Kalman Filter for Turtlebot Deployment

Aaron Ma

December 19, 2014

An Extension of Gu Peizhen's "Applying Kalman filter to Robot Networks"

University of California, San Diego

Abstract

Implementing the Kalman filter with the Turtlebots was motivated by problems that occur during deployment and Gu Peizhen's previous work on the matter. The Kalman filter can offer several improvements to localization including the filtering of noise and missing/incorrect pose data sent. With the Kalman Filter, the Turtlebot's will always be able to make an state position estimate even with measurement data that may be unreliable. This report serves as an extension to Gu Peizhen's work on "Applying Kalman filter to Robot Networks".

1 Big Picture

Kalman filters are important in state estimations where measurements may be unreliable. For example, the last time that the Turtlebots were deployed, one of the Aruco markers was not being read, or would only sometimes be read. Without the Kalman filter, the turtlebots currently have no means of estimating their predicted state when no measurement is provided.

1.1 Turtlebot Localization

Aruco Markers Insert Aruco Marker picture and description

Recently, Katherine Liu has upgraded the Turtlebot tracking scheme with Aruco markers. Previously, colored tags were placed on top of the turtlebots so that their pose may be measured by an overhead camera. The colored tags were unreliable due to light conditions, which provided motivation for a Kalman filter. The new Aruco markers provides more robust measurement, however there are occasions where state measurement is either missing or wrong.

1.2 Applying the Kalman Filter

My work aims to build Kalman filters for the Turtlebots, which will provide stable and precise localization information by combining both the camera data and the system prediction values. The filter node was written in C_{++}. Details about the node will be discussed later.

There are a few ways that the Kalman filter can be implemented into Turtlebot deployment. The first method would be a centralized scheme. Data would be collected from the camera and Turtlebot velocity nodes to the Kalman node, which then publishes the updated state estimates to ROS. Subscribed turtlebots would then find their respective poses and act accordingly. This process would be:

1. Receive velocity from Turtlebot node
2. Receive measurement data from Camera node
3. Update state estimate
4. Publish state estimate

Message Complexity: 2 (Velocity, State Estimate).

This process would have a message complexity over the wifi network of 2 (Assuming that the camera node and central computer is wired). This is the method I currently have the Kalman filter coded for, however the second means of implementation would be preferred for several reasons.

The second and probably preferred method is to give each of the turtlebots their own Kalman filter node. The process would then be:

1. Update measured state if measured pose is sent by camera node
2. Update state estimations x_k
3. Predict state estimations for x_{k-1}

Message Complexity: 1 (poseWithName)

This method would be preferred because it provides a more distributed means of calculating the turtlebot's pose which is more true to our lab's distributed algorithm research. Another benefit is that the message complexity is lower because the turtlebot does not have to send its velocity data, as it already has it. Message complexity is important because we currently work through the wifi network, and higher message complexity means there may be a higher chance of data loss. Message complexity also increases discrete time increment required for each estimate. Lastly, since the method is distributed, the Kalman filter algorithm is individually executed. If there are many agents, the distributed calculations would be quicker.

2 Personal Contributions

My initial objective was to debug and implement Gu Peizhen's Kalman filter into our Turtlebot Deployment. I started debugging the previous Kalman filter by making a node that publishes fake poses and velocity with variance. The Kalman filter would take the fake poses and publish updated poses. I found that there was a problem with the posteriori estimate covariance matrix, P, as the number of time steps increased. At some point, values in P would suddenly jump orders of magnitude. I had trouble going in and debugging the code because it didn't use a linear algebra library. I rewrote the code, using Gu Peizhen's work (determined R, Q, state space functions etc.) using a linear algebra library called "Eigen".
My next goal was to implement the Kalman goal in the Turtlebot deployment. In order to do this, I chose a centralized scheme which would subscribe to velocity and pose measurement data and publish the updated estimates. The current Kalman filter is able to take in as many Turtlebots as necessary and publish updated estimates as long as the Turtlebots provide unique ids.
I want to change the scheme to a decentralized method where each of the Turtlebots have their own Kalman filter. This would be more robust for reasons previously mentioned, and would not require many changes.

3 Preliminaries

3.1 The Extended Kalman Filter

The Kalman filter is a popular filter/estimation scheme. The Kalman gain, K_g , is used to weight measured state vs predicted based on an evolving covariance matrix P. The extended version of the Kalman filter is used for non-linear system models, and is used by linearizing matrices A and B.

Our state space equations are non-linear.

