통신학회 단기강좌:
자율주행 핵심기술 SLAM

실습 2. Kalman Filter의 응용

민세웅, 김동현, 강인성
Hanyang Univ.
(sewoong@hanyang.ac.kr
kissw@hanyang.ac.kr
kangis@hanyang.ac.kr)

HYU 한양대학교 HANYANG UNIVERSITY

ICSL 한양대학교지능통신연구실 Intelligent Communication Systems Lab

IMU 센서를 이용한 수평 자세 측정

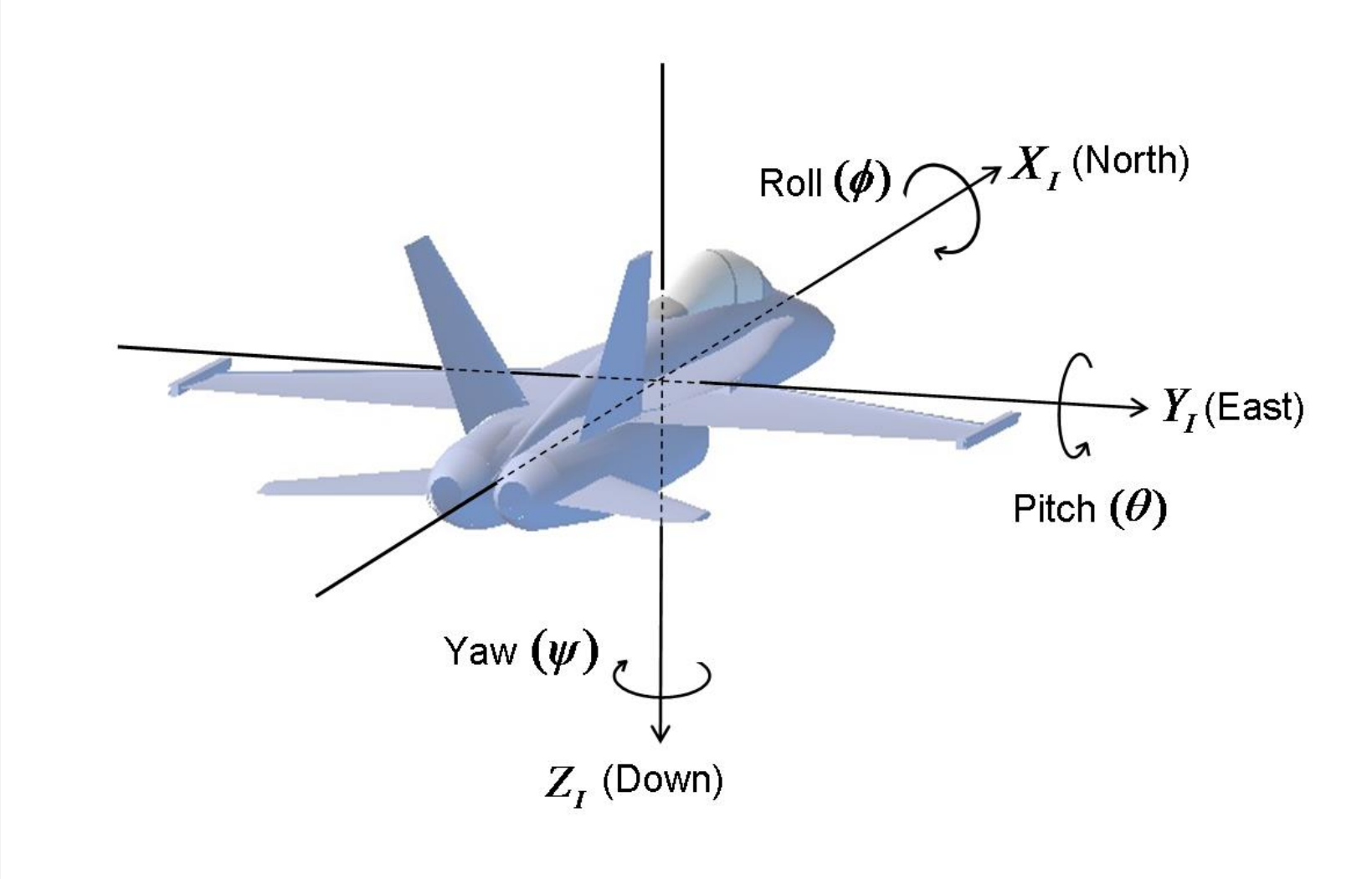
- IMU (Inertial Measurement Units)
  - 3축 가속도 측정(Acceleometer)
  - 3축 각속도 측정(Gyroscope)

