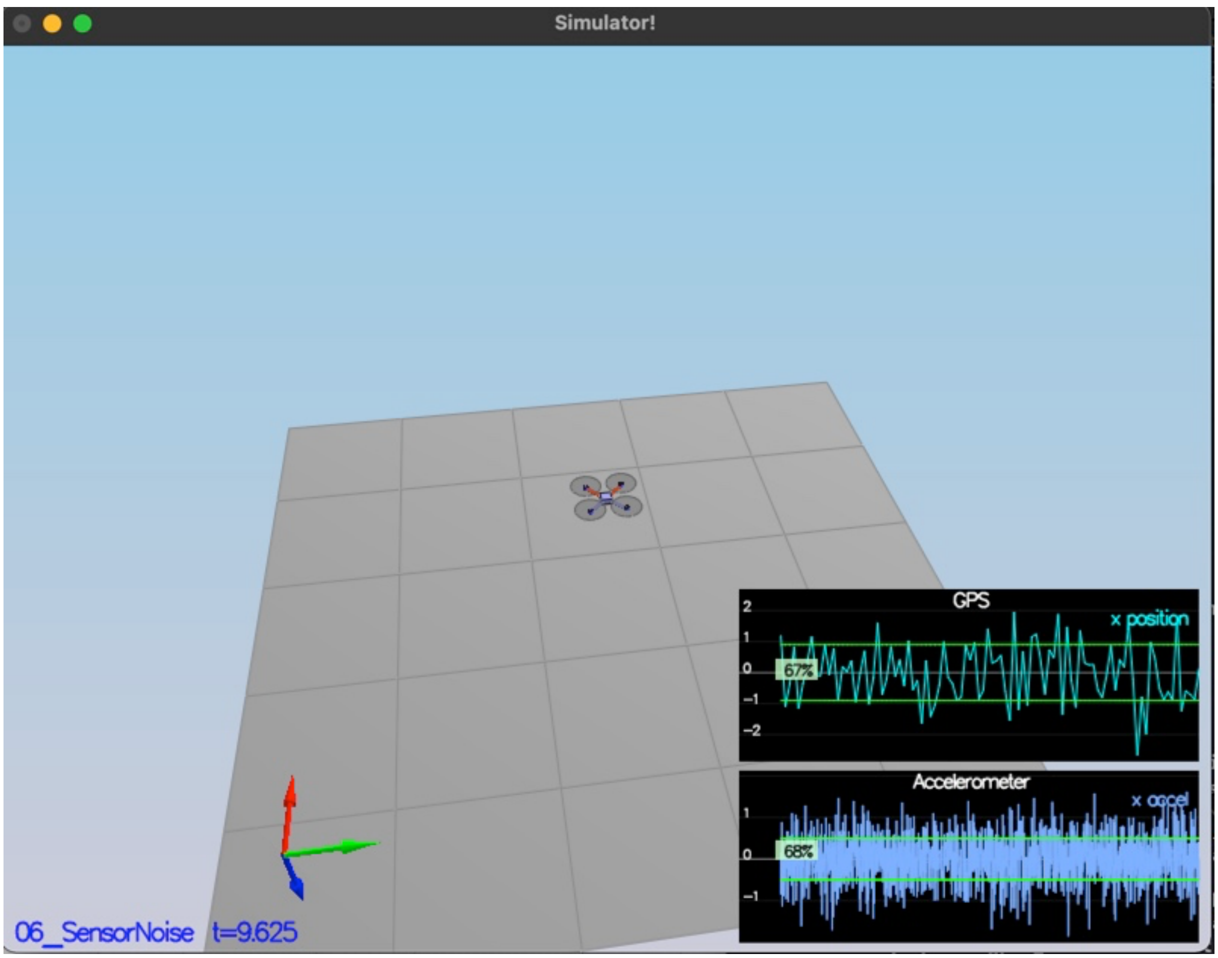
Estimation Project Write-up

06_NoisySensors

For this scenario MeaseuredStdDev_GPSPosXY and MeasuredStdDev_AccelXY were calculated based on the standard deviation from Graph1 and Graph2 data. My original estimation came as MeaseuredStdDev_GPSPosXY = 0.8 and MeasuredStdDev_AccelXY = 0.3 and I had to tune them to MeaseuredStdDev_GPSPosXY = 0.9 and MeasuredStdDev_AccelXY = 0.5, in order to satisfy the condition for scenario 6.



