Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Software Development Project

C library for filtering and prediction problems

Suhasini Venkatesh
Kaushik Manjunatha
Prabhudev Bengaluru Kumar

Supervised by

Sven Schneider

March 2021

Contents

1 Introduction **2**

2 A brief technical overview **2**

2.1 Kalman filter . . . 2

2.2 Extended Kalman filter . . . 5

3 Analysis of similarity **7**

4 Experiment and evaluation **10**

4.1 Use case 1: Position and velocity estimation of a point . . . 10

4.2 Use case 2: Position and orientation estimation of a differential drive 13

5 Using the library **20**

5.1 Commands to install GCC (GNU Compiler Collection) . . . 20

5.2 Commands to install GSL (GNU Scientific library) . . . 20

5.3 An example program . . . 21

5.4 Compiling and Linking . . . 23

6 List of routines available **24**

6.1 Kalman library . . . 24

6.2 Linear . . . 25

6.3 Differential drive . . . 27

7 Future Enhancements **29**

1 Introduction

The world is full of data and events that we want to measure and track, but we cannot rely on sensors to give us perfect information [1]. Sensors are noisy so as the world. The prediction helps in making a better estimate, but it also subject to noise. There are number of estimation algorithms used to the estimate the hidden variables' value of dynamic systems in the presence of uncertainty. Most of the algorithms are based on Bayesian probability. Bayesian probability determines, what is likely to be true based on past information[1]. Knowledge is uncertain, and we alter our beliefs based on the strength of the evidence[1]. Bayesian filters blend our noisy and limited knowledge of how a system behaves with the noisy and limited sensor readings to produce the best possible estimate of the state of the system[1]. Bayesian probability serves as the underlying principle for most of the filtering and prediction algorithms. This library provides the C routines that perform the atomic operations of the Bayesian filter independently, using which filtering and prediction problems of systems with different dynamics can be implemented.

2 A brief technical overview

Currently, the routines is the library is implemented after analysing the two important estimation algorithms: simple Kalman filter and extended Kalman filter. The underlying common atomic operations are implemented as C routines. The use of these routines in this manual is later described in detailed definitions of the functions, followed by example programs.

2.1 Kalman filter

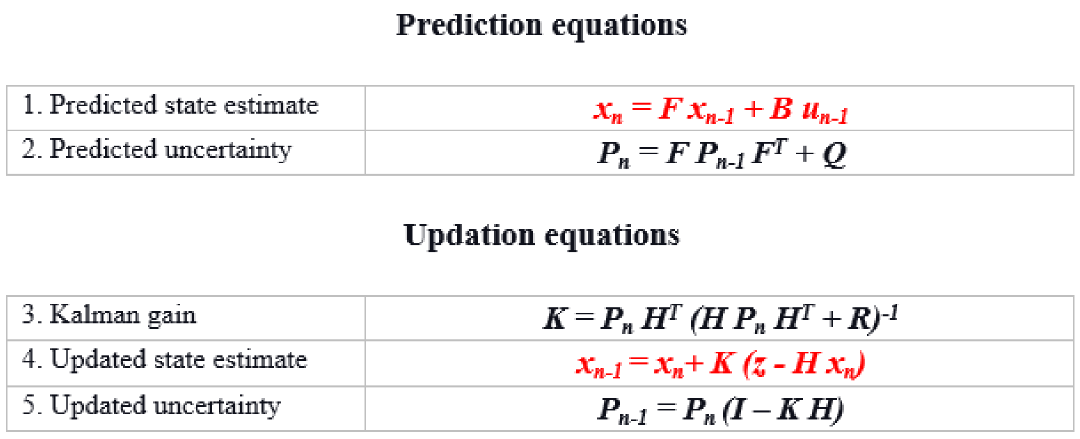
Kalman filter is a linear optimal state estimation method, which is known as one of the most famous Bayesian filter. The state equation is a linear representation of x_{n-1} and u_{n-1}. The observation equation is a linear representation of x_n and v_n. A dynamic model is presented with status equation and observation equation through the reliable estimation corrected by measurements. Kalman filter state equation is defined as follows:

$$x_n = Fx_{n-1} + Bu_{n-1} + w_n \tag{1}$$

Observation equation is defined as follows:

$$z_n = Hx_n + v_n \tag{2}$$

In the above formulas: $x_n, z_n, F, B, H, u_{n1}, w_n$, and v_n is the status vector, the observation vector, the status transition matrix, the observation matrix, the system control vector, the system noise vector, and the observation noise vector, respectively. The process noise vector w_{n-1} is assumed to be zero-mean Gaussian with the covariance Q and the measurement noise vector v_n is assumed to be zero-mean Gaussian with the covariance R.



Prediction equations

1. Predicted state estimate	$x_n = F\,x_{n-1} + B\,u_{n-1}$
2. Predicted uncertainty	$P_n = F\,P_{n-1}\,F^T + Q$

Updation equations

3. Kalman gain	$K = P_n\,H^T\,(H\,P_n\,H^T + R)^{-1}$
4. Updated state estimate	$x_{n-1} = x_n + K\,(z - H\,x_n)$
5. Updated uncertainty	$P_{n-1} = P_n\,(I - K\,H)$

Figure 1: *simple Kalman equations*

The Kalman filter produces the estimate of x_n at time n , given the initial estimate of x_0 , the series of measurement, $z_1, z_2, \ldots z_n$, and the information of the system described by F, B, H, Q,and R. Note that subscripts to these matrices are omitted here by assuming that they are invariant over time as in most applications.

Kalman filter algorithm consists of two stages: prediction and update [2]. The Kalman filter algorithm is summarized as follows: The predicted state estimate is evolved from the updated previous updated state estimate. The new term P is called the estimation uncertainty. It encrypts the error covariance that the filter thinks the estimate error hasKalmanNet. Estimation uncertainty becomes larger at the prediction stage due to the summation with Q , which means the filter is more uncertain of the state estimate after the prediction stepKalmanNet.