+

  - 3축 자기계 측정(Magnetometer)
  - 1축 기압계 측정(Barometer)

Euler Angle

- Roll (Phi)
- Pitch (Theta)
- Yaw (Psi)



자이로를 이용한 자세 결정

- 자이로로 측정한 각속도를 통해서 오일러 각을 구할 수 없음
- 자이로는 비행 기체의 각속도 (p, q, r)을 제공
- 동역학에서 오일러각과 각속도의 관계는 알려져 있음

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\varphi} \end{bmatrix} = \begin{vmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\cos\theta & \cos\phi\cos\theta \end{vmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

- 위 수식으로 구할 경우 phi, theta, psi에 대한 미분값이므로 적분시 오차가 누적 됨

EulerGyro 코드

- https://github.com/sewoongmin/KalmanFilter_ROS_Exam.git

- Import lib
  - Rospy – 파이썬을 사용하여 ROS 패치지 구성
  - Sensor_msgs.msg import IMU – Sensor_msgs에 있는 IMU 데이터 타입
  - Geometry_msgs.msg import Vector3 – phi, theta, psi의 구한 값 전송용
  - Numpy – 파이썬에서 행렬 연산용
  - Math – 삼각함수 라이브러리용

```
import rospy
from sensor_msgs.msg import Imu
from geometry_msgs.msg import Vector3
import numpy as np
import math
```

EulerGyro 코드

- 변수 초기화

```
 9    dt = 0.02
10    gyro = np.zeros((3,1))
11    prev_x = np.zeros((3,1))
12    euler = Vector3()
13    degree = Vector3()
```

- 각속도 값 받기

```
def CallBack(msg) :
    gyro[0][0] = msg.angular_velocity.x
    gyro[1][0] = msg.angular_velocity.y
    gyro[2][0] = msg.angular_velocity.z
```

EulerGyro 코드

- 추정값으로부터 삼각함수값 계산

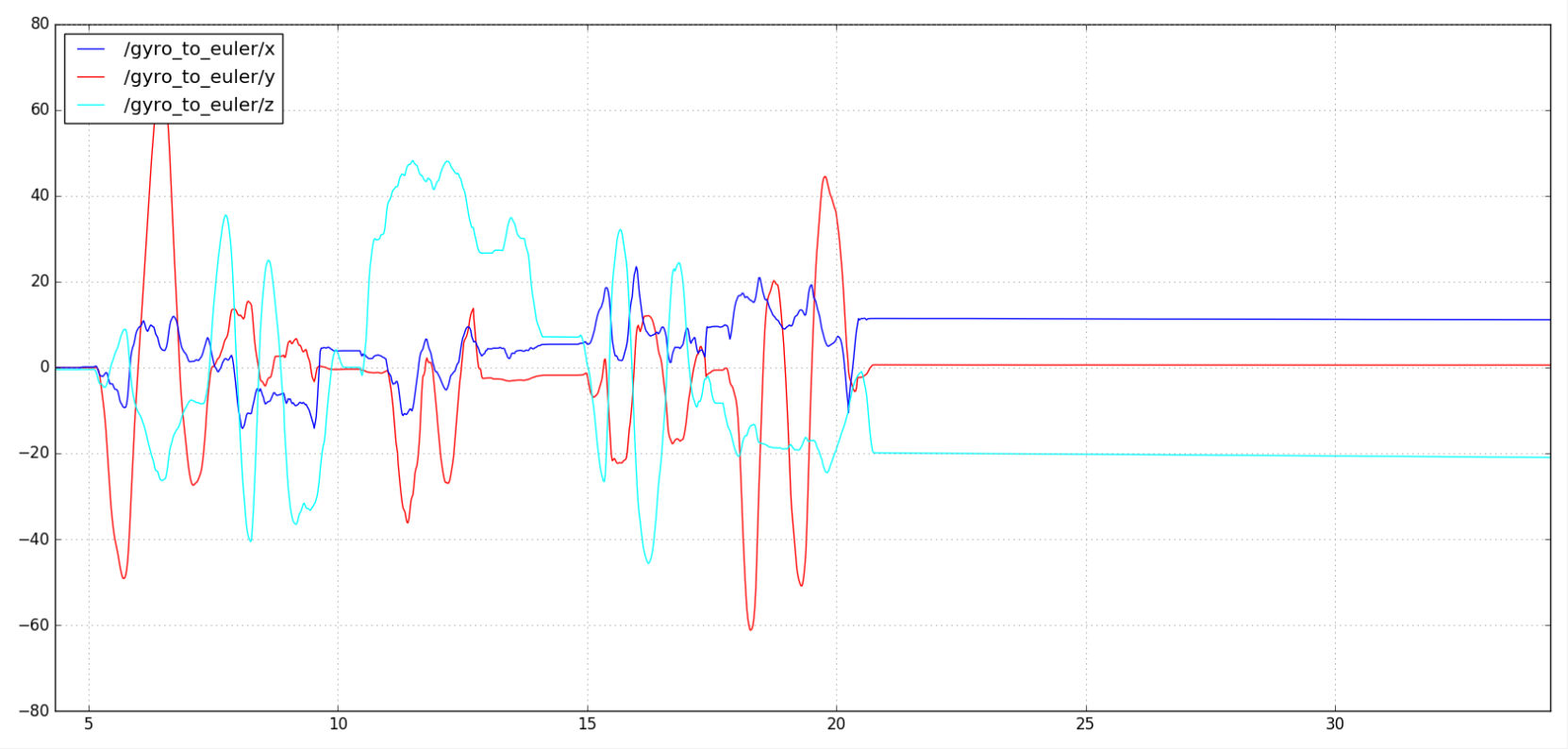
```
20      sinPhi = math.sin(prev_x[0][0])
21      cosPhi = math.cos(prev_x[0][0])
22      cosTheta = math.cos(prev_x[1][0])
23      tanTheta = math.tan(prev_x[1][0])
```