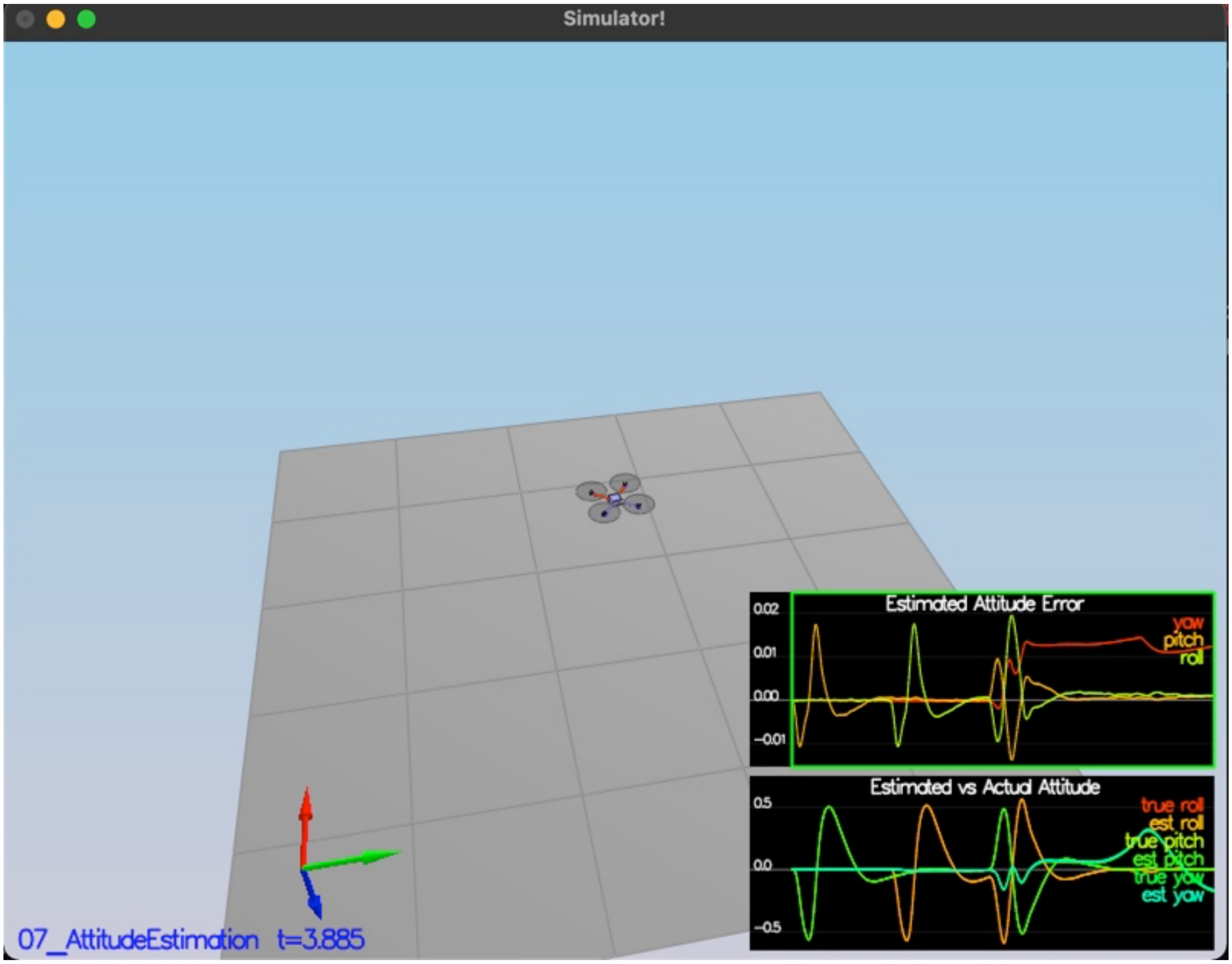
07_AttitudeEstimation

To implement better rate gyro attitude integration, the Euler's equation is used:

$$\begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = \begin{pmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi\sec\theta & \cos\phi\sec\theta \end{pmatrix} \times \begin{pmatrix} p \\ q \\ r \end{pmatrix}$$