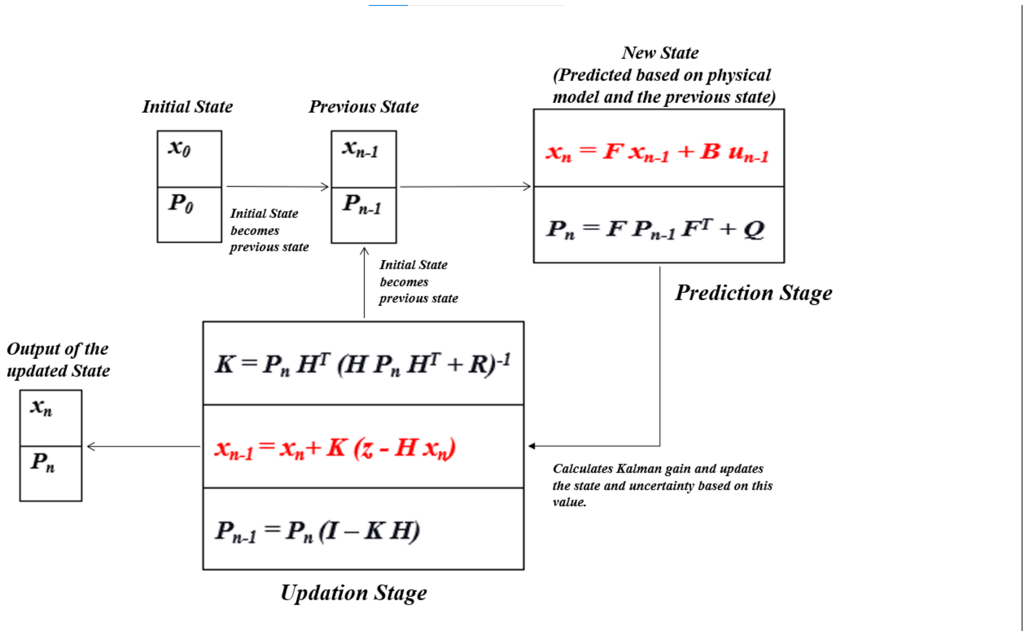

New State
(Predicted based on physical model and the previous state)
Initial State
Previous State
x_0
P_0
x_{n-1}
P_{n-1}
Initial State becomes previous state
$x_n = F x_{n-1} + B u_{n-1}$
$P_n = F P_{n-1} F^T + Q$
Initial State becomes previous state
Prediction Stage
Output of the updated State
x_n
P_n
$K = P_n H^T (H P_n H^T + R)^{-1}$
$x_{n-1} = x_n + K (z - H x_n)$
$P_{n-1} = P_n (I - K H)$
Calculates Kalman gain and updates the state and uncertainty based on this value.
Updation Stage

Figure 2: *Kalman filter overview*

In the update stage, the Kalman gain is computed which is the relative weight given to the measurements. With a high gain, the filter places more weight on the most recent measurements, and thus follows them more responsively. With a low gain, the filter follows the model predictions more closely. The Kalman gain is then multiplied with the measurement residual which is the difference between true measurement z and the estimated measurement Hx_n, to provide correction to the predicted estimate x_n . After it obtains the updated state estimate, the Kalman filter calculates the updated error covariance,[3] P_{n-1} , which will be used in the next prediction step. Note that the updated error covariance is smaller than the predicted error covariance, which means the filter is more certain of the state estimate after the measurement is utilized in the update stage[3]. Figure 2 represents the Kalman prediction and updation stages.

We need an initialization stage to implement the Kalman filter[3]. As initial values, we need the initial guess of state estimate, x_0 , and the initial guess of the error covariance matrix, P_0 [3]. Together with Q and R , x_0 and P_0 play an important role to obtain desired performance. There is a rule of thumb called "initial ignorance," which means that the user should choose a large P_0 for quicker convergence[3]. Finally, one can obtain implement a Kalman filter by implementing the prediction and update stages for each time step, $n = 1, 2, 3\ldots$, after the initialization of estimates[3]. Note that Kalman filters are derived based on the assumption that the process and measurement models are linear [3], i.e., they can be expressed with the matrices F, B, and H , and the process and measurement noise are additive Gaussian[3]. Hence, a Kalman filter provides optimal estimate only if the assumptions are satisfied[3].

2.2 Extended Kalman filter

The extended Kalman filter can be viewed as a nonlinear version of the Kalman filter that linearized the models about a current estimate[3]. It works by linearizing the system model for each update.

Consider the function $f(x) = x^2 - 2x$, this is a non-linear function. We want a linear approximation of this function so that we can use it in the Kalman filter[1]. During each Update of the Kalman filter we know its current state, so if we linearize the function at that value, we will have a close approximation[1].

Assume that our current state is $x = 15$. What can be a good linearization for this function?

We can use any linear function that passes through the curve at $(1.5, 0.75)$.[1]

For example, consider using $f(x) = 8x - 12.75$ as the linearization[1]. Then after applying linearization the approximation graph looks like as shown in the figure 3.

