Extended Kalman filter
UCSD Distributed ROS Project

Aaron Ma

March 22, 2015

University of California, San Diego

Abstract

The extended Kalman filter (EKF) is a powerful tool for state estimation. It is especially important to have strong state estimations for non-linear unicycle (TurtleBot) dynamics. In this paper, I will review the necessity of using a Kalman filter, as well how it was implemented and performance.

1 Big Picture

The extended Kalman filter is a means of fusing predicted $\hat{x}_{k|k-1}$, and measured state output, y_k. Fusing y_k and $\hat{x}_{k|k-1}$ will yield a state estimate that is better no matter how large the variance of y_k or $\hat{x}_{k|k-1}$. The EKF is also important because it gives us a way to deal with not receiving measurement update when the algorithm we want to demo is ready and waiting.

1.1 Measurement Output and Overhead Camera

One way to estimate a state is through direct observation (measurement output), y_k. The UCSD Distributed ROS Project team uses an overhead camera and Aruco markers to identify and find the positions of TurtleBots below. The camera nodes output the position of the identified TurtleBots in pixels. A reoccurring problem is that the cameras are not always able to pick up on the Aruco markers. Some reasons for this include, bad lighting, obstructions, and geometric distortions of the Aruco marker perceived by the cameras (When the TurtleBots are near the boundary of the camera vision). During some occasions, the camera won't pick up on the Aruco markers for long periods of time (seconds). This is unacceptable for the distributed algorithms which update around a tenth of a second. y_k varies from the true value, alike all sensors. Our cameras are relatively very accurate, and have a variance of around 2-4 pixels (On the order of a tenth of an inch). We assume that the measurement noise w_k is Gaussian, and our measurement dynamics are linear.

1.2 State Prediction

Another way to estimate a state is to predict the evolution of the state based on it's previous state, and known state dynamics. The TurtleBots act like nonholonomic unicycles and are non-linear. The non-linear kinematics are as follows.

$$\begin{aligned} x_{1,k+1} &= x_{1,k} + v cos(\theta_k) \\ x_{2,k+1} &= x_{2,k} + v sin(\theta_k) \\ \theta_{k+1} &= \theta_k + \omega \end{aligned}$$

With state dynamics we can estimate what the next state will be. Unfortunately, there are many states we are unaware of that affect the evolution of the true state, x_k. For example, given the same input, the TurtleBots will move differently, according to which TurtleBot it is, position on the map, slipping on the tile, dust etc. Because our state model cannot be perfect, there will be an error, $\tilde{x}$, between the state prediction and the true state. Some of the state disturbances, w_s, will be white and Gaussian, such as maybe the variance in motor speeds given an input, but others will not have zero mean. We deal with the Gaussian noise using the extended Kalman filter by fusing the predicted state with the measurement output, and we deal with the non-Gaussian disturbances by implementing an integral feedback loop, which will be explained in detail later in the paper. It is important to note that the unicycle kinematics are non-linear. This implies that small disturbances in our predicted evolution are not accurately depicted by the true variance, since small changes in the state can change the kinematics drastically. Because of this, we have to assume the state noise variance to be conservative.

1.3 Fusing the State Prediction and Measurement Output

Knowledge of the state and measurement noise variance allows us to fuse y_k and $\hat{x}_{k|k-1}$, yielding a better estimate (lower least square means estimate) as long as the variance of the state or measurement noise are not underestimated. Testing the TurtleBots for the state and measurement noise, we found that Q, the state spectral density, is very large in relation to R, the measurement spectral density. This means that the EKF will develop estimations which are more heavily weighted towards the measurement data. Fusing the measurement data, and the predicted state is as follows.

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} - L_k(y_k - H_k\hat{x}_{k|k-1})$$

Where $\hat{x}_{k|k}$ is our best state estimate, and L_k, the Kalman gain, determines whether y_k or $\hat{x}_{k|k-1}$ is weighted more heavily and is determined by the differential Riccati equation.

2 Personal Contributions

2.1 Exploring the Previously Built EKF

My initial objective was to debug and implement a previously made prototype extended Kalman filter into our TurtleBot deployment. I started debugging the previous extended Kalman filter by making a node that publishes fake poses and velocity with variance. The extended Kalman filter would take the fake measurement data, predict the next state with the given input, and publish the state estimate. Problems developed with the Covariance matrix P; after several steps, the covariance would spike and the algorithm would crash. The proposed fix for this was to reset the covariance every few iterations, however this is a bad fix and doesn't take advantage of the strength of the extended Kalman filter being able to recursively march the differential Riccati equation in real time. I found it was easier to remake the code with a linear Algebra library called "Eigen".