$x_k = x_{k-1} + T * v * cos(\theta) + w_1$
$y_k = y_{k-1} + T * v * sin(\theta) + w_2$
$\theta_k = \theta_{k-1} + T * \omega + w3$

The Extended Kalman filter process is as follows:

Update Stage
1. Compute Kalman Gain
$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1}$

2. Update estimate with measurement z_k
$\hat{x}_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, 0))$

3. Update the error covariance
$P_k = (I - K_k H_k) P_k^-$

Estimate Stage
4. Estimate next state
$x_k = x_{k-1} + T * v * cos(\theta) + w_1$
$y_k = y_{k-1} + T * v * sin(\theta) + w_2$
$\theta_k = \theta_{k-1} + T * \omega + w3$

5. Project the error covariance ahead
$P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T$

4 Methodology

In this section, details of the filter will be discussed

4.1 Mathematical Model

mathematical model) *** This section belongs to Gu Peizhen. I plan on revising in the future.
For more details, see An introduction to the Kalman filter)Welch G, Bishop G.1995)(1)
Here we define the state space model of of the Turtlebots. (from "Applying Kalman filter to Robot Networks", Peizhen)
$x_k = x_{k-1} + T * v * cos(\theta) + w_1$
$y_k = y_{k-1} + T * v * sin(\theta) + w_2$
$\theta_k = \theta_{k-1} + T * \omega + w3$
Which yields:
$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \text{A} \begin{bmatrix} x_k \\ y_k \\ \theta \end{bmatrix} + B \begin{bmatrix} v_{k-1} \\ \omega_{k-1} \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$
The state space model is not linear. We can use the extended Kalman filter and linearize the matrices.
$$A_{[i,j]} = \frac{\partial f_{[i]}}{\partial x_{[j]}}(\hat{x}_{k-1}, \hat{u}_{k-1}, 0) = \begin{pmatrix} 1 & 0 & -T * v * sin(\Theta_{k-1}) \\ 0 & 1 & T * v * cos(\Theta_{k-1}) \\ 0 & 0 & 1 \end{pmatrix}$$
$$B_{[i,j]} = \frac{\partial f_{[i]}}{\partial u_{[j]}}(\hat{x}_{k-1}, \hat{u}_{k-}, 0) = \begin{pmatrix} cos(\Theta_{k-1}) & 0 \\ sin(\Theta_{k-1}) & 0 \\ 0 & T \end{pmatrix}$$
$$H_{[i,j]} = \frac{\partial h_{[i]}}{\partial x_{[j]}}(\hat{x}_k^-, 0) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

4.2 Interface Design

This section is subject to change since the implementation of the Kalman filter hasn't happened yet. The proposed Kalman filter node will be distributed. the relevant loop will include:

Camera:
1. Determine poseWithName
2. Publish poseWithName

TurtleBot - Kalman Filter
3. Subscribe to poseWithName
4. Update estimated pose
5. Update Covariance P
6. Move

4.3 Code Design

As previously mentioned, my motivation for rewriting the code was to utilize linear algebra libraries. I found it important to use linear algebra for ease of debugging and modification. For example, if in the future we decide to use a centralized Kalman filter, it will be much easier to modify when the code is in linear algebra format.

4.3.1 Period T

The time period used in the previous algorithm was set to 2. This is because the voronoi algorithm used by the robots sometimes required 2 seconds, so more frequent use of the Kalman filter would be excessive. The voronoi algorithm's are being worked on, and hopefully will be quicker in the future.
One modification to the Kalman filter for the future, is to execute when necessary. This would be done by recording the previous time and pose, and estimating/updating the pose based the time step.

4.3.2 Matrix P

The matrix P is now stable with implementation of the "Eigen" library.

5 Tips

This section is incomplete because the implementation of the Kalman filter is incomplete.

6 Pitfalls

Currently the Kalman filter is not implemented in deployment.
There were several pitfalls when writing the Kalman filter code. The initial code had problems with the P-matrix, which could not be determined. I had troubles when trying to utilize the linear algrebra libraries. I initially started using a library called "Armadillo", which required linear algebra packages, "LAPACK" and "BLAS". I was unable to get LAPACK and BLAS to work correctly, so I found an alternative; Eigen. Eigen was used completely with header files.

7 Conclusions and Future Decisions

The Kalman filter will soon be utilized to make Turtlebot Deployment more robust. The code will be altered to be decentralized for the benefits previously mentioned.