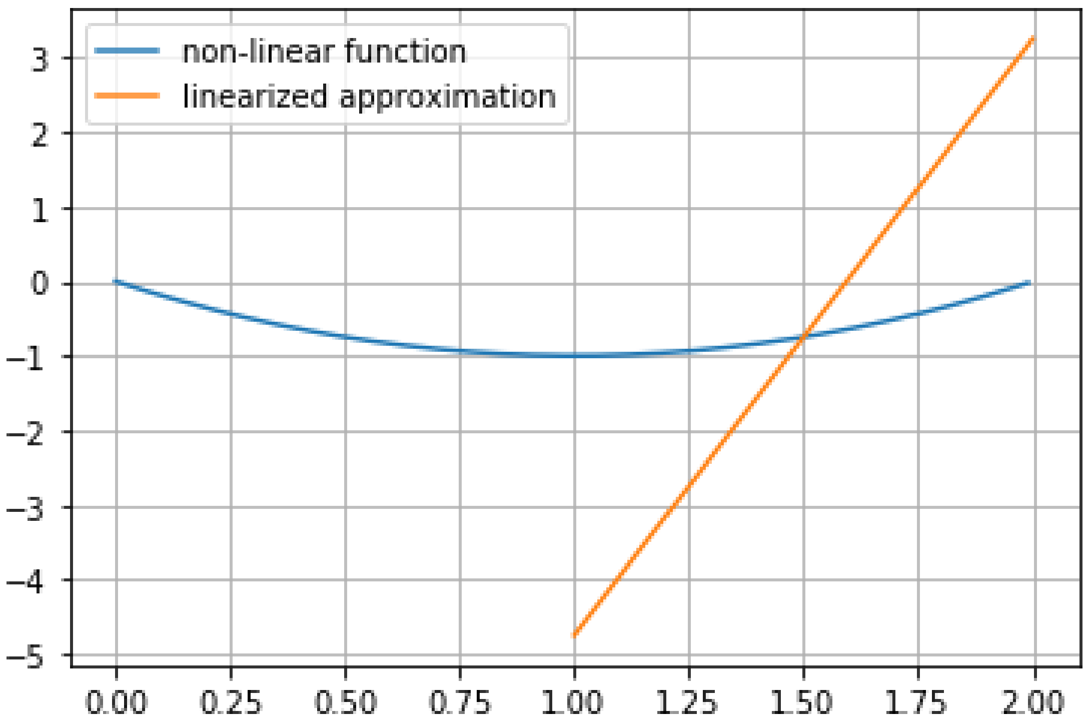


Figure 3: *Linearization : graph 1*

This is not a good linearization for $f(x)$. It is exact for $x = 1.5$, but quickly diverges when x varies by a small amount. [1] A much better approach is to use the slope of the function at the evaluation point as the linearization.[1]
We find the slope by taking the first derivative of the function:

$$f(x) = x^2 - 2x \tag{3}$$

$$\frac{dy}{dx} = 2x - 2 \tag{4}$$

So, the slope at $x = 1.5$ is 1. Substituting the slope, x and y values in the line equation we get the graph shown in the figure 4. This linearization is much better[1]. It is still accurate at $x = 1.5$, but the errors are very small as x changes.

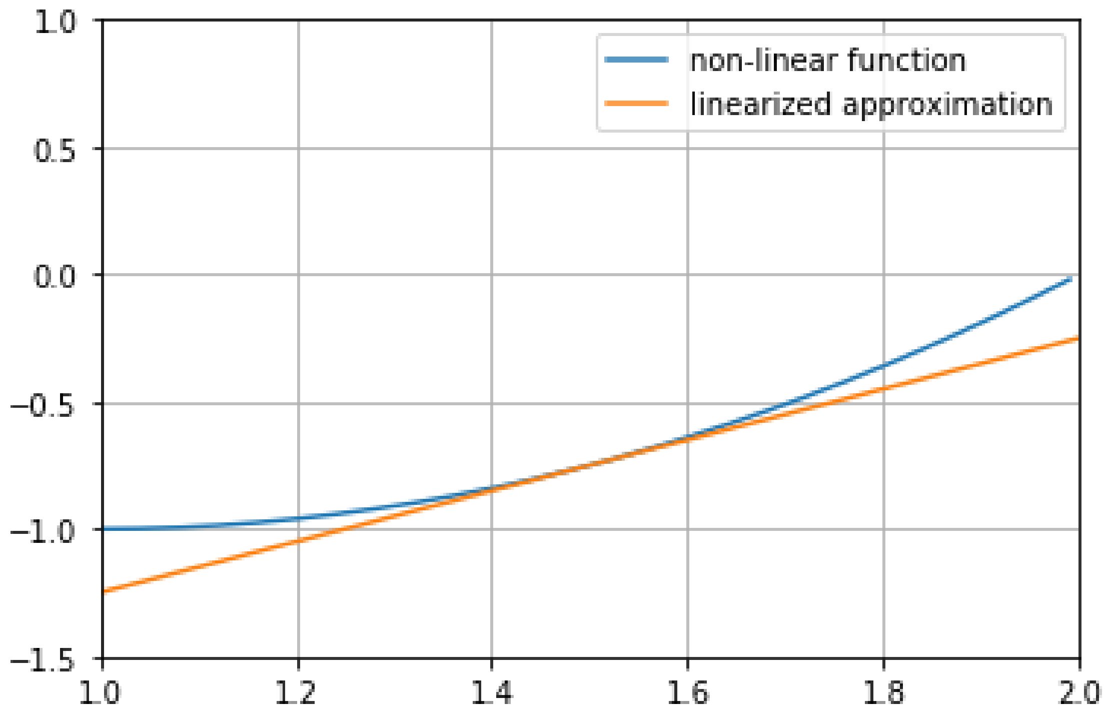


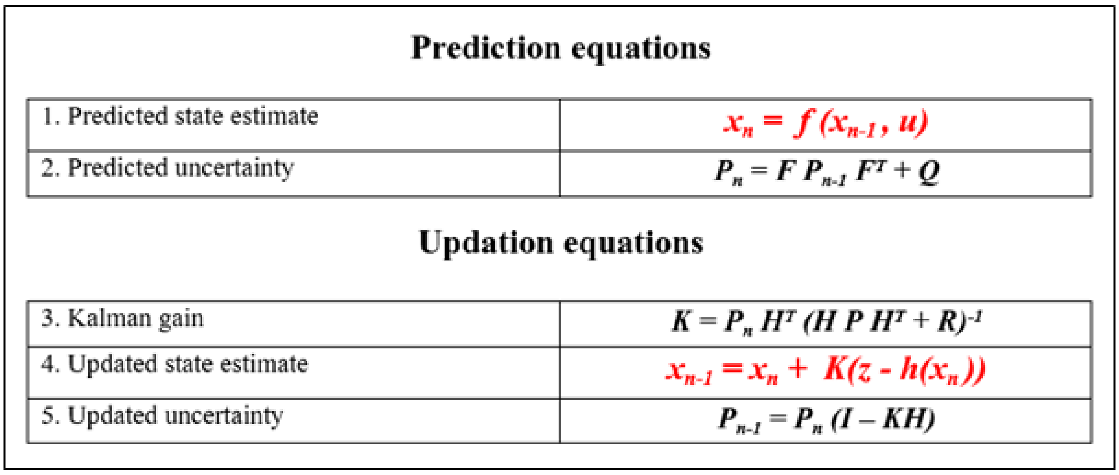
Figure 4: *Linearization : graph 2*

Figure 5 summarizes the equations used in extended Kalman filter to linearize a non-linear system and perform filtering and prediction.

