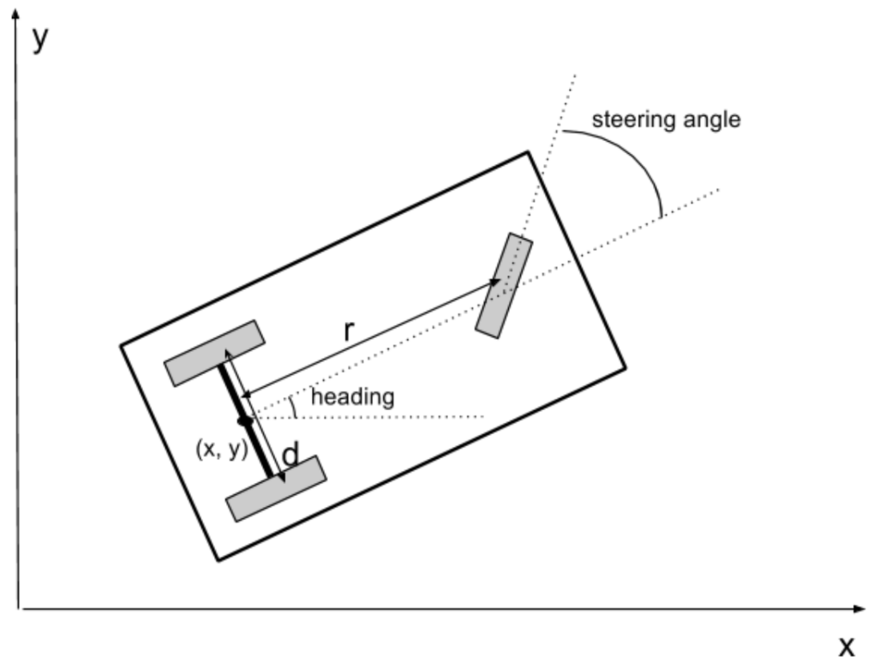
Documentation of pose estimator API:

July 21, 2018

1 Introduction:

The pose estimator API is designed to estimate the 2D position(x, y, heading) of a mobile platform based on odometry information using wheel encoder and gyroscope data. The mobile platform is a wheeled tricycle with a steering mechanism that has been attached to the front wheel, as shown in the figure below.



The front wheel drives the platform with the following controls

- The wheel is driven by a traction motor and is tracked by an encoder at 512 ticks per revolution.
- The wheel can also steer between $(-\pi/2, \pi/2)$ and is tracked by an absolute encoder.

The system parameters are:

- Front wheel radius = 0.2m
- Back wheels radius = 0.2m
- Distance from front wheel to back axis (r) = 1m
- Distance between rear wheels (d) = 0.75m

The vehicle is equipped with a gyroscope that can track the angular velocity of the 2D platform.

2 API call:

```
pose_2D estimate(float current_time,float steering_angle,int encoder_ticks,float angular_velocity);
```

returns the estimated pose: an object of class pose_2D(defined in “pose_estimator.cpp”)

```
class pose_2D {
public:
float x;
float y;
float heading;
pose_2D(float x0, float y0, float heading0);
void update(float x0, float y0, float heading0);};
```

Refer to readme.txt for file layout, testing and modifications.

2.1 Parameters:

- float current_time - Time of reading of the input data
- float steering_angle - Steering wheel angle
- int encoder_ticks - Number of ticks from the traction motor encoder
- float angular_velocity - Reading from the gyroscope measuring the rotation velocity of the platform around the Z axis

3 Features:

- Clean interface: The function only requires the current input and measurement information to estimate the current state. Information about past states are stored and handled internally.
- Scalability: The pose estimation function is implemented as generic as possible with room for scaling and adapting to system changes. System dependent parts of the algorithm have been moduled as separate functions to allow ease of modification.
- Computational efficiency: With the use of Boost libraries, the computational load of the estimation algorithm has been minimized. The effect is more profound if the function is modified to handle higher order systems

4 Methodology

The pose estimation is done using a computationally stable version of the Effective Kalman Filter. In the given system,

- The Wheel encoders (encoder_ticks and steering_angle) have been assigned as the input variables
- The gyroscope data (angular_velocity) has been assigned as the measurement data.