- 행렬식을 풀어서 이산시간에 대한 적분계산

```
25      phi = prev_x[0][0] + dt*( gyro[0][0] + gyro[1][0]*sinPhi*tanTheta + gyro[2][0]*cosPhi*tanTheta)
26      theta = prev_x[1][0] + dt*( gyro[1][0]*cosPhi - gyro[2][0]*sinPhi)
27      psi = prev_x[2][0] + dt*(gyro[1][0]*sinPhi/cosTheta + gyro[2][0]*cosPhi/cosTheta)
```

EulerGyro 결과

- x(phi), y(theta), z(psi) 전부 계산 가능
- 실제 결과 값의 경향성을 거의 완벽히 따라감
- 적분에 의한 편향(bias)이 시간이 지남에 따라 커짐



가속도를 이용한 자세 결정

- 역학적으로 이동속도와 회전 각속도 그리고 중력가속도에 의해 측정되는 가속도 (f_x, f_y, f_z) 는 아래 수식과 같음

$$\begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} = \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} + \begin{vmatrix} 0 & w & -v \\ -w & 0 & u \\ v & -u & 0 \end{vmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} + g \begin{bmatrix} \sin\theta \\ -\cos\theta \sin\phi \\ -\cos\theta \cos\phi \end{bmatrix}$$

- 여기서 u, v, w는 이동속도를 의미하고, p, q, r은 회전 각속도를 의미함
- 우변의 마지막항을 보면 롤 각인 phi와 피치각인 theta가 나옴
- 위 식에서 IMU를 통해 (f_x, f_y, f_z) 와 p, q, r 을 알 수 있음

가속도를 이용한 자세 결정

- 이때 알 수 없는 변수는 이동속도 (u, v, w)와 이동 가속도 $(\dot{u}, \dot{v}, \dot{w})$ 임
- 이 값들은 일반적으로 고가의 항법센서가 아니면 측정 불가
- 시스템이 정지해 있을 경우

$$\dot{u} = \dot{v} = \dot{w} = 0$$
$$u = v = w = 0$$

- 일정한 속도로 직진하는 경우

$$\dot{u} = \dot{v} = \dot{w} = 0$$
$$\mathrm{p} = q = r = 0$$

- 두 경우 모두 우변의 첫번째 항과 두번째 항이 0이 됨

가속도를 이용한 자세 결정

- 따라서 식은 다은과 같이 간단한 형태가 됨

$$\begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} = g \begin{bmatrix} \sin\ \theta \\ -\cos\theta \sin\phi \\ -\cos\theta \cos\phi \end{bmatrix}$$