3 Analysis of similarity

As we can see in figure 1 and figure 5 there are two minor changes to the Kalman filter equations with respect to extended Kalman filter, which are highlighted in red.

In the Kalman filter, Fx is how we compute the new state based on the old state. However, in a nonlinear system we cannot use linear algebra to compute this transition. So instead we hypothesize a nonlinear function $f(x, u)$ which performs this function[1]. Likewise, in the Kalman filter we convert the state to a measurement with the linear function Hx[1]. For the extended Kalman filter we replace



Prediction equations

1. Predicted state estimate	$x_n = f(x_{n-1}, u)$
2. Predicted uncertainty	$P_n = F P_{n-1} F^T + Q$

Updation equations

3. Kalman gain	$K = P_n H^T (H P H^T + R)^{-1}$
4. Updated state estimate	$x_{n-1} = x_n + K(z - h(x_n))$
5. Updated uncertainty	$P_{n-1} = P_n (I - KH)$

Figure 5: *Extended Kalman filter overview*

this with a nonlinear function $h(x)$[1]. The rest of the equations are unchanged, so $f(x, u)$ and $h(x)$ must produce a matrix that approximates the values of the matrices F and H at the current value for state[1]. This can be computed based on the partial derivatives of the state and measurements functions. State transition function is computed by taking the partial derivative of the function $f(x, u)$ with respect to the state x at time n.

$$F \equiv \frac{\partial f}{\partial x} | x, u \tag{5}$$

Now we can turn our attention to the noise. The noise is in the control space, this should be mapped from control space to state space. We choose our noise model M, which depicts the error in the system. For the non-linear motion model is we do not try to find a closed form solution to convert from control space to state space, but instead linearize it with a Jacobian i.e.., taking partial derivative of the function $f(x, u)$ with respect to the control u and the resultant matrix is named V. Then using VMV^T equation to find the noise matrix Q to contribute to the uncertainty P.

$$V \equiv \frac{\partial f}{\partial u} | x, u \tag{6}$$

$$Q = VMV^T \tag{7}$$

Hence the prediction equations are reduced as follows:

$x_n = f(x_{n-1}, u)$	$x_n = F\,x_n + V\,u$
$P_n = F\,P_{n-1}\,F^T + Q$	$P_n = F\,P_{n-1}\,F^T + Q$ *(remains same)*

Now after taking the partial derivative of the functions to compute state transition matrix, control matrix and noise matrix the equation can be reduced to the simple Kalman filter form.

The measurement function is used to take the system state xn and map it from state space to the measurement space. For a system with non-linear system dynamics the measurement function is also non-linear and needs to be linearized at some point. Again, the best way to linearize an equation at a point is to find its slope, which is by taking its derivative, where each element in the matrix is the partial derivative of the function h with respect to the variables x.

$$H \equiv \frac{\partial h}{\partial x}|x \tag{8}$$

Updation equations is now reduced as follows: This way the equations of the

$K = P_n\,H^T\,(H\,P\,H^T + R)^{-1}$	$K = P_n\,H^T\,(H\,P\,H^T + R)^{-1}$ *(remains same)*
$x_{n-1} = x_n + K(z - h(x_n))$	$x_{n-1} = x_n + K(z - H\,x_n)$
$P_{n-1} = P_n\,(I - KH)$	$P_{n-1} = P_n\,(I - KH)$ *(remains same)*

extended Kalman filter can be reduced to the simple Kalman filter equations form. These reduced common equations are implemented as C routines such that it can be conveniently reused.

4 Experiment and evaluation

4.1 Use case 1: Position and velocity estimation of a point

Lets consider a problem of a moving point in 2-dimension space. This is linear behaviour. The states of the point estimated in this case are position and velocity

$$x = \begin{bmatrix} Px \\ Py \\ Vx \\ Vy \end{bmatrix}$$

We are interested to calculate the position, we need velocity to calculate the position. If we assume that the point is moving in a constant velocity then the new position is represented by the equation

$$position_{new} = position_{initial} + velocity * delta_t$$

where, delta_t is the amount of time that's passed between the initial measurement and the current measurement.
Therefore, our state transition matrix can be formulated as,

$$F = \begin{bmatrix} 1 & 0 & delta_t & 0 \\ 0 & 1 & 0 & delta_t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Such that, only position is updated and the velocity remains constant.

$$Fx_{n-1} = \begin{bmatrix} Px + Vx * delta_t \\ Py + Vy * delta_t \\ Vx \\ Vy \end{bmatrix}$$

Control vector u represents the action that is affecting the object's state. In this case it can be acceleration which can be either increased or decreased. Therefore the control vector u is,

$$u = \begin{bmatrix} ax \\ ay \end{bmatrix}$$

We need to formulate a control matrix which transforms the control vector from control space to state space such that it can be added to the state.
Newton law's of motion says,

$$position_{new} = position_{initial} + velocity * delta_t + 0.5 * acceleration * (delta_t)^2$$

Therefore, our control matrix can be formulated as,

$$B = \begin{bmatrix} 0.5 * (delta_t)^2 \\ 0.5 * (delta_t)^2 \\ delta_t \\ delta_t \end{bmatrix}$$

Such that,

$$Bu = \begin{bmatrix} 0.5 * ax * (delta_t)^2 \\ 0.5 * ay * (delta_t)^2 \\ ax * delta_t \\ ay * delta_t \end{bmatrix}$$

Hence the state is estimated using the first Kalman prediction equation,

$$x_n = F * x_{n-1} + B * u$$

Everything is fine if the state evolves based on its own properties[4]. Everything is still fine if the state evolves based on external forces, so long as we know what those external forces are[4]. We can model the uncertainty associated with the world (i.e. things we aren't keeping track of) by adding some new uncertainty after

every prediction step[4]. P is the object covariance matrix. This represents the idea that the state of the system changes over time, but we do not know the exact details of when/how those changes occur, and thus we need to model them as a random process[5].
The second prediction equation is,

$$P_n = F * P_{n-1} * F^T + Q$$

where F is the state transition matrix and $F * P_{n-1} * F^T$ gives the state uncertainty and Q is the process uncertainty added.
At the beginning P will be an Identity matrix which is scaled by the state transition matrix at every iteration and Q is added to this.
Q is computed using the formula,

$$Q = BMB^T$$

Where M is the noise model, which is a diagonal matrix with the variances of the control inputs in the diagonal.