Rotational Metrix is defined by R and Euler_dot vector is calculated based on the equation above. Finally predicted values for pitch, roll and yaw was calculated using the euler_dot values

```
///////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
// SMALL ANGLE GYRO INTEGRATION:
// (replace the code below)
// make sure you comment it out when you add your own code -- otherwise e.g. you might integrate yaw twice

//float predictedPitch = pitchEst + dtIMU * gyro.y;
//float predictedRoll = rollEst + dtIMU * gyro.x;
//ekfState(6) = ekfState(6) + dtIMU * gyro.z; // yaw
float p = pitchEst;
float r = rollEst;

Mat3x3F R;
R(0,0) = 1;
R(1,0)= 0;
R(2,0)= 0;
R(0,1) = sin(r) * tan(p);
R(1,1) = cos(r);
R(2,1) = sin(r) / cos(p);
R(0,2) = cos(r) * tan(p);
R(1,2) = -sin(r);
R(2,2) = cos(r) / cos(p);
V3F euler_dot = R * gyro;

float predictedPitch = pitchEst + euler_dot.y * dtIMU;
float predictedRoll = rollEst + euler_dot.x * dtIMU;
ekfState(6) = ekfState(6) + euler_dot.z * dtIMU;

// normalize yaw to -pi .. pi
if (ekfState(6) > F_PI) ekfState(6) -= 2.f*F_PI;
if (ekfState(6) < -F_PI) ekfState(6) += 2.f*F_PI;

///////////////////////////////// END STUDENT CODE ///////////////////////////
```



The estimated error for each of Euler angles was within 0.1 rad, for at least 3 seconds.