2.2 Determining TurtleBot Dynamics

In order to create the extended Kalman filter it is necessary to understand the TurtleBot dynamics. The previous EKF assumed that y_k and $\hat{x}_{k|k}$ are determined in the same units of measurement, but this is incorrect. The camera nodes output y_k in units of camera pixels (which is roughly a tenth of an inch per pixel), while the TurtleBots work in the metric system. Since the conversion between camera pixels and metric units had not been previously tested, I decided to run a small experiment to develop an initial conversion guess (shown later in paper). It is important to note that the matrix H, the matrix that transforms $\hat{x}_{k|k}$ to y_k is really the conversion between pixels and metric units. Most of our algorithms run in units of pixels. For this reason, we take H to be the identity matrix, and convert our state dynamics to match the pixel units frame of reference.

$$\text{Plant: } x_{k+1} = \tilde{A}x_k,\ \tilde{A} = kA$$
$$\text{Output: } y_k = H_k\hat{x}_{k|k},\ H_k = I$$

Where k is the conversion between metric units to pixels (roughly 170 pixels per meter). Performing the test is also a way to get data regarding the state and measurement variances which is needed for Q and

R in the differential Riccati Equation. I determined Q by acquiring data from moving the TurtleBot at a certain speed for a set amount of time and determining the standard deviation in the distance traveled. It was quickly obvious that not all of the state noise was zero mean. Some of the disturbances were dependent on factors such as battery life, input (inputs were not completely scalar) etc. This became a large problem that I show the solution to a little later. The measurement variance is easy to calculate by determining the standard deviation of a stationary TurtleBot. It quickly follows that Q and R are...

$$\text{State Spectral Density: } Q = E(ww^T)$$
$$\text{Measurement Spectral Density: } R = E(rr^T)$$

Where w and r are the state and measurement noise respectively.

2.3 Creating the Extended Kalman Filter

At this point, we have the turtlebot Dynamics (unicycle dynamics), a universal unit of measure and conversion, and some okay values for Q and R. This is enough information to fuse $\hat{x}_{k|k-1}$ and y_k into $\hat{x}_{k|k}$, a better state estimation. The EKF is designed to work on board of each of the TurtleBots. Initially I had designed the code so that each EKF would filter every TurtleBot state, for communication reassurance and attempt at making it distributed, however it turned out to be unnecessary. Now each of the TurtleBots have an EKF and publish their state estimate to a topic where each of the TurtleBot estimates can be found.

A large part of the extended Kalman filter is dealing with multiple modes of operation. For example, a moving TurtleBot will have state noise while a stationary TurtleBot will not. In the extended Kalman filter, this corresponds to having a very high Q (in the moving case) or a very low Q (stationary TurtleBot). It is important in the future to modify the EKF to tend to other dynamic modes we use. We can improve the EKF by tuning the Q to the TurtleBot kinematics (which are not completely scalar), but for the time being, assuming a conservative Q is sufficient, as long as it is zero mean.

The biggest contribution that the extended Kalman filter provides, is the ability to track a TurtleBot when there are no measurements being published. This has been a persistent problem with the TurtleBots which is easily solved by the extended Kalman filter. When y_k isn't published by the camera node (e.g. if the camera cannot find the TurtleBot), then the measurement update of the extended Kalman filter will be skipped, and the $\hat{x}_{k|k} = \hat{x}_{k|k-1}$. That is to say, the state estimate will be what we predict the state to be given the previous state, and estimated kinematics. Of course, this estimate will be less accurate than when y_k and $\hat{x}_{k|k-1}$ are fused (especially with a high Q), but this provides a means of evolving the state estimate without a measurement.

2.4 Implementation and Integral Feedback

Once the code is created and tested with the dummy nodes, it can be tested on the TurtleBots. This was done by simply modifying some of our previously completed algorithms to accept state estimates published by the extended Kalman filter. After a lot of debugging, the Kalman filter becomes functional and can be used by most of the modified algorithms.

After running algorithms on the extended Kalman filter, a significant problem arose. Previously, it was discovered that our state noise was not truly white. In fact, the state noise, would vary greatly depending on the circumstances (which are mostly unknown at the moment). For example, if two different TurtleBots were given kinematic commands:

$$\text{Velocity: } v = v_1$$
$$\text{Angular Velocity } \omega = \frac{v}{r},\ r = radius$$

the TurtleBots would move at different speeds and travel a different radius. This is a big problem for some of the more highly dynamic distributed algorithms (such as cyclic pursuit), where the TurtleBots are supposed to be restricted to the same circle. Fortunately, this problem can be solved using the Kalman innovation $H_k y_k - \hat{x}_{k|k-1}$, which is the difference between the measured state and the predicted state. We assume our predicted state to be nominal given the previous state. An integral feedback uses the innovation to adjust the input given to the TurtleBot until the kinematics we have tends towards the kinematics we want.

$$\text{Integral: } I = K_I(H_k y_k - \bar{x})t$$
$$\text{Input: } u_{k+1} = u_k + I$$

Where $\bar{x}$ is $\hat{x}_{k|k-1}$, K_I is our integral gain, and t is the time step. The addition of the integral feedback proved to be significant, and forced the $\hat{x}_{k|k}$towards $\hat{x}_{k|k-1}$, within some ϵ caused by Gaussian noise.

3 Preliminaries

3.1 System Model

It is important to understand and develop an intuition regarding the TurtleBot dynamics in order to deploy the EKF to its full potential. The TurtleBots are subject to non-linear unicycle (under-actuated and nonholonomic) kinematics.

$$x_{1,k+1} = x_{1,k} + B_{1,k} u_{1,k} cos(\theta_k)$$
$$x_{2,k+1} = x_{2,k} + B_{1,k} u_{1,k} sin(\theta_k)$$
$$\theta_{k+1} = \theta_k + B_{2,k} u_{2,k}$$
$$\text{Generalized as } x_{k+1} = F_k x_k + G_k u_k + w_k$$

Where u_1, u_2 are the linear and angular velocities respectively, and are recursively being modified with an integral controller.