Observation matrix is the transformation matrix that transforms the state from state space to measurement space. Therefore H is simply an rectangular matrix of dimension $n_z X n_x$ with one's in the main diagonal and rest of the elements initialized to zero, where n_z is the number of actual measurements from the sensors and n_x is the number of state variables.
Kalman gain is the important factor in the Kalman filtering, it tells how much we have to change the estimate by the given measurement. If Kalman gain is high, we are confident about our measurements, were confident that the information we're obtaining is good enough for us to update/change our state estimates[6].

$$K = P_n H^T (HPH^T + R)^{-1}$$

where R is the measurement noise recorded during measurement. The matrix is represented as diagonal matrix with the sensor variations as the diagonal elements, H is the observation matrix and P_n is the estimated covariance matrix from prediction equation, this tells us the ”variability” of the state.

If P_n is large, then it means that the state is estimated to changes a lot. So you need to be able to change your estimates with new measurements. So the Kalman gain is high.

If P_n is small, then it means then it means the state estimated doesn’t change much so you don’t want to alter the estimates at every time instance. So the Kalman gain is low.

So the Kalman Gain depends on the covariances of our object and the measurement. That is, the more uncertain we are about our prediction, the more we allow our measurement to change the prediction. And the more uncertain we are about our measurement, the less we allow the measurement to change our prediction. In this equation We bring our predicted object state closer to our observed measurement, where the amount of influence the measurement has on our beliefs depends on how uncertain we are about our prediction and our measurement.

The next set of equations in the updation state is to to update the state and covariance based on the kalman gain.

$$x_{n-1} = x_n + K(z - Hx_n)$$

$$P_{n-1} = P_n(I - KH)$$

where z is the actual measurement from the sensors and I is the identity matrix.

4.2 Use case 2: Position and orientation estimation of a differential drive

Lets consider a problem of robot localization. Assuming that our robot is wheeled, which means that it moves by turning it’s wheels. When it does so, the robot turn around the rear axle while moving forward. This is non-linear behaviour.

At a first approximation an automobile steers by turning the front tires while moving forward[7]. The front of the drive moves in the direction that the wheels are pointing while turning around the rear tires. This simple description is complicated by issues such as slippage due to friction, the differing behavior of the rubber tires

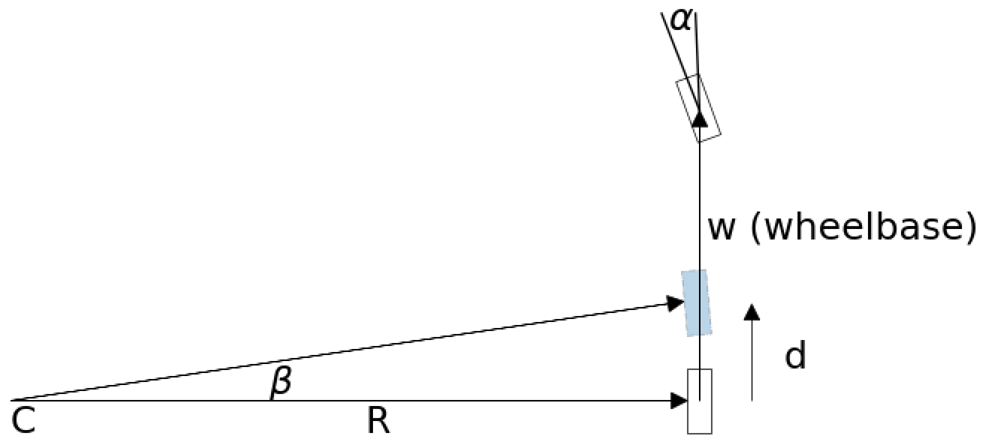


Figure 6: *Robot localization*

at different speeds, and the need for the outside tire to travel a different radius than the inner tire[1]. To Accurately model steering we need a set of differential equations. Referring to the figure 6, the front tire is pointing in direction α. Over a short time period the drive moves forward and the rear wheel ends up further ahead and slightly turned inward, as depicted with the blue shaded tire[1]. Over such a short time frame we can approximate this as a turn around a radius R[1]. Formula to compute the turn angle β is :

$$\beta = \frac{d}{w} \tan \alpha$$

Formula to compute the turning radius is:

$$R = \frac{d}{\beta}$$

The distance d is given by

$$d = v\Delta t$$

where, v is the forward velocity.
If θ is our current orientation then we can compute the position C as

$$C_x = x - R\sin(\theta)$$

$$C_y = y + R\cos(\theta)$$

After moving forward for time Δt the new position and orientation of the robot is

$$x = C_x + R\sin(\theta + \beta)$$

$$y = C_y - R\cos(\theta + \beta)$$

$$\theta = \theta + \beta$$

Substituting C_x & C_y, we get equations:

$$x = x - R\sin(\theta) + R\sin(\theta + \beta)$$

$$y = y + R\cos(\theta) - R\cos(\theta + \beta)$$

$$\theta = \theta + \beta$$

These equations represent the motion of the robot and its is evident that the system is non-linear by seeing the equations.