The kalman filter fuses the information from the two noisy data to estimate a more accurate pose of the system.

4.1 Time update equations:

The first step in the algorithm is to execute the a priori time update equations.

$$\hat{X}_k^- = f(X_{k-1}, u_k) \tag{4.1}$$

where, x_{k-1} is the pose of the previous state and u_k is the control input.

The given three-wheeled system is assumed to follow a circular path when the steering angle is fixed and non-zero, or move in a straight line when the steering angle is zero. Accordingly, the state transition can be modelled as [See appendix1 for derivation]:

When $\alpha \neq 0$,

$$\begin{bmatrix} \hat{x}_k^- \\ \hat{y}_k^- \\ \hat{\theta}_k^- \end{bmatrix} = \begin{bmatrix} x_{k-1} + r/\tan\alpha * (\sin(\theta_{k-1} + \sin(\alpha)\frac{2\pi n*rad}{512*r}) - \sin\theta_{k-1}) \\ y_{k-1} + r/\tan\alpha * (-\cos(\theta_{k-1} + \sin(\alpha)\frac{2\pi n*rad}{512*r}) + \cos\theta_{k-1}) \\ \theta_{k-1} + \sin\alpha\frac{2\pi n*rad}{512*r} \end{bmatrix} \tag{4.2}$$

When $\alpha = 0$,

$$\begin{bmatrix} \hat{x}_k^- \\ \hat{y}_k^- \\ \hat{\theta}_k^- \end{bmatrix} = \begin{bmatrix} x_{k-1} + \frac{2\pi n*rad}{512} * \cos\theta_{k-1} \\ y_{k-1} + \frac{2\pi n*rad}{512} * \sin\theta_{k-1} \\ \theta_{k-1} \end{bmatrix} \tag{4.3}$$

where, $\hat{x}_k, \hat{y}_k$ and $\hat{\theta}_k$are the predicted x, y and θ respectively, r is the distance between the front and back wheels, rad is the radius of the front wheel, α is the steering angle and n is the number of encoder ticks.

The measurement function 'h' for calculating angular velocity is $\hat{z_k} = \frac{\hat{\theta}_k^- - \theta_{k-1}}{t_k - t_{k-1}}$;

The state error covariance matrix is projected forward by the equation

$$P_k^- = A_k P_{k-1} A_k^T + B_k \Gamma_{k-1} B_k^T + Q_{k-1} \tag{4.4}$$

Where,

- A_K is the jacobian of f w.r.t X: $A_{[i,j]} = \frac{\partial f_{[i]}}{\partial x_{[j]}}(\hat{X}_k^-, u_k)$

When $\alpha \neq 0$,

$$A_k = \begin{bmatrix} 1 & 0 & r/\tan\alpha * (\cos(\theta_{k-1} + \sin(\alpha)\frac{2\pi n*rad}{512*r}) - \cos\theta_{k-1}) \\ 0 & 1 & r/\tan\alpha * (\sin(\theta_{k-1} + \sin(\alpha)\frac{2\pi n*rad}{512*r}) - \sin\theta_{k-1}) \\ 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

When $\alpha = 0$,

$$A_k = \begin{bmatrix} 1 & 0 & \frac{-2\pi n*rad}{512} * \sin\theta_{k-1} \\ 0 & 1 & \frac{2\pi n*rad}{512} * \cos\theta_{k-1} \\ 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

- B_K is the jacobian of f w.r.t u: $A_{[i,j]} = \frac{\partial f_{[i]}}{\partial u_{[j]}}(\hat{X}_k^-, u_k)$