3.2 The Kalman Filter

The extended Kalman filter is a non-linear state estimator, and is an extension of the Kalman filter, which is for linear dynamics. As previously described, the Kalman filter fuses y_k and $\hat{x}_{k|k-1}$ to get $\hat{x}_{k|k}$, the best state estimation, as long as the spectral densities, Q and R are known. The Kalman filtering process is as follows.

0. TurtleBots Move:

$x_{k+1} = F_k x_k + G_k u_k + w_k$ True State Evolution

$y_k = H_k x_k + J_k u_k + r_k$ True State Measurement

1. Predict the next State, Output and Covariance:

$\hat{x}_{k+1|k} = F_k \hat{x}_{k|k} + G_k u_k$ Predicted State

$y_k = H_k x_k + J_k u_k$ Measurement Output

$P_{k+1|k} = FkP_{k|k}\ F -_k^T + Q_k$ Predicted Covariance

2. Incorporate new measurements (Measurement Update):

$L_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1}$ Kalman gain

$\hat{x}_{k|k} = \hat{x}_{k|k-1} + L_k(y_k - \hat{y}_{k|k-1})$ Best State Estimate (Measurement Update)

$P_{k|k} = (I - L_k H_k) P_{k|k-1}$ Covariance Update

When assumed that no measurements are skipped, we can combine elements of the Kalman gain and covariance in one step called the Riccati difference equation (RDE).

$$P_{k+1|k} = F_k P_{k|k-1} F_k^T - F_k P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} H_k P_{k|k-1} F_k^T + Q_k$$
$$L_k = -P_k H_k^T R_k^{-1}$$

In a time-invariant model, a solution to P exists and can be determined by the discrete Algebraic Riccati Equation (dARE).

$$L_\infty = P_\infty^- H^T (H P_\infty^- H^T + R)^{-1}$$

3.2.1 The Extended Kalman Filter

There are several state estimation methods that can be used for non-linear dynamics including particle, scented Kalman, ensemble Kalman, linearized Kalman, and the extended Kalman filter. Some of these methods, such as the particle filter are more practical when the state is near a repulsive equilibrium, and the probability density function diverges as time increases. The most commonly used non-linear filter is the extended Kalman filter, which runs under the assumptions that the state evolution is non-linear and the measurement is linear(F is non-linear, H is linear). In order to run the extended Kalman filter, F needs to be linearized when solving the Riccati difference equation.

$$F = \frac{\partial f}{\partial x}(\hat{x}_{k-1|k-1}, \hat{u}_{k-1|k-1}, 0) = \begin{bmatrix} 1 & 0 & -tv_{k-1}sin(\theta_{k-1}) \\ 0 & 1 & tv_{k-1}cos(\theta_{k-1}) \\ 0 & 0 & 1 \end{bmatrix}$$

While the rest of the state predictions are done with the nonlinear dynamics.

4 Methodology

4.1 Determining System Dynamics and Noise

As described previously, deciding on a unit of measurement is necessary to fuse the measurement and state predictions. In order to find the conversion between pixels of the camera and metric units, simple tests were employed.

4.1.1 General Procedure

1. Determine Camera Variance
   - (a) Keep TurtleBot stationary
   - (b) Record data by capturing the published poses
   - (c) Move TurtleBot to another section
   - (d) Repeat many times
   - (e) Analyze the data, determine mean, standard deviation, and variance of output.
2. Determine Metric to Pixel Conversion and State Variance
   - (a) Tell the TurtleBot to move for set time
   - (b) Record distance traveled by capturing the published poses
   - (c) Repeat many times
   - (d) Repeat using different linear and angular velocity
   - (e) Analyze the data, determine mean, standard deviation, and variance of state evolution.

4.1.2 Results

Data and code used for analysis is in the back of this report. Using this test we can get a general idea for the state noise. I calculate the state noise to be 167 pixels to ros units, with a standard deviation of 19.4. The mean was close to zero in most areas.

4.2 Interface Design

The EKF uses ROS (Robotic Operating System) to communicate with other nodes. The node I created fits well into our existed network. In our existing network the camera node observes and publishes poses to a topic called amclPose. A node called "interRobotCommunication" subscribes to amclPose, processes and assign identification to the poses and publishes. "interRobotCommunication" has been modified to publish to "toKalmanFilter". Onboard the TurtleBots, the EKF node subscribes to the "toKalmanFilter" topic and determines whether the published pose matches the TurtleBots identification. If it does, the TurtleBot will accept the message as new measurement data. The Kalman filter also subscribes to "velocity" which contains the input sent to the TurtleBot base. With the input and measurement data, the Kalman filter can produce $\hat{x}_{k|k}$ fuse it with y_k. The best state estimate, $\hat{x}_{k|k}$ is now sent to a topic called "allPositions" along with its identification, and a local topic called "afterKalman" which goes directly to the TurtleBots other running nodes. Since the EKF node also subscribes to the "velocity" it is able to compare the measured state and the state prediction using the Kalman innovation. The EKF node uses product of the innovation and an integral gain to modify the input being sent to the TurtleBot, which adapts to fluctuations and biases in TurtleBot kinematics.