- 위 식에 따라 phi와 theta를 아래와 같이 유도 가능

$$\phi = \sin^{-1}\left(\frac{-f_y}{g\cos\theta}\right)$$

$$\theta = \sin^{-1}\left(\frac{f_x}{g}\right)$$

- 움직이는 속도가 충분히 느리거나 속도의 크기와 방향이 빠르게 변하지 않는 경우에 위 식을 통해 수평 자세를 구할 수 있음
- 이때 Yaw각인 psi는 알 수 없음
- 빠른 속도로 회전하거나 속도 변화가 심하면 오차가 커질 수 있음

EulerAccel 코드

- Import lib는 EulerGyro와 동일
- 변수 초기화 및 초기값 설정

```
 9    g = 9.8
10    prev_x = np.zeros((3,1))
11    euler = Vector3()
12    degree = Vector3()
```

- 앞의 식에 의거하여 가속도 값을 통해 phi와 theta 계산

```
def CallBack(msg) :
    theta = math.asin(msg.linear_acceleration.x / g)
    phi = math.asin( -msg.linear_acceleration.y / (g*math.cos(theta)))

    euler.x = phi
    euler.y = theta
    euler.z = 0
```

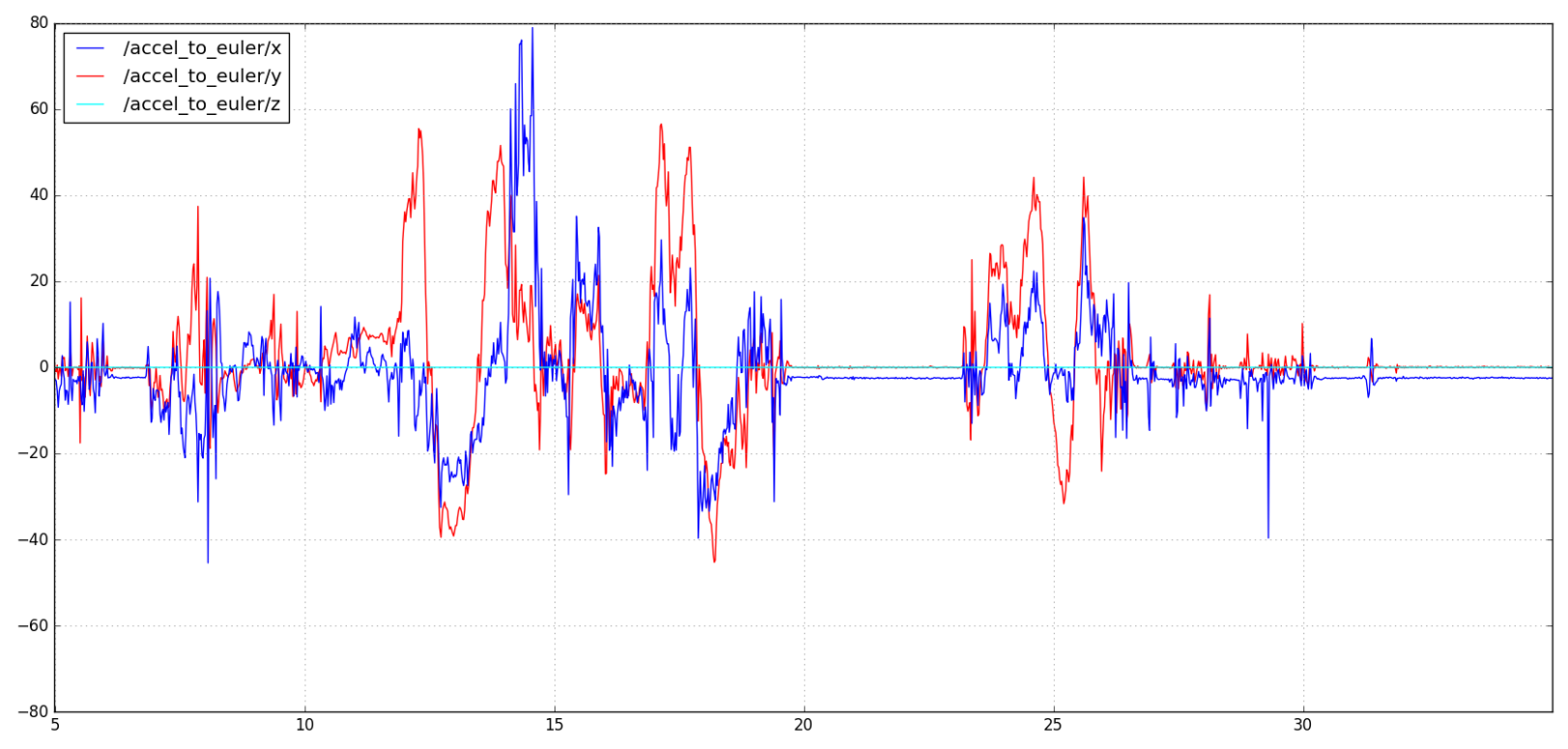
EulerAccel 코드

- “accel_to_euler”의 토픽이름으로 초에 50번씩 메시지 발행

```
22  rospy.init_node("euler_accel")
23  sub = rospy.Subscriber("mavros/imu/data_raw", Imu, CallBack)
24  pub = rospy.Publisher("accel_to_euler", Vector3, queue_size = 10)
25  rate = rospy.Rate(50)
26  while not rospy.is_shutdown() :
27      degree.x = euler.x * 180/math.pi
28      degree.y = euler.y * 180/math.pi