When $\alpha \neq 0$,

$$B_k = \begin{bmatrix} \frac{\sin\alpha}{\tan\alpha}\frac{2\pi rad}{512}\cos(\theta_{k-1} + \sin(\alpha)\frac{2\pi n*rad}{512*r}) & \frac{\cos\alpha}{\tan\alpha}\frac{2\pi n*rad}{512}\cos(\theta_{k-1} + \sin(\alpha)\frac{2\pi n*rad}{512*r}) \\ \frac{\sin\alpha}{\tan\alpha}\frac{2\pi rad}{512}\sin(\theta_{k-1} + \sin(\alpha)\frac{2\pi n*rad}{512*r}) & \frac{\sin\alpha}{\tan\alpha}\frac{2\pi n*rad}{512}\sin(\theta_{k-1} + \sin(\alpha)\frac{2\pi n*rad}{512*r}) \\ \sin(\alpha)\frac{2\pi rad}{512*r} & \cos(\alpha)\frac{2\pi n*rad}{512*r} \end{bmatrix} \tag{4.7}$$

When $\alpha = 0$,

$$B_k = \begin{bmatrix} \frac{2\pi rad}{512} * \cos\theta_{k-1} & 0 \\ \frac{2\pi n * rad}{512} * \sin\theta_{k-1} & 0 \\ 0 & \cos(\alpha)\frac{2\pi n * rad}{512 * r} \end{bmatrix} \tag{4.8}$$

- P_{k-1} is the state error covariance from the previous state estimation. P_0 is a (3x3) zero matrix since the starting position is fixed as per the problem statement. The subsequent P_k matrices are updated in the measurement update equations (Eq 4.15).
- Γ_{k-1} is the input error covariance from the previous state estimation. This covariance matrix is assumed to be time invariant. For illustration purposes, the following Γ_{k-1}has been used in section 6.

$$\Gamma_{k-1} = \Gamma_0 = \begin{bmatrix} 25 & 0 \\ 0 & 3.04 * 10^{-4} \end{bmatrix} \tag{4.9}$$

- Q_{k-1} is the system error covariance from the previous state estimation. This covariance matrix is assumed to be time invariant. For illustration purposes, the following Q_{k-1}has been used in section 6.

$$Q_{k-1} = Q_0 = \begin{bmatrix} 0.01 & 0 & 0 \\ 0 & 0.01 & 0 \\ 0 & 0 & 3.04 * 10^{-4} \end{bmatrix} \tag{4.10}$$

4.2 Measurement update equations

Once measurements z_k become available the Kalman gain matrix K_k is computed

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \tag{4.11}$$

Where,

- H_K is the jacobian of h w.r.t u: $A_{[i,j]} = \frac{\partial h_{[i]}}{\partial u_{[j]}}(\hat{X}_k^-, u_k)$

$$H_k = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{t_k - t_{k-1}} \end{bmatrix} \tag{4.12}$$

- R_K is the measurement error covariance matrix

$$R_k = \begin{bmatrix} 0.01 \end{bmatrix} \tag{4.13}$$

The pose of the system is updated based on the error between the gyroscope measurement and the predicted angular velocity measurement, scaled by the kalman gain.

$$x_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-)) \tag{4.14}$$

Finally, the state error covariance matrix can be updated using

$$P_k = (I - K_k H_k) P_k^-$$

In this algorithm, the computationally stable- Joseph form of the update equation has been used

$$P_k = (I - K_k H_k) P_k^- (I - K_k H_k)^T + K_k R_K K_k^T \tag{4.15}$$

5 Code breakdown:

The function is defined in the program file *"estimate.cpp"*. The matrices are defined using *boost::uBLAS*: a C++ template class library that provides BLAS level 1, 2, 3 functionality for dense, packed and sparse matrices.