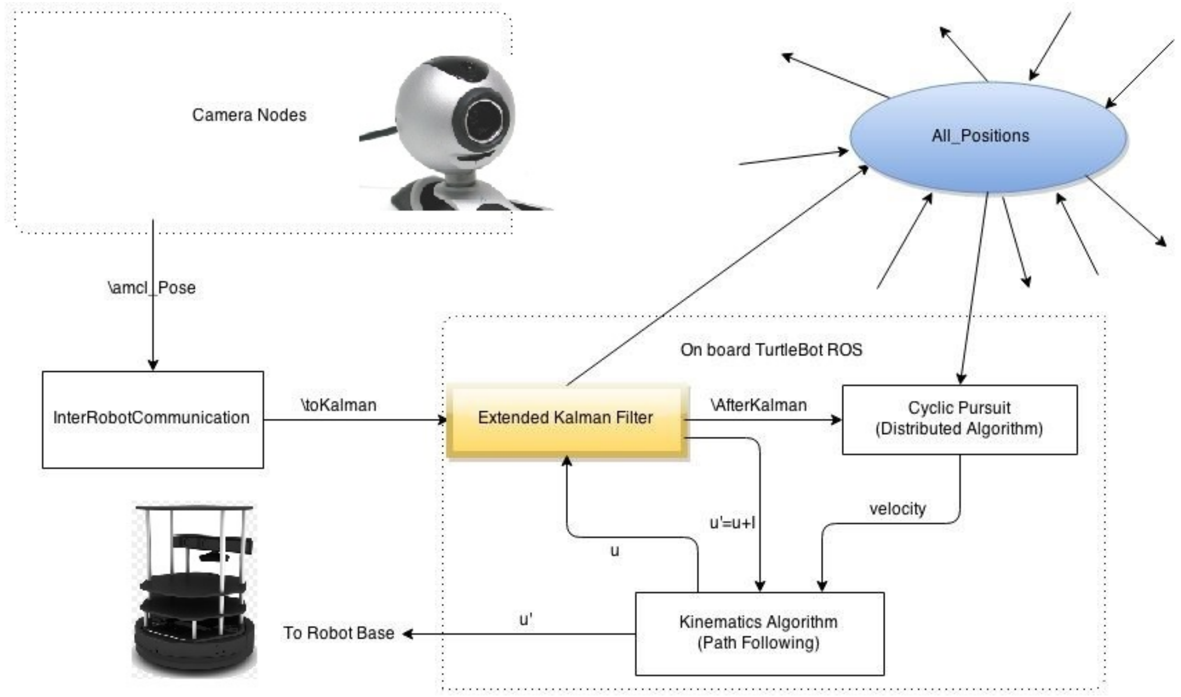


Figure 1: Above is the ROS network we use. The Extended Kalman filter is highlighted in yellow. Where lines start represents messages being published from a node to a topic. The line ends at a node that is subscribing to the topic. The Blue bubble is a global topic called allPositions. This topic has many subscribers and publishers, which is necessary for mass robot communication for our distributed algorithms.

4.3 Testing and Tuning the Extended Kalman Filter

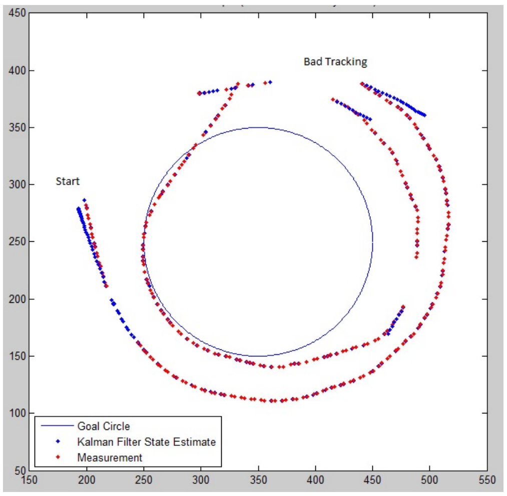
4.3.1 Missing Measurements

One of the most significant differences that the EKF node has made is the constant publishing of state estimate messages. Before, we depended on measurements from the camera for the TurtleBots to be able to run their deployment algorithms. The EKF handles missing measurements simply by skipping the measurement update segment of the Kalman sequence. This means that the state estimate, $\hat{x}_{k|k}$, will be based on the

$\hat{x}_{k|k-1}$, with more uncertainty, but it does allow for the TurtleBots to have a strong idea about where they are when measurements are not available.

4.3.2 State Prediction

Once EKF was ready to go, several problems had to be solved before it was fully functional. One of the hardest issues was trying to get the predicted state to be white. A lot of this issue was fixed through integral feedback, however larger problems occurred. While checking to make sure the state evolution on the Kalman filter was working, I found out that the dynamics were facing backwards. It turns out that the Aruco markers have been backwards the whole time. Most of our code is based on the existing orientation of the Aruco markers, so the best solution was to fix the orientation in the EKF node. The other main challenge was to most accurately guess the average state dynamics. In order to test how the state predictions were matching up to the measured data (how accurate the innovation was) I recorded the published poses from the camera and the state estimate from the EKF during a circular path following execution.



