Trilateration - EKF

December 12, 2019

1 Introduction

In robotics, one usually has to estimate the position of a robot to be able to implement control laws and perform a mission. Kalman filters are widely used for that purpose. While solving the problem described in this document, you will practice C++ programming, in particular matrix calculus wit Eigen. You will implement an Extended Kalman filter for a robotics localization problem, and you will use Vibes to display your results.

2 Problem description

A robot is moving on a plane, where three beacons have been deployed. The robot can measure its distance to each of the beacons. As the robot is moving along a predefined trajectory, the goal of this exercise is to implement a Kalman filter estimating the position of the robot.

The problem is represented by figure 1:

The robot is described by the following state and control[1] equations:

$$\begin{aligned}
\dot{\mathbf{x}} &= \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v\ \cos\theta \\ v\ \sin\theta \\ u_1 \\ u_2 \end{pmatrix} = \mathbf{f}_c(\mathbf{x}, \mathbf{u}) \\
\mathbf{u} &= \begin{pmatrix} -\frac{\sin\theta}{v} & \frac{\cos\theta}{v} \\ \cos\theta & \sin\theta \end{pmatrix} \cdot \begin{pmatrix} \ddot{x_d} + 2\left(\dot{x_d} - v\ \cos\theta\right) + (x_d - x) \\ \ddot{y_d} + 2\left(\dot{y_d} - v\ \sin\theta\right) + (y_d - y) \end{pmatrix}
\end{aligned}$$

with

$$\begin{aligned}
x_d &= 10\ \cos(0.1t) \\
y_d &= 8\ \sin(0.08t)
\end{aligned}$$

The robot is equipped with a sensor simultaneously measuring the distances between the robot and the three beacons, d_1, d_2 and d_3.

Then, we have the following observation equation:

[1]This control equation has been obtained using a feedback linearization method. You will come across this concept later on this year.

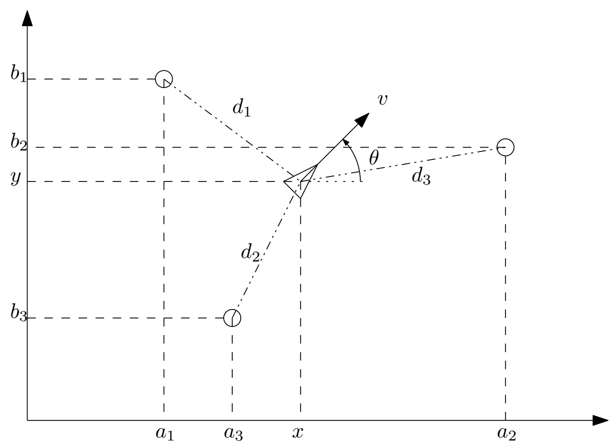

Figure 1: Trilateration problem

$$\mathbf{y} = \mathbf{g}\left(\mathbf{x},\ \mathbf{p}_1,\ \mathbf{p}_2,\ \mathbf{p}_3\right) = \begin{pmatrix} \|\mathbf{p}-\mathbf{p}_1\| \\ \|\mathbf{p}-\mathbf{p}_2\| \\ \|\mathbf{p}-\mathbf{p}_3\| \\ \theta \end{pmatrix}$$

where $\mathbf{p} = (x,\ y)^{\mathrm{T}}$ and $\mathbf{p}_i = (a_i,\ b_i)^{\mathrm{T}}$.

Notes

- x_d and y_d are the desired coordinates of the robot through time. The given control law will regulate the robot to follow this trajectory.
- $\mathbf{f}_c$ is the continuous version of the state equation of the robot. Do not forget to discretize it when implementing your Kalman filter.