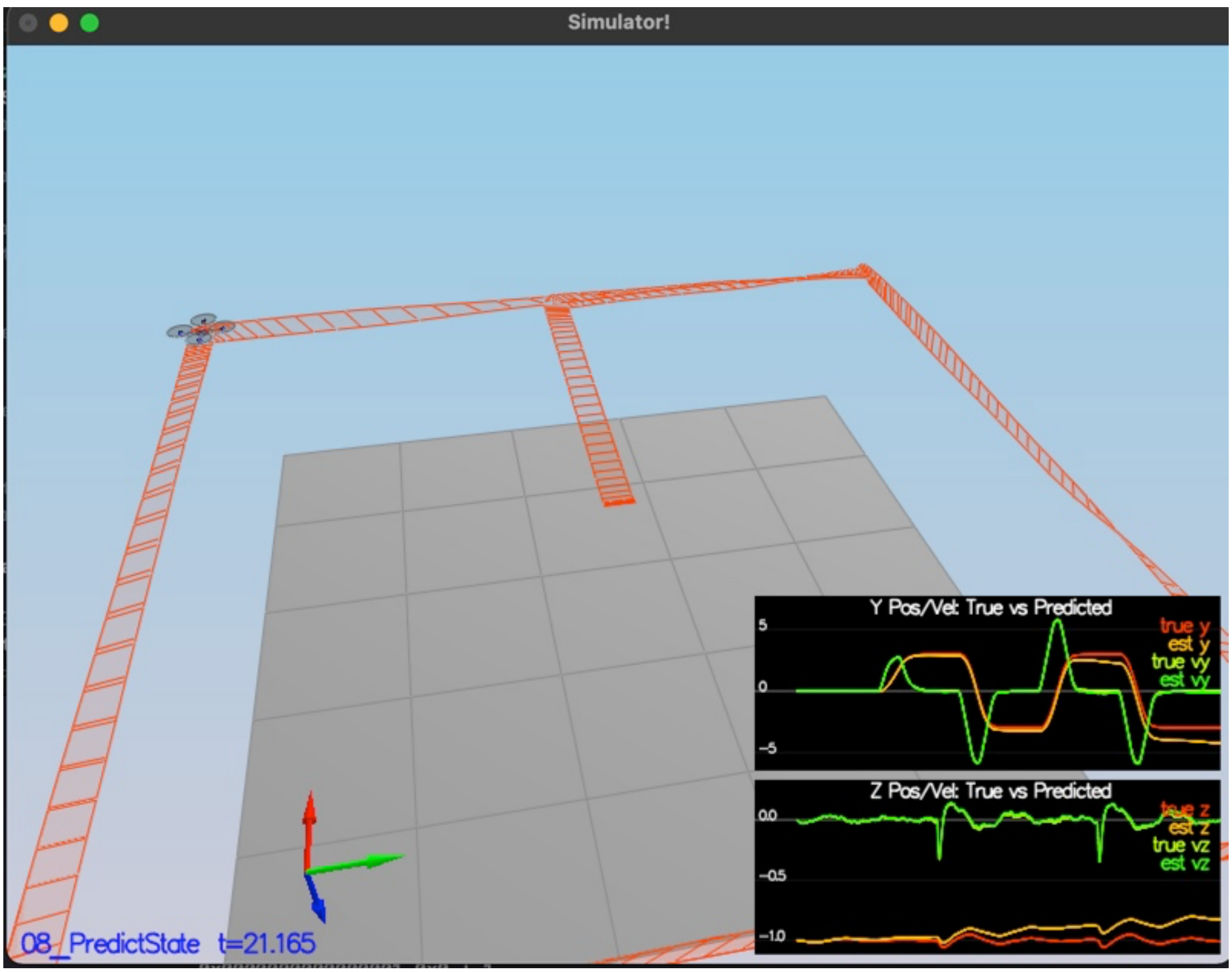
08_PredictState

In order to update predictedState vector, first acceleration was transformed into inertial frame by using rotation Metrix and adding gravitational acceleration. As directed, since the time duration for this prediction is shot a simplified integration method is used:

$$X_2 = X_1 + \dot{X}\,.\Delta t$$

```
    Quaternion<float> attitude = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, curState(6));

    ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
    V3F accelBtoI = attitude.Rotate_BtoI(accel) + V3F (0.0, 0.0, -CONST_GRAVITY);
    predictedState(0) = predictedState(0) + predictedState(3) * dt;
    predictedState(1) = predictedState(1) + predictedState(4) * dt;
    predictedState(2) = predictedState(2) + predictedState(5) * dt;
    predictedState(3) = predictedState(3) + accelBtoI.x * dt;
    predictedState(4) = predictedState(4) + accelBtoI.y * dt;
    predictedState(5) = predictedState(5) + accelBtoI.z * dt;

    ////////////////////////////// END STUDENT CODE ///////////////////////////

    return predictedState;
  }
```

As a results estimated state follows the actual state with slight drift.



09_PredictionCov

To work with realistic IMU, we start by implementing RbgPrime Metrix using the equation below:

$$R'_{bg} = \begin{bmatrix} -\cos\theta\sin\psi & -\sin\phi\sin\theta\sin\psi - \cos\phi\cos\psi & -cos\phi\sin\theta\sin\psi + \sin\phi\cos\psi \\ \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ 0 & 0 & 0 \end{bmatrix} \tag{52}$$

```
  ///////////////////////////// BEGIN STUDENT CODE ///////////////////////////
  RbgPrime(0,0) = - cos(pitch) * sin(yaw);
  RbgPrime(0,1) = -sin(roll) * sin(pitch) * sin(yaw) - cos(roll) * cos(yaw);
  RbgPrime(0,2) =  -cos(roll) * sin(pitch) * sin(yaw) + sin(roll) * cos(yaw);
  RbgPrime(1,0) = cos(pitch) * cos(yaw);
  RbgPrime(1,1) = sin(roll) * sin(pitch) * cos(yaw) - cos(roll) * sin(yaw);
  RbgPrime(1,2) = cos(roll) * sin(pitch) * cos(yaw) + sin(roll) * sin(yaw);
  ///////////////////////////// END STUDENT CODE ///////////////////////////

  return RbgPrime;
}
```

Later we use RbgPrime in Predict function to build in Jacobian of transition function as follow:

$$g'(x_t, u_t, \Delta t) = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \frac{\partial}{\partial x_{t,\psi}}(x_{t,\dot{x}} + R_{bg}[0:]u_t[0:3]\Delta t) \\ 0 & 0 & 0 & 0 & 1 & 0 & \frac{\partial}{\partial x_{t,\psi}}(x_{t,\dot{y}} + R_{bg}[1:]u_t[0:3]\Delta t) \\ 0 & 0 & 0 & 0 & 0 & 1 & \frac{\partial}{\partial x_{t,\psi}}(x_{t,\dot{z}} + R_{bg}[2:]u_t[0:3]\Delta t) \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{50}$$

