Building an Estimator

Project Setup and Details: https://github.com/udacity/FCND-Estimation-CPP

Step 1: Sensor noise

The code I used to find the Standard Deviation is:

```
import numpy as np
gps_x = np.loadtxt('/to/log/file,txt',delimiter=',',dtype='Float64',skiprows=1)[:,1]
acc_x = np.loadtxt('/to/log/file.txt',delimiter=',',dtype='Float64',skiprows=1)[:,1]
gps_x_std = np.std(gps_x)

print(f'GPS X Std: {gps_x_std}')
acc_x_std = np.std(acc_x)
print(f'Accelerometer X Std: {acc_x_std}')
```

This gave the values:
0.704
0.502

The Result in Scenario 6 is as follows:

```
### STUDENT SECTION

MeasuredStdDev_GPSPosXY =  0.704
MeasuredStdDev_AccelXY = 0.502
```

Result:

```
Simulation #3 (../config/06_SensorNoise.txt)
PASS: ABS(Quad.GPS.X-Quad.Pos.X) was less than MeasuredStdDev_GPSPosXY for 67% of the time
PASS: ABS(Quad.IMU.AX-0.000000) was less than MeasuredStdDev_AccelXY for 69% of the time
```

Step 2: Attitude Estimation

The improved integration scheme should result in an attitude estimator of < 0.1 rad for each of the Euler angles for a duration of at least 3 seconds during the simulation. The integration scheme should use quaternions to improve performance over the current simple integration scheme.

Integrating Euler rate into the estimated pitch and roll angle.

```
float predictedPitch = pitchEst + dtIMU * euler_dot.y;
float predictedRoll = rollEst + dtIMU * euler_dot.x;
ekfState(6) = ekfState(6) + dtIMU * euler_dot.z;  // yaw
// normalize yaw
if (ekfState(6) > F_PI) ekfState(6) -= 2.f*F_PI;
if (ekfState(6) < -F_PI) ekfState(6) += 2.f*F_PI;
```

Result:

```
Simulation #5 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seconds
```

Step 3: Prediction Step

State prediction based on the acceleration measurement by using Dead Reckoning method implementation:

```
/**
  x coordianate x= x + x_dot * dt
  y coordianate y= y + y_dot * dt
  z coordianate z= z + z_dot * dt
**/

predictedState(0) = curState(0) + curState(3) * dt;
predictedState(1) = curState(1) + curState(4) * dt;
predictedState(2) = curState(2) + curState(5) * dt;
//Convert the true acceleration
V3F acc_inertial = attitude.Rotate_BtoI(accel);
/**
  change in velocity along the x is a_x * dt
  change in velocity along the y is a_y * dt
  change in velocity along the z is a_z * dt by removing the gravity component
**/

predictedState(3) = curState(3) + acc_inertial.x * dt;
predictedState(4) = curState(4) + acc_inertial.y * dt;
predictedState(5) = curState(5) + acc_inertial.z * dt - CONST_GRAVITY * dt;
```

Partial derivative of the body-to-global rotation matrix in the function Calculation:

```
float theta = pitch;
float phi = roll ;
float psi = yaw ;
RbgPrime(0,0) = (- ( cos(theta) * sin(psi) ) );
RbgPrime(0,1) = (- ( sin(phi) * sin(theta) * sin(psi) ) - ( cos(phi) * cos(psi) ) );
RbgPrime(0,2) = (- ( cos(phi) * sin(theta) * sin(psi) ) + ( sin(phi) * cos(psi) ) );
RbgPrime(1,0) = ( cos(theta) * cos(psi) ) ;
RbgPrime(1,1) = ( sin(phi) * sin(theta) * cos(psi) ) - ( cos(phi) * sin(psi) );
RbgPrime(1,2) = ( cos(phi) * sin(theta) * cos(psi) ) + ( sin(phi) * sin(psi) );
RbgPrime(2,0) = 0;
RbgPrime(2,1) = 0;
RbgPrime(2,2) = 0;
```

Next, After Obtaining the Jacobian Matrix:

```
gPrime(0,3) = dt;
gPrime(1,4) = dt;
gPrime(2,5) = dt;

gPrime(3, 6) = (RbgPrime(0) * accel).sum() * dt;
gPrime(4, 6) = (RbgPrime(1) * accel).sum() * dt;
gPrime(5, 6) = (RbgPrime(2) * accel).sum() * dt;
ekfCov = gPrime * ekfCov * gPrime.transpose() + Q;
```

Step 4: Magnetometer Update

Implementation:

```
zFromX(0) = ekfState(6);
float diff = magYaw - ekfState(6);
if ( diff > F_PI ) {
  zFromX(0) += 2.f*F_PI;
} else if ( diff < -F_PI ) {
  zFromX(0) -= 2.f*F_PI;
}

hPrime(0, 6) = 1;
```

Result:

```
Simulation #16 (../config/10_MagUpdate.txt)
PASS: ABS(Quad.Est.E.Yaw) was less than 0.120000 for at least 10.000000 seconds
PASS: ABS(Quad.Est.E.Yaw-0.000000) was less than Quad.Est.S.Yaw for 68% of the time
```

Step 5: Closed Loop + GPS Update

Implementation:

```
  for ( int i = 0; i < 6; i++) {
  zFromX(i) = ekfState(i);
 }


for ( int i = 0; i < 6; i++) {
  hPrime(i,i) = 1;
}
```

Result:

```
Simulation #18 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds
```

Step 6: Adding Your Controller

This is accomplished by integrating our Controller from the Project 3 of the Nanodegree

Result:

```
Simulation #18 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds
```