5.1 Classes:

Other than boost class, the function uses three custom classes defined in their respective header files:

- class *pose_2D* from *pose_2D.h*: It serves as a template for storing the pose of the vehicle and also provides a constructor and an initializer.

```
class pose_2D {
public:
float x;
float y;
float heading;
pose_2D(float x0, float y0, float heading0);
void update(float x0, float y0, float heading0);};
```

- class *system_param* from *system_param.h*: This class provides the vehicle dimensions and a function *wheel_distance()* to calculate the distance travelled by the wheel, given the number of ticks of the encoder.

```
class system_param {
public:
const float pi;
const int ticks_PR;
const float wheel_rad;
const float r;
private:
float distance;

public: system_param() : pi(3.1416),ticks_PR(512),wheel_rad(0.2),r(1);
float wheel_distance(int n_ticks); };
```

- class *state_memory* from *state_memory.h*: This class allocates the static memory needed to store the previous state and its covariance matrices. *state_memory::measurement_function()* defines the measurement function $h(\hat{x}_k^-)$

```
class state_memory {
public:
static pose_2D prev_pose;
static int state;
static float prev_time;
static mat<float> prev_state_covariance;
static mat<float> prev_input_covariance;
static mat<float> prev_system_covariance;
static mat<float> prev_measurement_covariance;
private: mat<float> z;

public:
state_memory();
mat<float> measurement_function(mat<float> x_predicted, float current_time);};
```

5.2 Supporting functions:

The main function *estimate()* is supported by a few external functions that execute specific tasks in the algorithm.

- *initializing_ covariances()* from *"initializing_ covariances.cpp"* is a function that initializes the static error covariance matrices P_0, Q_0, Γ_0 and R_0 during the first function call. (Eq 4.4 and Eq 4.11)
- *jacobian_functions()* from *"jacobian_functions.cpp"* is a function that updates the jacobian functions A_k, B_k and H_k during the time update step of every call using current and previous state variables. (Eq 4.4 and Eq 4.11)
- *matrix_ inversion()* from *"matrix_ inversion.cpp"* performs the matrix inversion required by Eq 4.11 using lu decomposition.

5.3 Function estimate()

A step-by-step explaination of the pose estimation function is presented below.

- Create instances of required classes and evaluate temporary variables

```
system_param  inst;
state_memory  instance;

float  wheel_dist  =  inst.wheel_distance(encoder_ticks);
float  current_heading  =  instance.prev_pose.heading
                           +sin(steering_angle)*wheel_dist/inst.r;
```

- If the function is called during the same timestamp twice, exit function call by returing the previous pose.

```
if(current_time==instance.prev_time)  return  instance.prev_pose;
```

- Begin the time update step by predicting the current pose(*x_predicted(3,1)*) using the control inputs *steering angle* and *encoder ticks.* (Eq 4.2 and Eq 4.3)

```
if(steering_angle!=0){
x_predicted(0,0)=instance.prev_pose.x + inst.r/tan(steering_angle)*(sin(current_heading)
                                        -sin(instance.prev_pose.heading));

x_predicted(1,0)=instance.prev_pose.y + inst.r/tan(steering_angle)*(-cos(current_heading)
                                        +cos(instance.prev_pose.heading));

x_predicted(2,0)=instance.prev_pose.heading+sin(steering_angle)*wheel_dist/inst.r;}

else{
x_predicted(0,0)=instance.prev_pose.x + wheel_dist*cos(instance.prev_pose.heading);
x_predicted(1,0)=instance.prev_pose.y + wheel_dist*sin(instance.prev_pose.heading);
x_predicted(2,0)=instance.prev_pose.heading;}
```

- Initialize covariance matrices $P_0, Q_0, \Gamma_0$and R_0during the first function call using function *initializing_ covariances()*

```
if(instance.state==0){
instance.prev_state_covariance=initializing_covariances(STATE_COVARIANCE);
instance.prev_system_covariance=initializing_covariances(SYSTEM_COVARIANCE);
instance.prev_input_covariance=initializing_covariances(INPUT_COVARIANCE);
instance.prev_measurement_covariance=initializing_covariances(MEASUREMENT_COVARIANCE);}
```

- Update jacobian functions $A_k, B_k$and $H_k$during every function call using *jacobian_functions()*

```
system_A = jacobian_functions(SYSTEM_A,....);
input_B = jacobian_functions(INPUT_B,....);
measurement_H = jacobian_functions(MEASUREMENT_H,....);
```