29      degree.z = euler.z * 180/math.pi
30      pub.publish(degree)
31      rate.sleep()
32  rospy.spin()
```

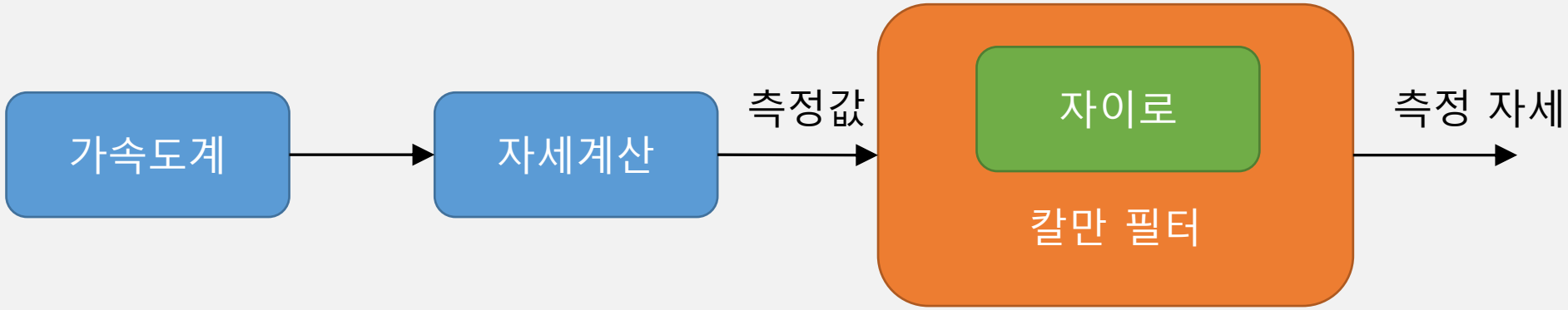
EulerAccel 결과

- Phi와 theta만 계산 가능
- 자이로 결과와 달리 경향성은 따라가나 부정확하고 노이지(noisy) 함
- 다만 시간에 따른 편향(bias) 가 없음



칼만 필터로 센서융합하여 자세 결정

- 자이로와 가속도 센서를 통한 자세 추정은 둘 다 단독으로 사용하기에는 단점이 있음 (각각 편향, 부정확함과 noisy)
- 따라서 아래와 같은 구조로 칼만필터를 적용하여 자세를 추정
- 자이로는 편향값으로 인해 보정값으로 사용 불가함으로 자이로와 가속도의 위치는 변경 할 수 없음



칼만 필터로 센서융합하여 자세 결정

- 자이로를 통해 자세를 구하는 식은 아래와 같음
- 여기서 상태변화 식인 $x_{k+1} = Ax_k + w_k$ 와 같이 변형이 불가능
- 따라서 선형 칼만 필터로는 구현 불가능

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\varphi} \end{bmatrix} = \begin{vmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\cos\theta & \cos\phi\cos\theta \end{vmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

칼만 필터로 센서융합하여 자세 결정

- 상태 수를 phi, theta, psi가 아닌 쿼터니언(Quatemion)으로 변경하면 선형 칼만 필터 구현 가능

$$\begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{bmatrix} = \frac{1}{2} \begin{vmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -q & 0 \end{vmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix}$$