In this example, we need to find position and orientation of the robot. Therefore our state variable X is,

$$X = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

We can include velocity also to our state but that makes the problem complicated. Therefore we will take velocity and steering angle as our control input u.

$$u = \begin{bmatrix} v \\ \alpha \end{bmatrix}$$

Equation for the system with nonlinear motion model.

$$\hat{x} = x + f(x, u)$$

We can expand the above given motion equations as

$$\begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{\theta} \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} + \begin{bmatrix} -R\sin(\theta) + R\sin(\theta + \beta) \\ R\cos(\theta) - R\cos(\theta + \beta) \\ \beta \end{bmatrix}$$

We linearize this with the derivative at x:

$$f(x, u) \approx x + \frac{\partial f(x, u)}{\partial x}$$

We replace $f(x, u)$ with our state X, and the derivative is the Jacobian of f.

$$F = \frac{\partial f(x, u)}{\partial x} = \begin{bmatrix} \frac{\partial \dot{x}}{\partial x} & \frac{\partial \dot{x}}{\partial y} & \frac{\partial \dot{x}}{\partial \theta} \\ \frac{\partial \dot{y}}{\partial x} & \frac{\partial \dot{y}}{\partial y} & \frac{\partial \dot{y}}{\partial \theta} \\ \frac{\partial \dot{\theta}}{\partial x} & \frac{\partial \dot{\theta}}{\partial y} & \frac{\partial \dot{\theta}}{\partial \theta} \end{bmatrix}$$

State transition matrix F in this case is the jacobian of the function $f(x, u)$ with respect to state X, which is:
The noise is in our control input, so it is in control space[1]. When we command a specific velocity and steering angle, we need to convert that into errors in x, y, θ. In a real system this might vary depending on velocity, so it will need to be

$$\begin{bmatrix} 1 & 0 & -\frac{w\cos(\theta)}{\tan(a)} + \frac{w\cos\left(\frac{tv\tan(a)}{w}+\theta\right)}{\tan(a)} \\ 0 & 1 & -\frac{w\sin(\theta)}{\tan(a)} + \frac{w\sin\left(\frac{tv\tan(a)}{w}+\theta\right)}{\tan(a)} \\ 0 & 0 & 1 \end{bmatrix}$$

recomputed for every prediction[1].

The noise model for this case is chosen as:

$$M = \begin{bmatrix} 0.01v^2 & 0 \\ 0 & \sigma_\alpha^2 \end{bmatrix}$$

$$V = \frac{\partial f(x,u)}{\partial u} = \begin{bmatrix} \frac{\partial \dot{x}}{\partial v} & \frac{\partial \dot{x}}{\partial \alpha} \\ \frac{\partial \dot{x}}{\partial v} & \frac{\partial \dot{y}}{\partial \alpha} \\ \frac{\partial \dot{\theta}}{\partial v} & \frac{\partial \dot{\theta}}{\partial \alpha} \end{bmatrix}$$

$$Q = VMV^T$$

Control matrix V in this case is the jacobian of the function $f(x,u)$ with respect to control u, which is:

$$\begin{bmatrix} t\cos\left(\frac{tv\tan(a)}{w}+\theta\right) & \frac{tv\left(\tan^2(a)+1\right)\cos\left(\frac{tv\tan(a)}{w}+\theta\right)}{\tan(a)} - \frac{w\left(-\tan^2(a)-1\right)\sin(\theta)}{\tan^2(a)} + \frac{w\left(-\tan^2(a)-1\right)\sin\left(\frac{tv\tan(a)}{w}+\theta\right)}{\tan^2(a)} \\ t\sin\left(\frac{tv\tan(a)}{w}+\theta\right) & \frac{tv\left(\tan^2(a)+1\right)\sin\left(\frac{tv\tan(a)}{w}+\theta\right)}{\tan(a)} + \frac{w\left(-\tan^2(a)-1\right)\cos(\theta)}{\tan^2(a)} - \frac{w\left(-\tan^2(a)-1\right)\cos\left(\frac{tv\tan(a)}{w}+\theta\right)}{\tan^2(a)} \\ \frac{t\tan(a)}{w} & \frac{tv\left(\tan^2(a)+1\right)}{w} \end{bmatrix}$$

After deriving the state transition matrix, control matrix and noise, our final form of prediction equation reduces to:

$$x_n = Fx_{n-1} + Vu$$

$$P_n = FP_{n-1}F^T + Q$$

For this problem we are assuming that we have a sensor that receives a noisy bearing and range to multiple known locations in the landscape[1].
The measurement model must convert the state

$$X = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

into a range and bearing to the landmark.

$$range = r = \sqrt{(p_x - x)^2 + (p_y - y)^2}$$

$$bearing = \phi = \arctan \frac{p_y - y}{p_x - x} - \theta$$

Thus, our function is

$$x = h(x, p) + MeasurementNoise$$

$$x = \begin{bmatrix} \sqrt{(p_x - x)^2 + (p_y - y)^2} \\ \arctan \frac{p_y - y}{p_x - x} - \theta \end{bmatrix} + MeasurementNoise$$

Observation matrix H in this case is the jacobian of the function $h(x)$ with respect to state X, which is:

$$\begin{bmatrix} \frac{-px+x}{\sqrt{(px-x)^2+(py-y)^2}} & \frac{-py+y}{\sqrt{(px-x)^2+(py-y)^2}} & 0 \\ -\frac{-py+y}{(px-x)^2+(py-y)^2} & -\frac{px-x}{(px-x)^2+(py-y)^2} & -1 \end{bmatrix}$$

Measurement noise is defined in measurement space, hence it is linear. It is reasonable to assume that the range and bearing measurement noise is independent, hence

$$R = \begin{bmatrix} \sigma^2_{range} & 0 \\ 0 & \sigma^2_{bearing} \end{bmatrix}$$

After deriving the observation matrix our final form of updation equation reduces to:

$$K = P_n H^T (H P_n H^T + R)^{-1}$$

$$x_{n-1} = x_n + K(z - H x_n)$$

$$P_{n-1} = P_n(I - KH)$$

Dimensions of the terms used in the equations

Term	Name	Dimensions
x	State Vector	$(n_x, 1)$
F	State Transition Matrix	(n_x, n_x)
B	Control Matrix	(n_x, n_u)
u	Control Input	$(n_u, 1)$
P	Estimate Uncertainty	(n_x, n_x)
Q	Process Noise Uncertainty	(n_x, n_x)
R	Measurement Uncertainty	(n_z, n_z)
K	Kalman Gain	(n_x, n_x)
H	Observation Matrix	(n_z, n_x)
z	Measurement	$(n_z, 1)$
w	Process noise Vector	$(n_x, 1)$

5 Using the library

The routines have been written from scratch in C and present a unique approach to solve the filtering and prediction problems with different system dynamics.
Note:

- The below installation steps covers the commands to install libraries in Ubuntu 16.04 platform.
- To able to add new repositories and install packages on your Ubuntu system, you must be logged in as root or user with "*sudo*" privileges

5.1 Commands to install GCC (GNU Compiler Collection)

The GNU Compiler Collection (GCC) is a collection of compilers and libraries for C, C++, Objective-C, Fortran, Ada, Go, and D programming languages.