- Evaluate projected state covariance P_k^-using Eq 4.4

```
projected_state_covariance =

prod(system_A,matrix<float>(prod(instance.prev_state_covariance,trans(system_A))))
+prod(input_B,matrix<float>(prod(instance.prev_input_covariance,trans(input_B))))
+instance.prev_system_covariance;
```

- In the measurement update step, calculate the kalman gain vector using Eq 4.11

```
kalman_temp1 = prod(projected_state_covariance,trans(measurement_H));

kalman_temp2 = matrix_inversion(matrix<float>(prod(measurement_H,
matrix<float>(prod(instance.prev_state_covariance,trans(measurement_H))))
+instance.prev_measurement_covariance));

kalman_gain = prod(kalman_temp1,kalman_temp2);
```

- Update the current pose using the angular velocity measurement and the kalman gain vector(Eq 4.14)

```
x_updated = x_predicted + prod(kalman_gain,matrix<float>
(measurement_z-instance.measurement_function(x_predicted,current_time)));
```

- Update the state covariance matrix using Eq 4.15

```
covariance_temp = identity_matrix<float> (3,3) - prod(kalman_gain,measurement_H);

updated_state_covariance =
prod(covariance_temp,mat<float>(prod(instance.prev_state_covariance,trans(covariance_temp)
+prod(kalman_gain,mat<float>(prod(instance.prev_measurement_covariance,trans(kalman_gain))
```

- Update the state memory with the current state values for the next function call and return the current pose.

```
instance.state++;
instance.prev_pose.update(x_updated(0,0),x_updated(1,0),x_updated(2,0));
instance.prev_state_covariance = updated_state_covariance;
instance.prev_time = current_time;

return instance.prev_pose;
```

6 Testing the code:

In order to test function *estimate()*, two additional programs have been attached.

- *genrating_ data_ file.cpp* simulates the states of the three wheeled mobile platform system and prints the data to a file(“*input.dat”* by default). Each state is represented by a row of numbers: *current_ time,steering_ angle, encoder_ ticks* and *angular_ velocity* followed by the true *x,y and heading* of the system. A variable error of upto 5% is also added to these variables to simulate a real system.

```
0.1       0.08727  104       0.2118    0.2469    0.002537  0.02054
0.2       1.033    188       1.938     0.3367    0.01428  0.212
0.3       0.2382   245       0.316     0.4956    0.04657  0.2407
0.4       0.8613   295       0.8867    0.5703    0.06828  0.335
```

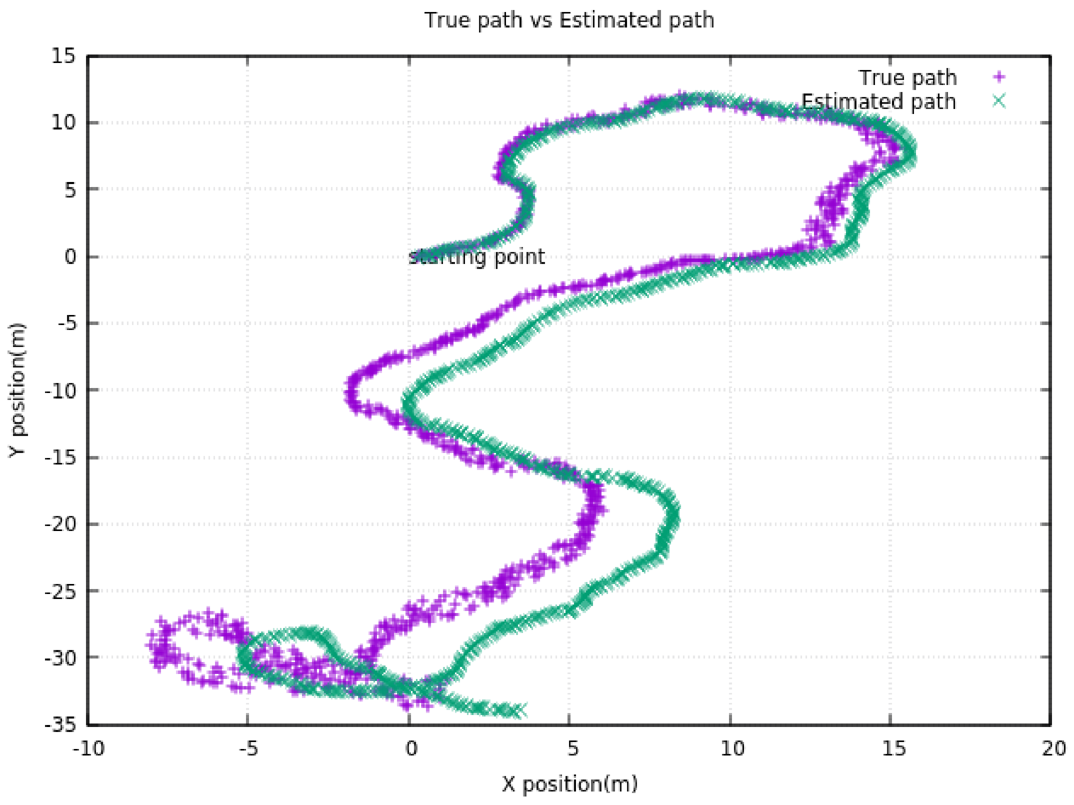
- *test_ estimate.cpp* is a parent code that reads the data file generated above and feeds the input data *current_ time,steering_ angle, encoder_ ticks* and *angular_ velocity* to the function call *estimate()*. It outputs the current time and estimated pose at each time step to a file(“*output.dat”* by default)

```
0.1       0.251725            0.00277204          0.0187795
0.2       0.368572            0.0161129           0.217194
0.3       0.500265            0.0472129           0.247898
0.4       0.575473            0.0693583           0.333781
```