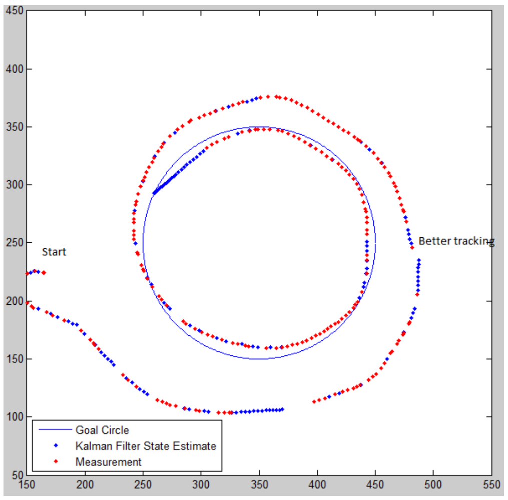


Figure 2: Left: This is the first test conducted to find the predicted state accuracy. I discovered while looking at this data that the Aruco markers are backwards, as the state predictions evolve the wrong direction. The blue dots show the state estimate, while red is the measured state. For testing purposes, I would put my hand over the Aruco marker every once in a while, which would disrupt the measurement data. Notice how the state estimate quickly recovers when a measured state is received. That is because the spectral density of state noise, Q, is much greater than the spectral density of measurement R.

Figure 3: Right: This is one of the last tests conducted. Notice how the state evolution tracks the state much better when I obstruct the Aruco marker. This is easily seen by looking for large gaps between the red markers. If the blue markers (state estimate) connect the trajectory, then the predicted state is more or less accurate. These figures depict how the EKF will react to not receiving measurements; the predicted state will now become the best state estimate until new measurement data is available.

4.4 Integral Feedback

Before integral feedback was added to the system, cyclic pursuit would not run very well. There would be very obvious steady state error between two TurtleBots pursuing each other as a result of reacting differently to inputs. While running the circular pursuit algorithm, the TurtleBots would converge to different size circles, which is unacceptable. A solution to this is to implement an integral control by taking advantage of the innovation in the Kalman sequence.

$I_0 = 1$ Initialize the Integral term

$I_{k+1} = I_k + K_I t(H_k^{-1} y_k - \hat{x}_{k|k-1})$ Modify the Integral term

$u_{k+1} = I u_k$ Incorporate the Integral term

It becomes apparent while running two TurtleBots that the integral feedback forces convergence to the same circle radius. It is also interesting to see how different steady state integral terms can be (up to $2x$ difference).

5 Tips for Using

The EKF node should work very well without modification and should be used for any purpose. To use the extended Kalman filter, make sure that the interRobotCommunication that is active is publishing to "toKalman". The EKF node will publish to the global topic, "/all_positions", which all TurtleBots will have access to. This is the topic that you will want the TurtleBots to subscribe to if you want to learn about other TurtleBots state estimation. The topic "/afterKalman" is used for sending state estimate locally to other nodes on the same TurtleBot. As long as "/mobile_base/commands/velocity" is being published to by your deployment algorithm, the extended Kalman filter will subscribe to the topic and receive input data. The extended Kalman filter will automatically publish the integral terms needed to correct inputs so that the state dynamics match what you would expect. The integral term messages are Float64 values and are published to local topics "cal10" and "cal1D", which are the angular velocity and the linear velocity integral terms respectively.

A. Getting own TurtleBot's state estimation
   1. Make sure your algorithm subscribes to "/afterKalman".
   2. Message will be of type PoseWithName, it is necessary to include PoseWithName.h

B. Getting other TurtleBots' state estimation
   1. Make sure your algorithm subscribes to "/all_positions".
   2. Message will be of type PoseWithName, it is necessary to include PoseWithName.h
   3. You can identify the pose obtained from "all_positions" by getting the PoseWithName.name value

C. Getting Integral terms for state input correction
   1. Make sure your algorithm subscribes to "cal10" and "cal1D".
   2. Message will be of type Float64, you may need to convert.
   3. Your input to the TurtleBot base should look like $u_k = \alpha I v$, where α would be the metric to pixel conversion, I, would be the integral term from subscription, and v would be the desired velocity term.

D. Getting the Kalman Error
   1. Subscribe to the local topic "ke".

The EKF node publishes every .1s. I noticed that publishing from the TurtleBots too frequently can cause them to temporarily freeze. For the time being, .1s publishing rate is adequate.

The ROS to pixel conversion is somewhere around 170 pixels per 1 ROS unit (might be meter but depends on surface), while the radial units are given in radians per second.

6 Future Improvements

Many of the difficulties with implemented the extended Kalman filter have already been mentioned. There are several issues that can be dealt with to improve the EKF. Improving the state dynamics further is a priority. Quicker convergence of the predicted state and the true state is important for the beginning of our deployment schemes. This can be done by implementing a proportional controller in the integral feedback loop (PI). I currently have a small proportional term which lowers oscillation slightly, but there is work to be done to determine optimal integral, and proportional gains.

Determining our system model also might be a priority. There are a lot of causes for state disturbance in our lab. The TurtleBots respond to some areas of the tile differently, and it might be worthwhile trying to determine what could be causing it.

An assumption made in the TurtleBot kinematics, is having a linear response to input. For example, having twice the velocity with twice the input published. This doesn't exist, as the real dynamics are nonlinear (dust, static friction, etc). The state noise also depends on magnitude of input published as well. For the time being, state noise is a function of whether or not the TurtleBot is moving.