8 Notes for Variables

Currently in Code

```
#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 <math.h>
#include "PoseWithName.h"
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include "eigen/Eigen/Dense"
#include <vector>
#include <time.h>
#include <stdlib.h>

/*
    ------------------------------------------------------------------------------------------------

CREDIT
Credit to
Eigen for their opensource linear algebra library and headers
Alan for previous iteration of Kalman filter
ROS opensource
*/



/*

Notes for meeting-

```

```
-Report
-Finish multiple robotic intake
-Try on actual robots

*/

/*

Changelog

To Do
-clean up includes
-comment everything
-Multi-agent time based

Done
-Multi-agent capability
-Correct compilation and running

----------------------------------------------------------------------------------------
BEGIN
*/

using namespace std;
using namespace Eigen;
using Eigen::MatrixXd;

//***Remove later
const float T =2;
/*---------------------------------------------------------------------------------------
**  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);

/*
    ------------------------------------------------------------------------------------
```

```
**  Initialize Vector of variable matrices
**  This allows for multi-agent capability
**
**  index=0 corresponds to first robot, index=2 corresponds to second etc.
*/

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

/*------------------------------------------------------------------------------------------
**  Define Class "agent"
**  x:  x-coordinate
**  y:  y-coordinate
**  theta:  angle about the z-axis
**  velo:   linear velocity in direction of theta
**  omega:  angular velocity about z-axis
**  t:  time increment since last update
**  name:   name of agent (matches name published by cameras and turtlebot_base)
*/

class agent{
public:
    float x, y, theta, velo, omega, t;
string name;
};
vector<agent> agentVector;

/*------------------------------------------------------------------------------------------
**  Creates class Kalmanfilter
**  function poseCallback:  executed when PoseWithName is published
**  function iptCallback:   executed when TwistStamped is published
**  Subscribers and publishers initialized


    ***I think I can get rid of initialized x,y,theta etc..
*/

class Kalmanfilter
{
public:
    Kalmanfilter();
private:
    // Methods
    void poseCallback(const turtlebot_deployment::PoseWithName::ConstPtr& pose);
```

```
    void iptCallback(const geometry_msgs::TwistStamped::ConstPtr&);
    // ROS stuff
    ros::NodeHandle nh_ ;
    ros::Subscriber pos_sub_ ;
    ros::Subscriber ipt_sub_ ;
    ros::Publisher gl_pub_ ;
    ros::Publisher sf_pub_;
    // Other member variables
    turtlebot_deployment::PoseWithNamePtr newPose_;
    bool got_pose_;
    double theta,
    x,
    y;

}  ;

Kalmanfilter::Kalmanfilter():
got_pose_(false) ,
newPose_(new turtlebot_deployment::PoseWithName),
theta(0.0),
x(0.0),
y(0.0)
{

    pos_sub_= nh_.subscribe<turtlebot_deployment::PoseWithName>("PoseWithName", 10, &Kalmanfilter::poseCallback, this);
    ipt_sub_=nh_.subscribe<geometry_msgs::TwistStamped>("mobile_base/commands/velocity",10,&Kalmanfilter::iptCallback, this);

    gl_pub_ = nh_.advertise<turtlebot_deployment::PoseWithName>("/all_positions", 10, true);
        //std::cout<<"pose="<</<<"\n";
    sf_pub_=nh_.advertise<turtlebot_deployment::PoseWithName>("afterKalman",10,true);

}


/*----------------------------------------------------------------------------------------------------------------
**  function poseCallback
**  Runs when a PoseWithName is recieved
**  Updates Pose of Robot with identification name
*/

void Kalmanfilter::poseCallback(const turtlebot_deployment::PoseWithName::ConstPtr& posePtr)
{
/*----------------------------------------------------------------------------------------------------------------
**  size: number of robots detected by filter
**  iTemp:  Index of robot name in vector<string>robots.  Also corresponds to index of particular
**    robot specific matrices and "agent" class
```

```
**      -When this function is ran, iTemp is found in order to perform filter on iTemp index
*/

int size = robots.size();
int iTemp;
agent a;          //Temp agent used if a new agent is introduced

/*-------------------------------------------------------------------------------------------
**  iTemp search algorithm:
**  Determines if posePtr->name (name passed by poseWithName) is in vector<string>robots
**  and returns iTemp, it's index of that vector
*/
  if (size>0){
    for (iTemp=0; iTemp<size;iTemp++)
    {
      if (robots[iTemp]==posePtr->name)
      {
        break;
      }
    }
  }
  else {iTemp=0;
  }
    /*-------------------------------------------------------------------------------------------
    **  If posePtr->name is not found in vector<string>robots, the following add's the new
    **  robot information and initialization to the vector.