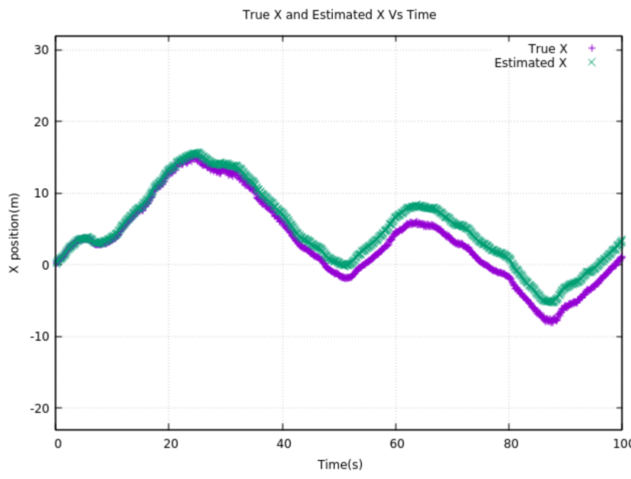
A sample run of *generating_ data_ file.cpp* generated an “*input.dat”* file with 1000 states at an interval of *0.1s* between the states. This input data was processed by the *test_ estimate.cpp* program and the estimated poses were stored in an “*output.dat*” file.

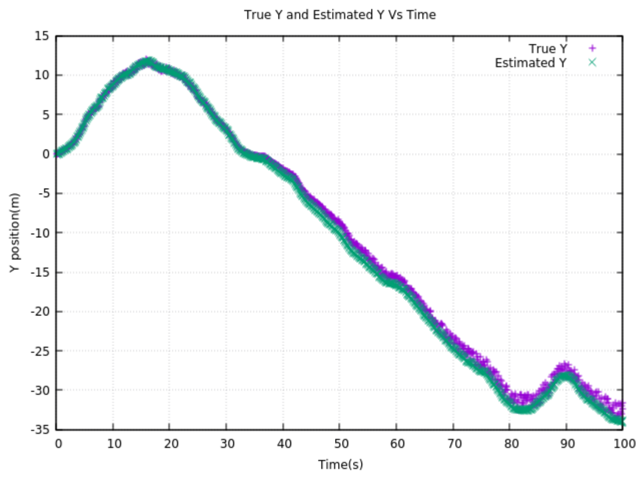
6.1 Plots:

