  // this->P.conservativeResizeLike(Eigen::MatrixXf::Zero(this->P.rows() + 2

  // Create a new sparse matrix of the correct size
  Eigen::SparseMatrix<float> newP(this-> P.rows() + 2, this->P.cols() + 2);

  // Copy the values from P into newP
  for (int i = 0; i < this->P.rows(); i++) {
    for (int j = 0; j < this->P.cols(); j++) {
      newP.coeffRef(i, j) = this->P.coeff(i, j);
    }
  }

  // Replace P with newP
  this->P.swap(newP);

  this->_colors.push_back(observation_data.color);
```

```
    return this->X.size() - 2;
  }
```

Inside this step the first thing that is done is transforming the referential from the car to the track, this is done with the function below:

```
Eigen::Vector2f ObservationModel::inverse_observation_model(
    const Eigen::VectorXf &expected_state, const ObservationData &observatio
  Eigen::Matrix3f transformation_matrix = Eigen::Matrix3f::Identity();
  transformation_matrix(0, 0) = cos(expected_state(2));
  transformation_matrix(0, 1) = -sin(expected_state(2));
  transformation_matrix(0, 2) = expected_state(0);
  transformation_matrix(1, 0) = sin(expected_state(2));
  transformation_matrix(1, 1) = cos(expected_state(2));
  transformation_matrix(1, 2) = expected_state(1);
  Eigen::Vector3f observation =
      Eigen::Vector3f(observation_data.position.x, observation_data.position
  Eigen::Vector3f observed_landmark_absolute_position = transformation_matri

  return Eigen::Vector2f(observed_landmark_absolute_position(0),
                         observed_landmark_absolute_position(1));
}
```

Here we basically create a transformation matrix that transforms the referential based on the angle and position this matrix is based on the matrices below:

Rotation Matrix

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} = R(a)$$

Matrix Translation

$$\begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} = T(T_x, T_y)$$

After that we check every cone is our state and register the one that is closest to the observation and then we calculate a score that will tell us if we accept that that landmark(cone) is the one in the state or if it is a new one:

```
bool ExtendedKalmanFilter::cone_match(const double x_from_state, const doubl
                                      const double x_from_perception,
                                      const double y_from_perception,
                                      const double distance_to_vehicle) {
  double delta_x = x_from_state - x_from_perception;
  double delta_y = y_from_state - y_from_perception;
  double delta = std::sqrt(std::pow(delta_x, 2) + std::pow(delta_y, 2));
  auto limit_function = [](double distance) {
    double curvature = 8.0;
    double initial_limit = 0.5;
    return pow(M_E, distance / curvature) - (1 - initial_limit);
  };
  return limit_function(distance_to_vehicle) > delta;
}
```

The function above is the one that tells us the score. The limit function is an exponential function that uses two constants: curvature and initial_limit. The curvature constant controls the rate of increase of the limit with distance, and the initial_limit constant sets the limit value when the distance is zero, it allows for a larger positional error when the cone is further away from the vehicle.
If the positional error is above the limit we consider the cone a new one and update all the matrices for the new size else we accept the closest cone and the discovery returns the index.

Matrix H calculation

```
Eigen::MatrixXf ObservationModel::get_state_to_observation_matrix(
    const Eigen::VectorXf &expected_state, const unsigned int landmark_inde
    const unsigned int state_size) const {

  Eigen::MatrixXf reformating_matrix = Eigen::MatrixXf::Zero(5, state_size)
  reformating_matrix(0, 0) = 1;
  reformating_matrix(1, 1) = 1;
  reformating_matrix(2, 2) = 1;
  reformating_matrix(3, landmark_index) = 1;
  reformating_matrix(4, landmark_index + 1) = 1;

  Eigen::MatrixXf low_jacobian = Eigen::MatrixXf::Zero(2, 5);

  low_jacobian(0, 0) = -cos(-expected_state(2));
  low_jacobian(1, 0) = -sin(-expected_state(2));
  low_jacobian(0, 1) = sin(-expected_state(2));
  low_jacobian(1, 1) = -cos(-expected_state(2));
  low_jacobian(0, 2) = -expected_state(landmark_index) * sin(expected_state
                       expected_state(landmark_index + 1) * cos(-expected_st
                       expected_state(0) * sin(expected_state(2)) -
                       expected_state(1) * cos(-expected_state(2));
  low_jacobian(1, 2) = -expected_state(landmark_index) * cos(-expected_state
                       expected_state(landmark_index + 1) * sin(expected_sta
                       expected_state(0) * cos(-expected_state(2)) +
                       expected_state(1) * sin(expected_state(2));
  low_jacobian(0, 3) = cos(-expected_state(2));
  low_jacobian(1, 3) = sin(-expected_state(2));
  low_jacobian(0, 4) = -sin(-expected_state(2));
  low_jacobian(1, 4) = cos(-expected_state(2));

  Eigen::MatrixXf validation_jacobian = low_jacobian * reformating_matrix;

  return validation_jacobian;
}
```

This function calculates the Jacobian matrix of the observation model with respect to the state. The Jacobian matrix is used in the Extended Kalman Filter to linearize the non-linear observation model around the current estimate of the state.

The equations for expected_landmark_x and expected_landmark_y are given by the obervation model:

```
Eigen::Vector2f ObservationModel::observation_model(const Eigen::VectorXf &e
                                                    const unsigned int landr
  Eigen::Matrix3f transformation_matrix = Eigen::Matrix3f::Identity();
  // transformation matrix is the matrix that transforms the landmark positi
  // coordinate system to the robot coordinate system for future comparison
  // landmark position

  transformation_matrix(0, 0) = cos(-expected_state(2));
  transformation_matrix(0, 1) = -sin(-expected_state(2));
  transformation_matrix(0, 2) =
      -expected_state(0) * cos(-expected_state(2)) + expected_state(1) * sir
  transformation_matrix(1, 0) = sin(-expected_state(2));
  transformation_matrix(1, 1) = cos(-expected_state(2));
  transformation_matrix(1, 2) =
      -expected_state(0) * sin(-expected_state(2)) - expected_state(1) * cos
  Eigen::Vector3f observation =
      transformation_matrix *
      Eigen::Vector3f(expected_state(landmark_index), expected_state(landmar

  return Eigen::Vector2f(observation(0), observation(1));
}
```

with the equations below representing the calculation for each coordinate:

```
expected_landmark_x = I * cos(t) + m * sin(t) - x * cos(t) - y * sin(t); ->
expected_landmark_y = -I * sin(t) + m * cos(t) + x * sin(t) - y * cos(t); ->
```

These equations represent a transformation from the track's coordinate system to the vehicle coordinate system.With those equations we then calculate the jacobian by deriving in terms of the state variables(x,y,theta):

$$H_t^i = \frac{\partial h(\bar{\mu}_t)}{\partial \bar{\mu}_t}$$

After that we just use the regular EKF formulas to complete the correction step:

$$4: \quad K_t = \bar{\Sigma}_t \, H_t^T (H_t \, \bar{\Sigma}_t \, H_t^T + Q_t)^{-1}$$
$$5: \quad \mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$$
$$6: \quad \Sigma_t = (I - K_t \, H_t) \, \bar{\Sigma}_t$$

The correction is made based on the difference between Zt(the observed position of current cone) and z_hat or h(Ut), in the image. Z_hat comes from mapping the cone in the state to the vehicle coordinate system (observation model).