칼만필터 코드

- Import lib – euler를 quaternion으로 변경해주는 ros lib 추가

```
3   import rospy
4   from sensor_msgs.msg import Imu
5   from geometry_msgs.msg import Vector3
6   from tf.transformations import quaternion_from_euler
7   import numpy as np
8   import numpy.linalg as lin
9   import math
```

- 값 초기화

```
def __init__(self) :
    self._imu_sub = rospy.Subscriber("mavros/imu/data_raw", Imu, self.CallBack)
    self._euler_sub = rospy.Subscriber("accel_to_euler", Vector3, self.EulerCallBack)
    self._kalman_pub = rospy.Publisher("kalman", Vector3, queue_size = 10)
    self._dt = 0.02
    self._H = np.eye(4)
    self._Q = 0.0001*np.eye(4)
    self._R = 10*np.eye(4)
    self._P = np.eye(4)
    self._x = np.array([[1],[0],[0],[0]])
    self._A = np.zeros((4,4))
    self._K = np.zeros((4,4))
    self._xp = np.zeros((4,1))
    self._Pp = np.zeros((4,4))
    self._z = np.zeros((4,1))
    self._degree = Vector3()
    self._euler = Vector3()
```

칼만필터 코드

- System Mode 행렬인 A 구하기

```
64      def Create_A(self, p, q, r) :
65          A = np.zeros((4,4))
66          A[0][1] = -p
67          A[0][2] = -q
68          A[0][3] = -r
69          A[1][0] = p
70          A[1][2] = r
71          A[1][3] = -q
72          A[2][0] = q
73          A[2][1] = -r
74          A[2][3] = p
75          A[3][0] = r
76          A[3][1] = q
77          A[3][2] = -p
78          I = np.eye(4)
79          self._A = I+self._dt/2*A
```

칼만필터 코드

- 값 예측

```
55      def Prediction(self) :
56          self._xp = self._A.dot(self._x)
57          self._Pp = self._A.dot(self._P.dot(self._A.T)) + self._Q
```

- 칼만 게인 계산

```
59      def GetKalmanGain(self) :
60          temp = self._H.dot(self._Pp.dot(self._H.T)) + self._R
61          inverse_temp = lin.inv(temp)
62          self._K = self._Pp.dot(self._H.T.dot(inverse_temp))
```

칼만필터 코드

- 가속도를 통해 들어온 측정값을 쿼터니언으로 변경

```
49      def EulerCallBack(self, msg) :
50          phi = msg.x*math.pi/180
51          theta = msg.y*math.pi/180
52          psi = msg.z*math.pi/180
53          self._z = quaternion_from_euler(phi, theta, 0)
```

- 추정값 계산 및 오차공분산 계산

```
36          self._x = self._xp + self._K.dot(self._z - self._H.dot(self._xp))
37          self._P = self._Pp - self._K.dot(self._H.dot(self._Pp))
38
39          self._euler.x = math.atan2( 2*(self._x[2][0]*self._x[3][0] + self._x[0][0]*self._x[1][0]), 1 - 2*(self._x[1][0]**2 + self._x[2][0]**2))
40          self._euler.y = -math.asin( 2*(self._x[1][0]*self._x[3][0] - self._x[0][0]*self._x[2][0]))
41          self._euler.z = math.atan2( 2*(self._x[1][0]*self._x[2][0] + self._x[0][0]*self._x[3][0]), 1 - 2*(self._x[2][0]**2 + self._x[3][0]**2))
42
43          self._degree.x = self._euler.x * 180/math.pi
44          self._degree.y = self._euler.y * 180/math.pi
45          self._degree.z = self._euler.z * 180/math.pi
46          print(self._degree)
47          self._kalman_pub.publish(self._degree)
```

확장 칼만필터를 통한 자세추정 코드

- Import lib – 역함수를구하기 위한 numpy.linalg lib 추가

```
3   import rospy
4   from sensor_msgs.msg import Imu
5   from geometry_msgs.msg import Vector3
6   from tf.transformations import quaternion_from_euler
7   import numpy as np
8   import numpy.linalg as lin
9   import math
```