3 Questions

3.1 Preliminary steps

1. Create a workspace that will contain all your files.
2. Create the main file of your program: *main.cpp*
3. Copy the files related to vibes (*vibes.cpp* and *vibes.h*) in your workspace (cf C++ tutorials).
4. Create a *CMakeLists.txt*, that you will use to build your workspace.
5. In your *main.cpp*, create a function main initializing a Vibes figure (see algorithm 1)

Your workspace should resemble the one depicted in algorithm 2:

Algorithm 1 Vibes figure

```
vibes::beginDrawing();
vibes::newFigure("trilateration");
vibes::setFigureProperties("trilateration",
                           vibesParams("x", 100, "y", 100, "width", 800, "height", 800));
vibes::axisLimits(-15, 15, -15, 15);
vibes::axisLabels("x", "y", "trilateration");
...
...
...
vibes::endDrawing();
```

Algorithm 2 Workspace

```
|root
    |include
        |Kalman.h
        |Robot.h
        |vibes.h
    |src
        |Kalman.cpp
        |Robot.cpp
        |main.cpp
        |vibes.cpp
    |CMakeLists.txt
```

3.2 Robot class

1. Create a robot class in the files *Robot.cpp* and *Robot.h*.
2. Using an Eigen vector, create an attribute storing the current state of the robot $\mathbf{x}$.
3. Initialize these attributes in the constructor of your *Robot* class.
4. Write a private function f implementing the state equation of the robot.
5. Write a private function g implementing the observation equation of the robot.
6. Write a private function h implementing the control equation of the robot.
7. Write a public function *display* for your robot, that will draw a tank in the current Vibes figure.
8. Write a public function *move* for your robot, that will make the robot move during one time step (use an Euler method). It should take the current simulation time as an input.
9. Write a public function *measure* for your robot, returning the vector $\mathbf{y}$, and taking a *std_ vector* of three Eigen vectors, each of them containing the position of a beacon. Add a small white Gaussian noise β to the output (remember that it is a vector, use Eigen).
10. Write a public function *imu* for your robot, returning the imu measurement (noisy command $\left(\dot{\theta},\ \dot{v}\right)^{\mathrm{T}}$). Add a small white Gaussian noise α to the output (remember that it is a vector, use Eigen).

Note You can use the function given in algorithm 3 to generate a random vector:
For starters, you can set all the standard deviations to 0.01.

Algorithm 3 randn

```
Eigen::VectorXd randn(const Eigen::VectorXd &mu, const Eigen::VectorXd &sigma) {
    uint N = mu.rows();
    assert(N == sigma.rows());
    Eigen::VectorXd result(N);
    std::mt19937 gen((std::random_device()) ());
    for (uint i = 0; i < N; ++i) {
        std::normal_distribution<double> nd(mu(i), sigma(i));
        result(i) = nd(gen);
    }
    return result;
}
```

3.3 Make your robot move

1. Instantiate a *Robot* object in your *main* function, and make it move over a time loop. Use a time step $dt = 0.01$.
2. Place beacons in the scene (display them as disks for example). Take for example $a_1 = -11$, $b_1 = 11$, $a_2 = -11$, $b_2 = -11$, $a_3 = 11$, $b_3 = 0$.
3. Print the measurements made by the robot and check that the values are correct.
4. Check that the imu measurements of the robot are correct.

3.4 Kalman filter class

1. Create a Kalman class in the files *Kalman.cpp* and *Kalman.h*.
2. Using Eigen types, create attributes for storing the current state estimation $\hat{\mathbf{x}}_k$, the current state covariance $\mathbf{\Gamma}_k$, the measurement noise covariance Γ_β, the system noise covariance Γ_α and the time step dt.
3. Initialize these attributes in your constructor.
4. Create a private function A, returning the Jacobian of the discrete state function of the robot.
5. Create a private function C, returning the Jacobian of the observation function of the robot.
6. Implement the f (discrete) and g function inside your Kalman class.
7. Create a public *predict* function, implementing the prediction step of the Kalman filter, and updating the internal state of the class.
8. Create a public *correct* function, implementing the correction step of the Kalman filter, and updating the internal state of the class.
9. Create a public *display* function, that draws a tank at the estimated position of the real robot.

3.5 Estimate the position of your robot

1. Instantiate a Kalman filter in your *main* function.
2. Feed your Kalman filter with the measurement of the real robot via the *correct* function, and display the result.
3. Now, add the imu input to your Kalman filter via the *predict* function, and display the result.

Your Kalman filter should converge and estimate the real position of the robot.

3.6 Play around with parameters

1. What happens if you change the noise parameters ?
2. What happens if you only perform the prediction step ?
3. What happens if you only perform the measurement step ?
4. Try to only perform a measurement step every 5 prediction step. What happens ?