1. Start by package update

```
$  sudo apt update
```

2. Install "build-essential" package : This package contains the GCC compiler and a lot of libraries and other utilities required for compiling software.

```
$  sudo apt install build-essential
```

3. Install manual pages about using GNU (optional step)

```
$  sudo apt-get install manpages-dev
```

4. Validate the installation

```
$  gcc --version
```

5.2 Commands to install GSL (GNU Scientific library)

The GNU Scientific Library (GSL) is a collection of routines for numerical computing.

1. Start by package update

```
$ sudo apt update
```

2. Install GSL binary package

```
$ sudo apt install gsl-bin
```

3. Install GSL development package

```
$ sudo apt-get install libgsl-dev
```

4. Install GSL debug symbols package

```
$ sudo apt-get install libgsl-dbg
```

5. Install GSL library package

```
$ sudo apt-get install libgsl2
```

6. Validate the installation

```
$ gsl-config --version
```

5.3 An example program

To predict the state of system by computing the required input matrices for a predicting the system state of a moving point in one dimension.

```
#include<stdio.h>
#include "linear.h"
#include "kalmanLibrary.h"
void main()
{
    int i, j;
    float *state_array=malloc(2 * sizeof(float));
```

```
// defining the initial position and initial velocity
state_array[0] = 10;
state_array[1] = 3;

// allocating memory and initializing state variables
gsl_vector * state = gsl_vector_alloc (2);
for (i = 0; i < 2; i++)
{
     gsl_vector_set (state, i, state_array[i]);
}

float *control_array=malloc(1 * sizeof(float));

//acceleration at which the body is moving
control_array[0] = 4;

// allocating memory for control variables
gsl_vector * control = gsl_vector_alloc (1);
gsl_vector_set (control, 0, control_array[0]);

//defining the time interval at which the measurements are made
float delta_t = 2;

// formulating state transition matrix
gsl_matrix * state_transition_matrix = gsl_matrix_calloc (2, 2);
state_transition_linear_pos_vel(delta_t, state_transition_matrix);

// formulating control matrix
gsl_matrix * control_matrix = gsl_matrix_calloc (2, 1);
control_matrix_linear_acc(delta_t, control_matrix);

// predict state
gsl_vector * predicted_state = gsl_vector_calloc (2);
predict_state(state_transition_matrix, state, control_matrix,
    control, predicted_state);
    printf("\n Predicted state:\n");
    for (i = 0; i < 2; i++)
    {
      printf ("%g\t", gsl_vector_get (predicted_state, i));
      printf("\n");
    }
}
```

5.4 Compiling and Linking

GCC compiler is used for compiling the C program. The header files used in the program should be linked to the program file by passing the files as a parameter to the compiling command. For example, the above program uses linear and kalmanLibrary header file therefore the corresponding files are passed as the parameters with the test file which needs to be compiled. GSL library is also linked to the file through command line.

```
$ gcc Test_file.c linear.c kalmanLibrary.c ‘pkg-config --cflags --libs
    gsl‘ -o Test_file
```

The resultant object code is stored in the object file Test file, which can be further executed using the below command.

```
$ ./Test_file
```

6 List of routines available

6.1 Kalman library

This library contains the basic filtering and prediction operations used in any Bayesian filters. These functions are defined in the header file **kalmanLibrary.h**

To use any function listed below, the program should include the kalmanLibrary header file in the beginning of the program.

```
void predict_state(const gsl_matrix * state_transition_matrix,
    gsl_vector * updated_state, const gsl_matrix * control_matrix, const
    gsl_vector * control, gsl_vector * predicted_state)
```

This function predicts the next state of system with any model dynamics, provided with the parameters such as state transition matrix, updated state, control matrix and control vector. The result is stored in the parameter predicted state.

```
void predict_covariance(const gsl_matrix * state_transition_matrix,
    gsl_matrix * covariance_matrix, const gsl_matrix *
    process_uncertainity, gsl_matrix * predicted_covariance)
```

This function predicts the uncertainty of a system with any model dynamics, provided with the parameters such as state transition matrix, covariance matrix, process uncertainity and the result is stored in the parameter predicted covariance.

```
void compute_kalman_gain(gsl_matrix * predicted_covariance, gsl_matrix
    * observation_matrix, gsl_matrix * measurement_uncertainity,
    gsl_matrix * kalman_gain)
```

This function computes the Kalman gain of a system with any model dynamics, provided with the parameters such as predicted covariance, observation matrix, measurement uncertainity and the result is stored in the parameter kalman gain.

```
void update_state(gsl_vector * predicted_state, gsl_matrix *
    kalman_gain, gsl_matrix * observation_matrix, gsl_vector *
    measurement_vector, gsl_vector * updated_state)
```

This function updates the state of the system with any model dynamics based on the Kalman gain belief, provided with the parameters such as predicted state, kalman gain, observation matrix and measurement vector. The result is stored in the parameter updated state.

```
void update_covariance(gsl_matrix * predicted_covariance, gsl_matrix *
    kalman_gain, gsl_matrix * observation_matrix, gsl_matrix *
    Identity_matrix, gsl_matrix * updated_covariance)
```

This function updates the covariance of system with any model dynamics based on the Kalman gain belief, provided with the parameters such as predicted covariance, kalman gain, observation matrix and Identity matrix. The result is stored in the parameter updated covariance.