확장 칼만필터를 통한 자세추정 코드

- 변수 초기화

```
11  class ExtendedKalman(object) :
12
13      def __init__(self) :
14          self._imu_sub = rospy.Subscriber("mavros/imu/data_raw", Imu, self.CallBack)
15          self._euler_sub = rospy.Subscriber("accel_to_euler", Vector3, self.EulerCallBack)
16          self._kalman_pub = rospy.Publisher("ekf", Vector3, queue_size = 10)
17          self._dt = 0.02
18          self._H = np.zeros((2,3))
19          self._H[0][0] = 1
20          self._H[1][1] = 1
21          self._Q = 0.0001*np.eye(3)
22          self._Q[2][2] = 0.1
23          self._R = 6*np.eye(2)
24          self._P = np.eye(3)
25          self._x = np.zeros((3,1))
26          self._A = np.zeros((3,3))
27          self._K = np.zeros((3,2))
28          self._xp = np.zeros((3,1))
29          self._Pp = np.zeros((3,3))
30          self._z = np.zeros((2,1))
31          self._xhat = np.zeros((3,1))
32          self._degree = Vector3()
```

확장 칼만필터를 통한 자세추정 코드

- System 모델 함수인 A의 Jacobian 계산

```
62      def JacobA(self, xhat, rates) :
63          A = np.zeros((3, 3))
64          phi = xhat[0][0]
65          theta = xhat[1][0]
66
67          p = rates[0][0]
68          q = rates[1][0]
69          r = rates[2][0]
70
71          A[0][0] = q*math.cos(phi)*math.tan(theta) - r*math.sin(phi)*math.tan(theta)
72          A[0][1] = q*math.sin(phi)/(math.cos(theta)**2) + r*math.cos(phi)/(math.cos(theta)**2)
73          A[1][0] = -q*math.sin(phi) - r* math.cos(phi)
74          A[2][0] = q*math.cos(phi)/math.cos(theta) - r*math.sin(phi)/math.cos(theta)
75          A[2][1] = q*math.sin(phi)/math.cos(theta)*math.tan(theta) + r*math.cos(phi)/math.cos(theta)*math.tan(theta)
```

확장 칼만필터를 통한 자세추정 코드

- System 모델 함수 fx 구하기

```
79      def Fx(self, xhat, rates) :
80          phi = xhat[0][0]
81          theta = xhat[1][0]
82
83          p = rates[0][0]
84          q = rates[1][0]
85          r = rates[2][0]
86
87          xdot = np.zeros((3, 1))
88          xdot[0][0] = p+q*math.sin(phi)*math.tan(theta) + r*math.cos(phi)*math.tan(theta)
89          xdot[1][0] = q*math.cos(phi) - r*math.sin(phi)
90          xdot[2][0] = q*math.sin(phi)/math.cos(theta) + r* math.cos(phi)/math.cos(theta)
91
92          self._xp = xhat + self._dt*xdot
```

확장 칼만필터를 통한 자세추정 코드

- 오차 공분산 예측 및 회전 각속도 입력

```
34      def CallBack(self, msg) :
35          rates = np.zeros((3,1))
36          rates[0][0] = msg.angular_velocity.x
37          rates[1][0] = msg.angular_velocity.y
38          rates[2][0] = msg.angular_velocity.z
39
40          self.JacobA(self._xhat, rates)
41          self.Fx(self._xhat, rates)
42          self._Pp = self._A.dot(self._P.dot(self._A.T)) + self._Q
```

확장 칼만필터를 통한 자세추정 코드

- 가속도 센서를 통해 추정한 자세 값 입력

```
54      def EulerCallBack(self, msg) :
55          phi = msg.x*math.pi/180
56          theta = msg.y*math.pi/180
57          psi = msg.z*math.pi/180
58          self._xhat[0][0] = phi
59          self._xhat[1][0] = theta
60          self._xhat[2][0] = psi
```

- 칼만게인 계산 및 추정값과 오차공분산 계산

```
44          temp = self._H.dot(self._Pp.dot(self._H.T)) + self._R
45          inverse_temp = lin.inv(temp)
46          self._K = self._Pp.dot(self._H.T.dot(inverse_temp))
47          self._x = self._xp + self._K.dot(self._z - self._H.dot(self._xp))
48          self._p = self._Pp - self._K.dot(self._H.dot(self._Pp))
```

확장 칼만필터를 통해 자세추정 결과

- 센서 융합을 통해 가속도 센서의 장점인 편향(bias)이 없고 자이로센서의 장점으로 정확한 각 계산 가능