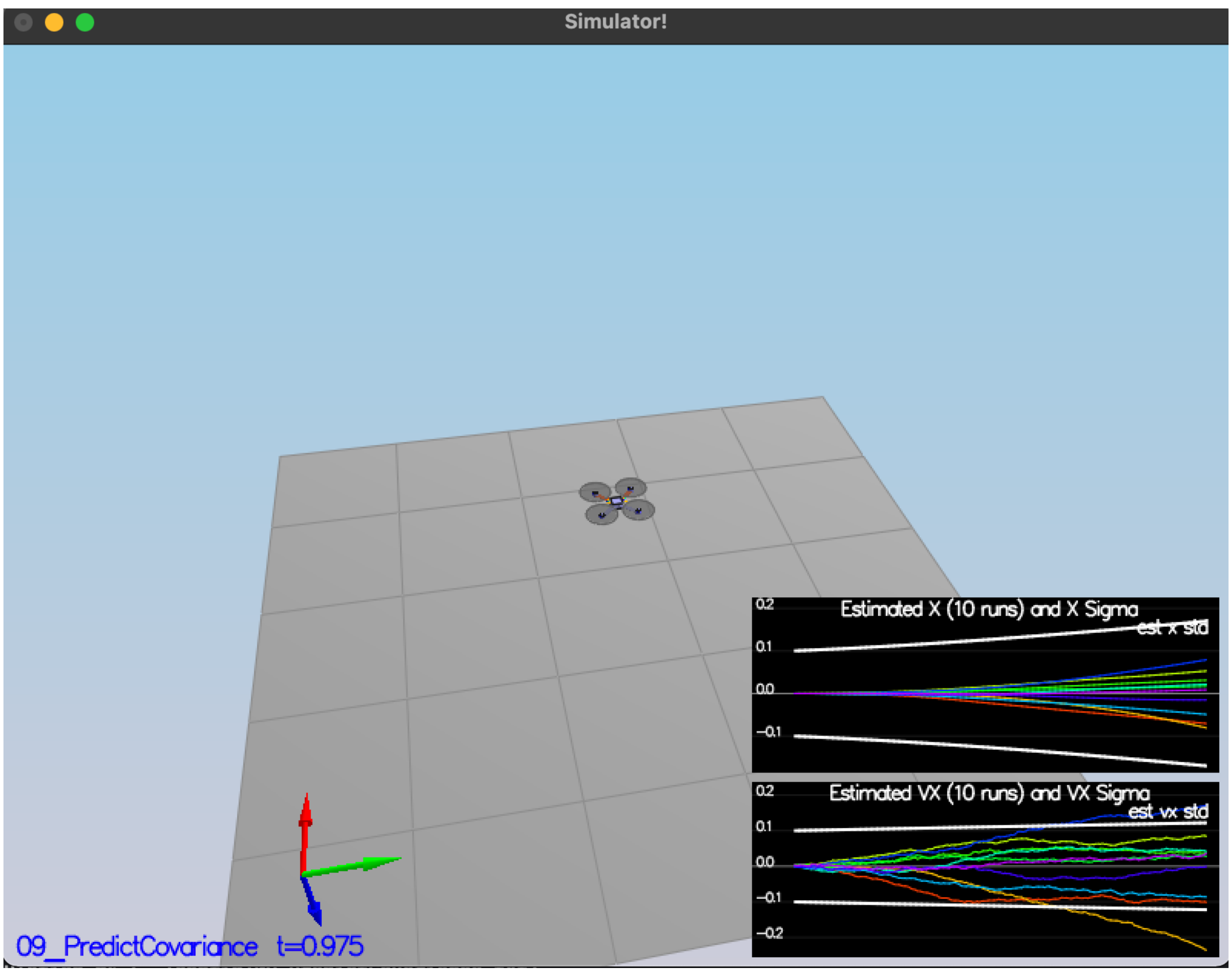
$$= \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & R'_{bg}[0:]u_t[0:3]\Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 & R'_{bg}[1:]u_t[0:3]\Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 & R'_{bg}[2:]u_t[0:3]\Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{51}$$

And calculating new covariance:

$$G_t = g'(u_t, x_t, \Delta t)$$
$$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + Q_t$$

```
  // we'll want the partial derivative of the Rbg matrix
  MatrixXf RbgPrime = GetRbgPrime(rollEst, pitchEst, ekfState(6));

  // we've created an empty Jacobian for you, currently simply set to identity
  MatrixXf gPrime(QUAD_EKF_NUM_STATES, QUAD_EKF_NUM_STATES);
  gPrime.setIdentity();

  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
  gPrime(0, 3) = dt;
  gPrime(1, 4) = dt;
  gPrime(2, 5) = dt;
  gPrime(3, 6) = (RbgPrime(0, 0) * accel.x + RbgPrime(0, 1) * accel.y + RbgPrime(0, 2) * accel.z) * dt;
  gPrime(4, 6) = (RbgPrime(1, 0) * accel.x + RbgPrime(1, 1) * accel.y + RbgPrime(1, 2) * accel.z) * dt;
  gPrime(5, 6) = (RbgPrime(2, 0) * accel.x + RbgPrime(2, 1) * accel.y + RbgPrime(2, 2) * accel.z) * dt;
  gPrime(6, 6) = 1;

  ekfCov = gPrime * ekfCov * gPrime.transpose() + Q;
  ////////////////////////////// END STUDENT CODE ///////////////////////////

  ekfState = newState;
}
```