7 Conclusions

With the addition of the extended Kalman filter, I expect algorithms to be completed much quicker. A lot of the issues that occur when trying to implement a deployment algorithm is the state estimation, and how to deal with infrequent measurement data. The extended Kalman filter completely fixes that problem. Once the EKF was functional, implementing cyclic pursuit was very quick and only involved coding and a small amount of debugging.

8 Source Code

8.1 ekf.cpp

```
#include <iostream>
#include <stdio.h>
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include "geometry_msgs/PoseWithCovarianceStamped.h"
#include <geometry_msgs/PoseStamped.h>
#include <tf/tf.h>
#include <fstream>
#include <math.h>
#include "PoseWithName.h"
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include "eigen/Eigen/Dense"
#include <vector>
#include <stdlib.h>
#include "std_msgs/String.h"
#include <std_msgs/Float64.h>
#include <sstream>
/* -------------------------------------------------------------------------------------
CREDIT
Credit to
Eigen for their opensource linear algebra library and headers
Allen for previous iteration of Kalman filter
ROS opensource

------------------------------------------------------------------------------------
```

```
BEGIN
*/
using namespace std;
using namespace Eigen;
using Eigen::MatrixXd;
//***Remove later
/*------------------------------------------------------------------------------------------
** Initialization Block
**
** Initialize Q,R,W, I(identity) as a permanent matrices determined by
** covariance in our model
**
** Initialize P,H,X,XT,A,K,Z as matrices that vary
** during iterations
*/
Matrix3f Q= Matrix3f::Zero();
Matrix3f R= Matrix3f::Zero();
Matrix3f W= Matrix3f::Identity();
Matrix3f I= Matrix3f::Identity();
Matrix3f P= Matrix3f::Zero();
Matrix3f H= Matrix3f::Identity();
MatrixXf X(3,1);
VectorXf XT(3);
Matrix3f A;
Matrix3f K;
VectorXf Z(3);
vector<Vector3f> Xv;
vector<Vector3f> XTv;
vector<Matrix3f> Av;
vector<Matrix3f> Kv;
vector<Matrix3f> Pv;
vector<Matrix3f> Hv;
vector<string> robots; //Initialize a vector of robot names
std::string name_;
turtlebot_deployment::PoseWithNamePtr newPose_;
bool got_pose_, stationary;
double theta,
x,
y;
double T =10;
int counter11=0;
class agent{
public:
float x, y, theta, velo, omega, t;
string name;
};
vector<agent> agentVector;
class getName
{
public:
getName();
private:
ros::NodeHandle ph1_;
std::string name1_;
} ;
getName::getName():
ph1_("~"),
name1_("no_name")
{
```

```
ph1_.param("robot_name", name1_, name1_);
name_=name1_;
}
void poseCallback(const turtlebot_deployment::PoseWithName::ConstPtr& posePtr)
{
std::cout<<"name_ "<<name_<<"\n";
std::cout<<posePtr->name<<"\n";
int size = robots.size();
int iTemp;
agent a; //Temp agent used if a new agent is introduced
if (posePtr->name==name_){
iTemp=0;
got_pose_=true;
if (iTemp==size)
{
//Add elements to necessary vectors
robots.push_back(posePtr->name); //Adds robot name element to vector<string>robots
agentVector.push_back(a); //Adds class "agent" element to vector<agent>agentVector
//Initialize new detected robot matrices and state
P=Matrix3f::Zero(); //Initialize Matrix P(confidence) to be "loose"
P(0,0)=900;
P(1,1)=900;
P(2,2)=900;
X<<1,2,3; //Define initial position
XT=X;
//Add new agent's matrix elements to corresponding vectors
Xv.push_back(X);
Pv.push_back(P);
XTv.push_back(XT);
//Set new agent's position
agentVector[iTemp].name=posePtr->name;
agentVector[iTemp].x=posePtr->pose.position.x;
agentVector[iTemp].y=posePtr->pose.position.y;
agentVector[iTemp].theta = tf::getYaw(posePtr->pose.orientation)+3.14;

}
else{
/*
** This block runs if posePtr->name was found in found, ie if robot's id was previously detected.
*/
std::cout<<"pass";
//Set found agent's position
agentVector[iTemp].x=posePtr->pose.position.x;
agentVector[iTemp].y=posePtr->pose.position.y;
agentVector[iTemp].theta = tf::getYaw(posePtr->pose.orientation)+3.14;
}
}
else{
}
}
void iptCallback(const geometry_msgs::Twist::ConstPtr& ipt)
{
//Determine if subscribed .name is already in group. If not, this initiallizes a new one
int size = robots.size();
int iTemp;
iTemp=0;
agent a;
/* -----------------------------------------------------------------------------------------
** Same iTemp search algorithm as in poseCallback
** ***This algorithm may become a separate function
*/
if (size==0){
robots.push_back(name_);
agentVector.push_back(a);
agentVector[iTemp].name=name_;
P=Matrix3f::Zero();
P(0,0)=900;
P(1,1)=900;
P(2,2)=900;
H=Matrix3f::Identity();
Pv.push_back(P);
Hv.push_back(H);
X<<1,2,3;
Xv.push_back(X);
XT=X;
XTv.push_back(XT);std::cout<<"Measured: \n"<<Z<<"\n\n";
agentVector[iTemp].velo=ipt->linear.x;
agentVector[iTemp].omega=ipt->angular.z;
}
else{
agentVector[iTemp].velo=ipt->linear.x;