6.2 Linear

This library contains the operations that compute the matrices and vectors required to predict or update the hidden variables of a linear system which estimates the position and velocity of a point in one, two or three dimensions. These functions are defined in the header file **linear.h**

To use any function listed below, the program should include the linear header file in the beginning of the program.

```
void state_transition_linear_pos_vel(float delta_t, gsl_matrix *
    state_transition_matrix)
```

This function formulates the state transition matrix to estimate position and velocity for a linear system in either one, two or three dimensions, provided with the parameters such as Δt, time stamp at which the state of the system is measured. The result is stored in the state transition matrix.

```
void control_matrix_linear_acc(float delta_t, gsl_matrix *
    control_matrix)
```

This function formulates the control matrix for a linear system with acceleration as input in either one, two or three dimensions, provided with the parameters such as Δt, time stamp at which the state of the system is measured. The result is stored in the control matrix.

```
void process_uncertainty_linear_acc(float variance, gsl_matrix *
    control_matrix, gsl_matrix * process_uncertainty)
```

This function formulates the process uncertainty for a linear system with acceleration as input in either one, two or three dimensions, provided with the parameters such as variance, the variance of the input factors and control matrix. The result is stored in the process uncertainty.

```
void observation_matrix_linear_pos_vel(gsl_matrix * observation_matrix)
```

This function formulates the observation matrix for a linear system in either one, two or three dimensions to map the state from state space to measurement space . The result is stored in the observation matrix.

```
void measurement_uncertainity_linear_pos_vel(const float Variance[],
    gsl_matrix * measurement_uncertainty)
```

This function formulates the measurement uncertainty for a linear system for the measured position and velocity in a linear system, in either one, two or three dimensions, provided with the parameters such as variance, the variance of the sensors used to measure the state. The result is stored in the measurement uncertainty.

6.3 Differential drive

This library contains the operations that compute the matrices and vectors required to predict or update the hidden variables of a differential drive robot (non-linear system) which estimates the positions x, y and the orientation theta of a differential drive robot given velocity v and steering angle alpha as the control input. These functions are defined in the header file **differential_drive.h**

To use any function listed below, the program should include the differential drive header file in the beginning of the program.

```
void state_transition_matrix_diff_drive(float delta_t, float
    wheel_base, gsl_vector * state, gsl_vector * control, gsl_matrix *
    state_transition)
```

This function formulates the state transition matrix to estimate position and orientation of a differential drive robot, provided with the parameters such as Δt, time stamp at which the state of the system is measured, wheel base, the current state and the control vector. The result is stored in the parameter state transition matrix.

```
void control_matrix_diff_drive(float delta_t, float wheel_base,
    gsl_vector * state, gsl_vector * control, gsl_matrix *
    control_matrix)
```

This function formulates the control matrix for a linear system with acceleration as input in either one, two or three dimensions, provided with the parameters such as Δt, time stamp at which the state of the system is measured. The result is stored in the parameter control matrix.

```c
void process_uncertainity_diff_drive(float variance[], gsl_vector *
    control, gsl_matrix * control_matrix ,gsl_matrix *
    process_uncertainity)
```

This function formulates the process uncertainty for a differential drive robot with velocity and steering angle as input, provided with the parameters such as variance, the variance of the input factors and the control matrix. The result is stored in the parameter process uncertainty.

```c
void observation_matrix_diff_drive(float landmark_positions[],
    gsl_vector * state, gsl_matrix * observation_matrix)
```

This function formulates the observation matrix for a differential drive robot to map the state from state space to measurement space, provided with the parameter landmark positions which is the values of range and bearing measured by the sensors . The result is stored in the parameter observation matrix.

```c
void measurement_uncertainity_diff_drive(float variance[], gsl_matrix *
    measurement_uncertainity)
```

This function formulates the measurement uncertainty for a differential drive robot for the measured range and bearing, provided with the parameters such as variance, the variance of the sensors used to measure the state. The result is stored in the parameter measurement uncertainty.

7 Future Enhancements

Currently, the library is based on two important bayesian filters, namely Kalman and Extended Kalman filter. Also this library includes specific functionalities for the use case of moving point position velocity estimation and differential drive position and orientation estimation.The library can further be expanded by analysing and adding the functionalities common with respect to other filters such as unscented Kalman filter, particle filter so on and so forth. More use case specific functionalities can also be further defined.

References

[1] R. Labbe. (2020) Kalman-and-bayesian-filters-in-python. [Online]. Available: https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python

[2] Kalman filter tutorial. [Online]. Available: https://www.kalmanfilter.net/default.aspx

[3] Y. Kim. (2018) Introduction to kalman filter and its applications. [Online]. Available: 10.5772/intechopen.80600

[4] T. Babb. (2015) how-a-kalman-filter-works-in-pictures. [Online]. Available: http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/

[5] R. E. Kalman, “A new approach to linear filtering and prediction problems,” *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[6] Raghavan. (2019) Kalman filter explained. [Online]. Available: https://medium.com/@raghavan99o/kalman-filter-explained-example-95b3f2d324ee

[7] T. Lacey. (2019) Tutorial: The kalman filter. [Online]. Available: http://web.mit.edu/kirtley/kirtley/binlustuff/literature/control/Kalmanfilter.pdf

[8] T. Basar, *A New Approach to Linear Filtering and Prediction Problems*, 2001, pp. 167–179.

[9] Kalman filter. [Online]. Available: https://en.wikipedia.org/wiki/Kalman_filter

[10] M. OpenCourseWare. (2008) Lec 16 - mit 18.085 computational science and engineering i. [Online]. Available: https://www.youtube.com/watch?v=d0D3VwBh5UQ

[11] G. B. Greg Welch. (2001) An introduction to the kalman filter. [Online]. Available: http://www.cs.unc.edu/~tracker/media/pdf/SIGGRAPH2001_CoursePack_08.pdf

[12] S. Srinivasan. (2018) Kalman filter - an algorithm for making sense from the insights of various sensors fused together. [Online]. Available: https://towardsdatascience.com/kalman-filter-an-algorithm-for-making-sense-from-the-insights-of-various-sensors-fused-together-