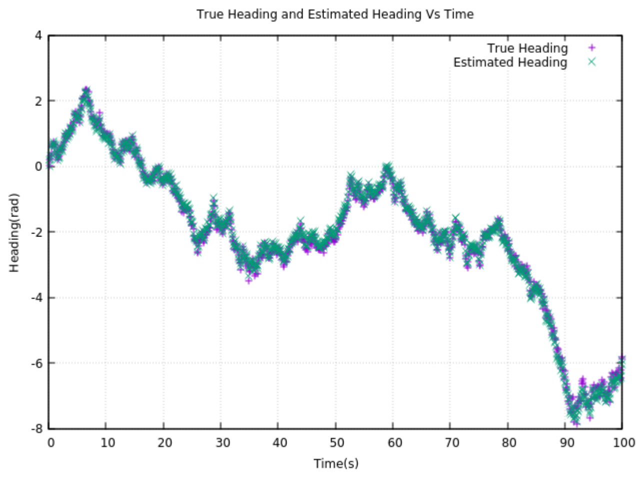
The following plots were made using *gnuplot.* The estimated pose tracks the true path reasonably well, as seen from this plot.



Each element of the estimated pose is plot against time along with the true pose counterparts.







The feedback obtained from using only the gyroscope measurement has its limitations. It is evident from the plots obtained, that the heading element of the pose is fairly accurate. However, x and y coordinates cannot be both guaranteed to be estimated accurately.

The RMSE for each variable was calculated to be

- RMSE for x = 1.8393
- RMSE for y = 0.8202
- RMSE for heading = 0.0892

7 References:

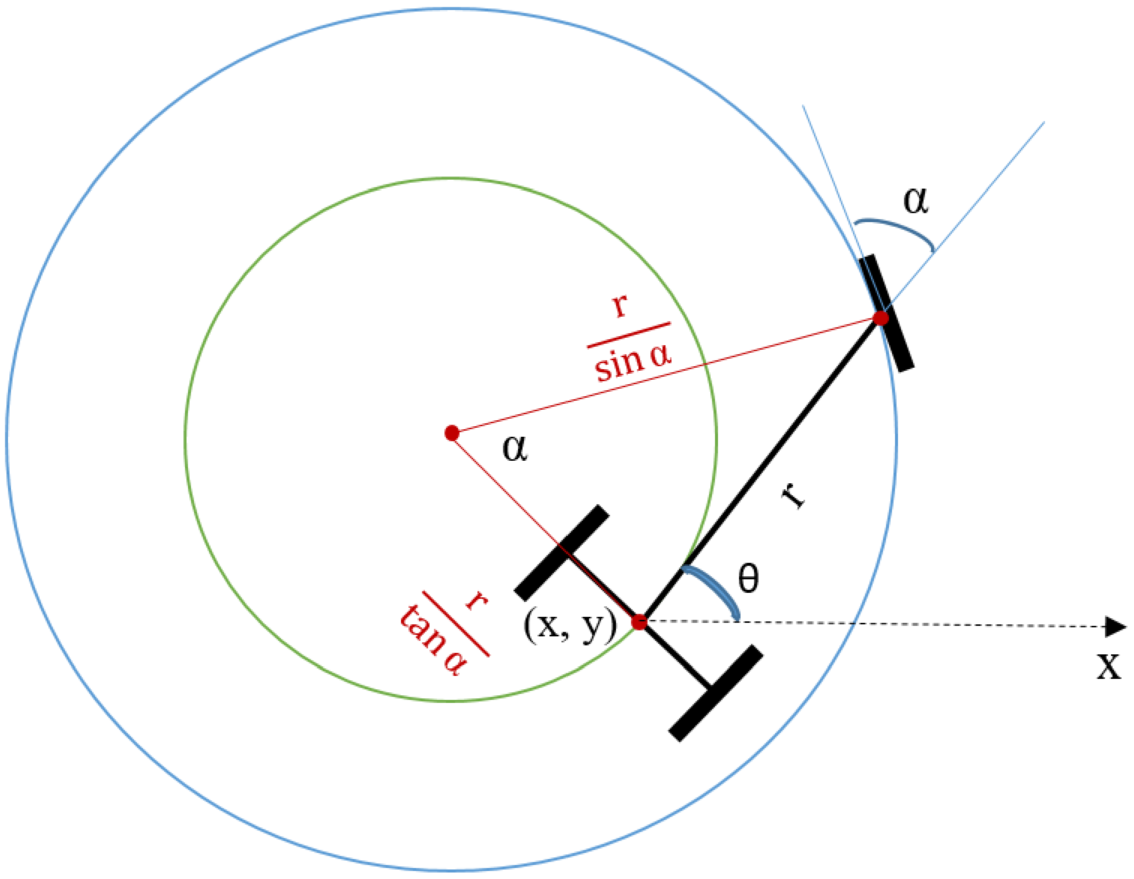
1. Kiriy, E., & Buehler, M. (2002). Three-state extended kalman filter for mobile robot localization. McGill University., Montreal, Canada, Tech. Rep. TR-CIM, 5, 23.