agentVector[iTemp].omega=ipt->angular.z;
}
}
int main(int argc, char **argv)
{
ros::init(argc, argv, "ekf_temp"); //Ros Initialize
ros::start();
ros::Rate loop_rate(T); //Set Ros frequency to 50/s (fast)
getName getname;
ofstream myfile;
const char *path="/home/turtlebot/ekfData.txt";
myfile.open(path);
myfile <<"Data";
myfile.close();
ros::NodeHandle nh_, ph_, gnh_;
ros::Subscriber pos_sub_ ;
ros::Subscriber ipt_sub_ ;
ros::Publisher gl_pub_ ;
ros::Publisher sf_pub_;
ros::Publisher nm_pub_;
ros::Publisher cal0_pub_;
ros::Publisher calD_pub_;
ros::Publisher kalmanError;
void poseCallback(const turtlebot_deployment::PoseWithName::ConstPtr& pose);
void iptCallback(const geometry_msgs::Twist::ConstPtr&);
// ROS stuff
// Other member variables
got_pose_=false;
stationary=false;
Q(0,0)=0;
Q(1,1)=0;
Q(2,2)=0;
R(0,0)=.01;
R(1,1)=.01;
R(2,2)=.01;
double OmegaC=1.5;
double OmegaD=.7;
double counter12=0;
double x0=0;
double y0=0;
double ec=0;
double ed=0;
double id=.7;
double ic=1.5;
double ed0=0;
double ec0=0;
std_msgs::Float64 floatMsg, floatMsg2, ke;
int iTemp;
iTemp=0;
int size = robots.size();
pos_sub_= nh_.subscribe<turtlebot_deployment::PoseWithName>("toKalmanfilter", 1,poseCallback);
ipt_sub_=nh_.subscribe<geometry_msgs::Twist>("mobile_base/commands/velocity",1,iptCallback);
gl_pub_ = gnh_.advertise<turtlebot_deployment::PoseWithName>("/all_positions", 1, true);
sf_pub_= gnh_.advertise<turtlebot_deployment::PoseWithName>("afterKalman",1,true);
nm_pub_= gnh_.advertise<turtlebot_deployment::PoseWithName>("nametest", 5);
cal0_pub_= gnh_.advertise<std_msgs::Float64>("cal0", 1,true);
calD_pub_= gnh_.advertise<std_msgs::Float64>("calD", 1,true);
calD_pub_= gnh_.advertise<std_msgs::Float64>("calD", 1,true);
kalmanError=gnh_.advertise<std_msgs::Float64>("ke",1,true);
while (ros::ok()) {
got_pose_=false;
ros::spinOnce();
if (got_pose_==true){
cout<<"got_pose_: "<<got_pose_<<"\n";
R(0,0)=1;
R(1,1)=1;
R(2,2)=1;
}else{
R(0,0)=10000;
R(1,1)=10000;
R(2,2)=10000;
}
if (stationary==true){
Q(0,0)=0;
Q(1,1)=0;
Q(2,2)=0;
}else{
Q(0,0)=5;
Q(1,1)=5;
Q(2,2)=5;
}
if (0<robots.size()){
///
VectorXf Z(3);
Matrix3f temp;
P=Pv[iTemp];
//Stage 1
Z << agentVector[iTemp].x,agentVector[iTemp].y,agentVector[iTemp].theta;

//Calibration
if (counter11>10){
XT <<
    XT(0)+agentVector[iTemp].velo*167/T*cos(XT(2))/OmegaD,XT(1)+agentVector[iTemp].velo*167/T*sin(XT(2))/OmegaD,XT(
}
X <<
    X(0)+agentVector[iTemp].velo*167/T*cos(X(2))/OmegaD,X(1)+agentVector[iTemp].velo*167/T*sin(X(2))/OmegaD,X(2)+ag
```

```
cout<<"Velocity: "<<agentVector[iTemp].velo*167/T<<"\n";
//Stage 2
if (got_pose_==true){
A << 1, 0, -agentVector[iTemp].velo*167/T*sin(agentVector[iTemp].theta),0,
    1,agentVector[iTemp].velo*167/T*cos(agentVector[iTemp].theta),0, 0, 1;
P=A*P*A.transpose()+W*Q*W.transpose();
//Stage 3
temp=(W*P*W.transpose()+W*R*W.transpose());
K=P*W.transpose()*temp.inverse();
//Stage 4
X=X+K*(Z-X);
//Stage 5
P=(I-K*W)*P;

//PID FEEDBACK
    if (counter12+5<counter11){
        ed=(sqrt((XT(1)-y0)*(XT(1)-y0)+(XT(0)-x0)*(XT(0)-x0))-sqrt((X(1)-y0)*(X(1)-y0)+(X(0)-x0)*(X(0)-x0)));
        id=id+.01*ed;
      OmegaD=0*ed+id+0*(ed-ed0);//PID
       ed0=ed;
      if ((XT(2)-X(2))>3.14){
        ec=((XT(2))-(X(2)+2*3.14));
    //OmegaC=OmegaC+.2*((XT(2))-(X(2)+2*3.14));
          ke.data = XT(2)-(X(2)+2*3.14);
      }
    else{
        ec=((XT(2))-(X(2)));
        //OmegaC=OmegaC+.2*((XT(2))-(X(2)));
        ke.data=XT(2)-X(2);
    }
    ic=ic+.2*ec;
    OmegaC=0*ec+ic+0*(ec-ec0);
    ec0=ec;



      XT=X;
      counter12=counter11;
      if (OmegaD<.4){OmegaD=.4;id=.4;}
      if (OmegaD>1.2){OmegaD=1.2; id=1.2;}
      if (OmegaC<.5){OmegaC=.5;ic=.5;}
      if (OmegaC>4){OmegaC=4;ic=4;}
      floatMsg.data=OmegaC;
      floatMsg2.data=OmegaD;
      cal0_pub_.publish(floatMsg);
      calD_pub_.publish(floatMsg2);
      kalmanError.publish(ke);
      x0=X(0);
      y0=X(1);

    }
}

//Set Vectors
Xv[iTemp]=X;
Pv[iTemp]=P;
turtlebot_deployment::PoseWithName goalPose;
goalPose.pose.position.x = X(0);
goalPose.pose.position.y = X(1);
goalPose.name=agentVector[iTemp].name;
goalPose.pose.orientation =tf::createQuaternionMsgFromYaw(X(2)+3.14);
//added if Statement
//if (goalPose.name!=""){
gl_pub_.publish(goalPose);
goalPose.name=name_;
nm_pub_.publish(goalPose);
sf_pub_.publish(goalPose);
//}
counter11=counter11+1;
cout<<"Counter: "<<counter11<<"\n";
cout<<"Number of Robots: "<<size<<"\n";
cout<<"Robot #: "<<iTemp<<"\n";
std::cout<<"Measured: \n"<<Z<<"\n\n";
std::cout<<"OmegaC\n"<<OmegaC<<"\n---------\n";
std::cout<<"OmegaD\n"<<OmegaD<<"\n---------\n";
std::cout<<"Goal Pose\n"<<goalPose<<"\n---------\n\n\n\n";

std::cout<<"-------------------------------------------------------------------";
//}
///
loop_rate.sleep();
//sleep(1/50);
}
}
}
//END
```