    */
        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)=9;
      P(1,1)=9;
      P(2,2)=9;
      //TESTBREAK***H=Matrix3f::Identity();        //Initialize Matrix H(

      //TESTBREAK***Hv.push_back(H);
      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);

    }
    else{
      /*
      **  This block runs if posePtr->name was found in found, ie
    if robot's id was previously detected.
      */

      //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);

  }

}

/*
    ------------------------------------------------------------------------------------------------

**  Function iptCallback:
**  This function has dual purpose subject to change*
**  1:  Takes twistStamped values and updates velo, and omega with
    respect to frame_id (robot name)
**  2:  Computes Kalmanfilter everytime a TwistStamped message is
    recieved
**  ***This function may change.  If it is determined that we want
    to compute Kalman filter with different conditions
**    such as when both poseWithName and ipt subscriptions are
    recieved, or if we want to publish at a set rate
**    In which case, the kalmanfilter process will become its own
    function.
*/

void Kalmanfilter::iptCallback(const geometry_msgs::TwistStamped::
    ConstPtr& ipt)
{
//Determine if subscribed .name is already in group.  If not, this
    initiallizes a new one
int size = robots.size();
int iTemp;
agent a;

/*
    ------------------------------------------------------------------------------------------------

**  Same iTemp search algorithm as in poseCallback
**  ***This algorithm may become a separate function
*/
  if (size>0){
```

```
    for (iTemp=0; iTemp<size;iTemp++)
    {
      if (robots[iTemp]==ipt->header.frame_id)
      {
        break;
      }
    }
  }
  else{iTemp=0;
  }
          if (iTemp==size)
    {

          robots.push_back(ipt->header.frame_id);
          agentVector.push_back(a);
          agentVector[iTemp].name=ipt->header.frame_id;

      P=Matrix3f::Zero();
      P(0,0)=9;
      P(1,1)=9;
      P(2,2)=9;
      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);
      //


    }
    else{
      agentVector[iTemp].velo=ipt->twist.linear.x;
            agentVector[iTemp].omega=ipt->twist.angular.z;
  }


/*  ***TO BE INDEPENDENT TIMER!~~~~
//void updateFilter(){
clock_t time;
time=clock();
agentVector[iTemp].t=clock();
*/
//TESTBREAK***int i,j,k,ad,i1,j1;
//TESTBREAK***float t1,t2,t3;

//Define Temperary matrices for KalmanFilter calculation
VectorXf Z(3);
Matrix3f temp;
X=Xv[iTemp];
XT=XTv[iTemp];
P=Pv[iTemp];

/*
**  ------------------------------Begin Kalman Filter Operation
    ------------------------------------
```

```
**     Composed of Update and Prediction stages
*/

//Stage 1
Z << agentVector[iTemp].x,agentVector[iTemp].y,agentVector[iTemp].theta;
X << XT(0)+T*agentVector[iTemp].velo*cos(agentVector[iTemp].theta),XT(1)+T*agentVector[iTemp].velo*sin(agentVector[iTemp].theta),XT(2)+T*agentVector[iTemp].omega;
cout<<"theta: "<<theta<<"\n";
//Stage 2
A << 1, 0, -T*agentVector[iTemp].velo*sin(agentVector[iTemp].theta),0, 1,T*agentVector[iTemp].velo*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;

//Stage 6
XT=X;

//Set Vectors
Xv[iTemp]=X;
XTv[iTemp]=XT;
Pv[iTemp]=P;


/*--------------------------------------------------------------------------------------------------------
**   Publish Block
**   Creates temperary "goalPose" ie, pose published by Kalmanfilter for deployment processing
*/

  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));

gl_pub_.publish(goalPose);
sf_pub_.publish(goalPose);

/*--------------------------------------------------------------------------------------------------------
**   Diagnostics Block
```

```
**    -Reports to terminal
*/

cout<<"Number of Robots: "<<size<<"\n";
cout<<"Robot #: "<<iTemp<<"\n";
std::cout<<"Measured: \n"<<Z<<"\n\n";
std::cout<<"Goal Pose\n"<<goalPose<<"\n----------\n\n\n\n";
std::cout
    <<"--------------------------------------------------------------------------------";

}

/*
**  Begin Main Routine
**  1:Initialize permanent matrices Q,R
**
*/
int main(int argc, char **argv)
{
//Initialize Matrix vectors
//Q R
Q(0,0)=.1;
Q(1,1)=.1;
Q(2,2)=.05;

R(0,0)=.1;
R(1,1)=.1;
R(2,2)=.05;
// Set initial temps
/* TESTBREAK***
//P(0,0)=9;
//P(1,1)=9;
//P(2,2)=9;
//Pv.push_back(P);
//Hv.push_back(H);
//X<<1,2,3;
//Xv.push_back(X);
//XT=X;
//XTv.push_back(XT);


//
//agent a;
//a.x=1;
//a.y=1;
//a.theta=1;
//a.omega=1;
//a.velo=1;
//robots.push_back("null");
//agentVector.push_back(a);
*/

    ros::init(argc, argv, "Kalmanfilter");    //Ros Initialize
    Kalmanfilter kalmanfilter;         //Kalmanfilter class initialization
    ros::Rate loop_rate(500);        //Set Ros frequency to 500/s (fast)
```

```
/*
**  Forever loop block
**  This repeatedly checks for new subscriptions and computes
    kalmanfilter in response
*/
    while (ros::ok()) {
         ros::spin();
         loop_rate.sleep();
     }
}

//END
```