2. Brown, R. G., & Hwang, P. Y. (1992). Introduction to random signals and applied Kalman filtering (Vol. 3). New York: Wiley.

8 APPENDIX

8.1 Derivation of state transition function $f()$

The three wheeled platform moves in a circular path when the steering angle is fixed. The front wheel and the centre of the back wheels(x,y) move in concentric circles of different radii. The centre of these circles is the turning point of the vehicle. The radius of the inner circle is $r/\tan\alpha$ and outer circle is $r/\sin\alpha$.



If the front wheel moves a distance Δs_f at an angle α, the vehicle turns an angle

$$\Delta\theta = \Delta s_f \frac{\sin\alpha}{r} \tag{8.1}$$

Correspondingly, distance moved by the point (x, y): $\Delta s_b = \Delta\theta\frac{r}{\tan\alpha} \Rightarrow \Delta s_b = \Delta s_f \cos\alpha$

If Δs_f is very small, i.e. $\Delta s_f \to ds_f$ and consequently $\Delta s_b \to ds_b$, the point (x,y) can be approximated to move in a straight line at an angle θ for the short distance ds_b.

The path of the point can be modelled as:

$$dx = ds_b \cos\theta = ds_f \cos\alpha\cos\theta \tag{8.2}$$

$$dy = ds_b \sin\theta = ds_f \cos\alpha\sin\theta \tag{8.3}$$

$$d\theta = ds_f \frac{\sin\alpha}{r} \tag{8.4}$$

For fixed steering angle α, the state variables x, y and θ can be derived as the integral over dx, dy and $d\theta$ respectively.

Integrating Eq 6.2: $\int dx = \int ds_f \cos\alpha \cos\theta = \int \frac{rd\theta}{\sin\alpha} \cos\alpha \cos\theta$

$$x_k - x_{k-1} = \frac{r}{\tan\alpha}(\sin\theta_k - \sin\theta_{k-1}) \tag{8.5}$$

Integrating Eq 6.3: $\int dy = \int ds_f \cos\alpha \sin\theta = \int \frac{rd\theta}{\sin\alpha} \cos\alpha \sin\theta$

$$y_k - y_{k-1} = \frac{r}{\tan\alpha}(-\cos\theta_k + \cos\theta_{k-1}) \tag{8.6}$$

Integrating Eq 6.4: $\int d\theta = \int ds_f \frac{\sin\alpha}{r}$

$$\theta_k - \theta_{k-1} = s_f \frac{\sin\alpha}{r} \tag{8.7}$$

Where, $s_f = \frac{n}{512} 2\pi rad$, n is the number of encoder ticks, rad is the radius of the front wheel.

Rewriting Eq 6.5, 6.6 and 6.7 in the form of Eq 4.1

$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} + \frac{2\pi n * rad}{512} * \cos\theta_{k-1} \\ y_{k-1} + \frac{2\pi n * rad}{512} * \sin\theta_{k-1} \\ \theta_{k-1} \end{bmatrix}$$