8.2 State Dynamics Data and Quick Analysis

```
clear
clc

%% Determine Q, R, and conversion

%% Data
Xtemp=[534.8
217.3

443.8
209.5


443.8
209.5

352.3
201.1


352.3
201.1

274.5
194.2
```

```
274.5
194.2

182.0
186.2


175.9
183.4

264.2
189.5


264.2
189.5

356.8
196.7


356.8
196.7

442.8
204.7


442.8
204.7

524.65
212.15


534.5
213.75

450.4
219.6


450.4
219.6

365
225.3


365
225.3

315.7
229.3


315.7
229.3
```

```
239.2
234.5


239.2
234.5

171.15
238.8


190
230

249.6
229.4


249.6
229.4

332.1
228.8];

for i=1:length(Xtemp)
    n=rem(i,4);
    if n==0
       n=4;
    end
    X(n,ceil(i/4))=Xtemp(i);
end

for i=1:length(X)
   D(i)=sqrt((X(3,i)-X(1,i))^2+(X(4,i)-X(2,i))^2);
end
D(11)=[];
D=D*10/5;
Du=mean(D)
DMedian=median(D)
Dstd=std(D)
Dcov=(std(D))^2

Ytemp=[-.976
2.15

-.713
.703

.025
.996

.685
.727

-.10
.000

-.925
.373
```

```
-.335
.945

.345
.937

.931
.363

-.975
.222

-.484
.875

.285
.958

.913
.406

-.959
.285

-.423
.906

.295
.955

.915
.402

-.98
.195

-.612
.790

.155
.987

.845
.535

-.978
.206

-.488
.873

.277
.960

.922
.388

-.983
```

```
.180

-.521
.853

.127
.991

.818
.574

-.991
.127

-.745
.665

.025
.999

.746
.665

-.998
.065];

for i=1:length(Ytemp)
   n=rem(i,2);
    if n==0
       n=2;
    end
    Y(n,ceil(i/2))=Ytemp(i);
end
for i=1:length(Y)
[r1(i) r2(i) r3(i)] = quat2angle([(Y(1,i)) 0 0 (Y(2,i))]);
end
r1=r1*360/2/pi+180;

% fprintf('angle: %f \n',r1)

for i=1:length(Y)-1
   angleV(i)=r1(i+1)-r1(i);
   if abs(angleV(i))>180
      angleV(i)=angleV(i)-360;
   end
end
n=0;
for i=1:length(angleV)
      if abs(angleV(i-n))<55
        angleV(i-n)=[];
        n=n+1;
    end
end
angleV=angleV/5;
dAngleMean=mean(angleV)
dAnglemedian=median(angleV)
dAngleStd=std(angleV)
```

```
%% Results (linear about V=.1, omega=1
% deltaAngle = 18.2 1 ros angle/s to degrees/s(CCW), STD = 2.05
% dDistance = 167 1 ros unit/s to pixel/s       STD = 19.4
```

References