After tunning QPosXYStd and QVelXYStd process parameters, the magnitude of the error is captured as below:



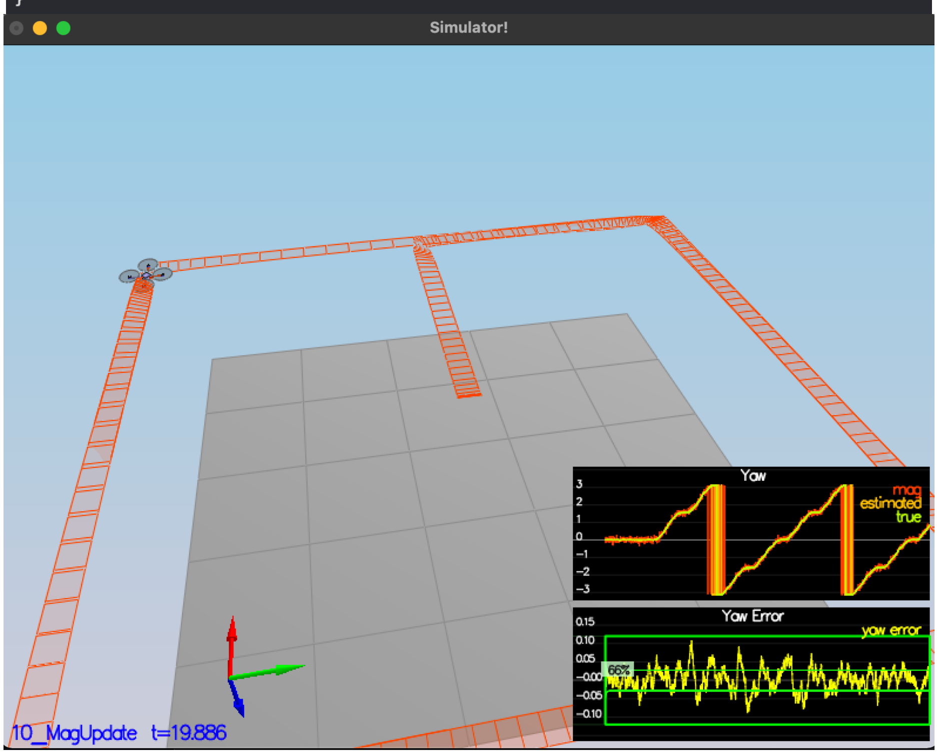
10_MagUpdate

UpdateFromMag function is written based on the measurement model below:

$$h(x_t) = \begin{bmatrix} x_{t,\psi} \end{bmatrix}$$

$$h'(x_t) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

```
void QuadEstimatorEKF::UpdateFromMag(float magYaw)
{
  VectorXf z(1), zFromX(1);
  z(0) = magYaw;

  MatrixXf hPrime(1, QUAD_EKF_NUM_STATES);
  hPrime.setZero();

  // MAGNETOMETER UPDATE
  // Hints:
  //  - Your current estimated yaw can be found in the state vector: ekfState(6)
  //  - Make sure to normalize the difference between your measured and estimated yaw
  //    (you don't want to update your yaw the long way around the circle)
  //  - The magnetomer measurement covariance is available in member variable R_Mag
  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
  hPrime(0, 6) = 1;
  zFromX(0) = ekfState(6);
  float yawDiff = z(0) - zFromX(0);

  // normalize yaw to -pi .. pi
  if (yawDiff > F_PI) z(0) -= 2.f * F_PI;
  else if (yawDiff < -F_PI) z(0) += 2.f * F_PI;

  ////////////////////////////// END STUDENT CODE ///////////////////////////

  Update(z, hPrime, R_Mag, zFromX);
}
```



11_GPSUpdate

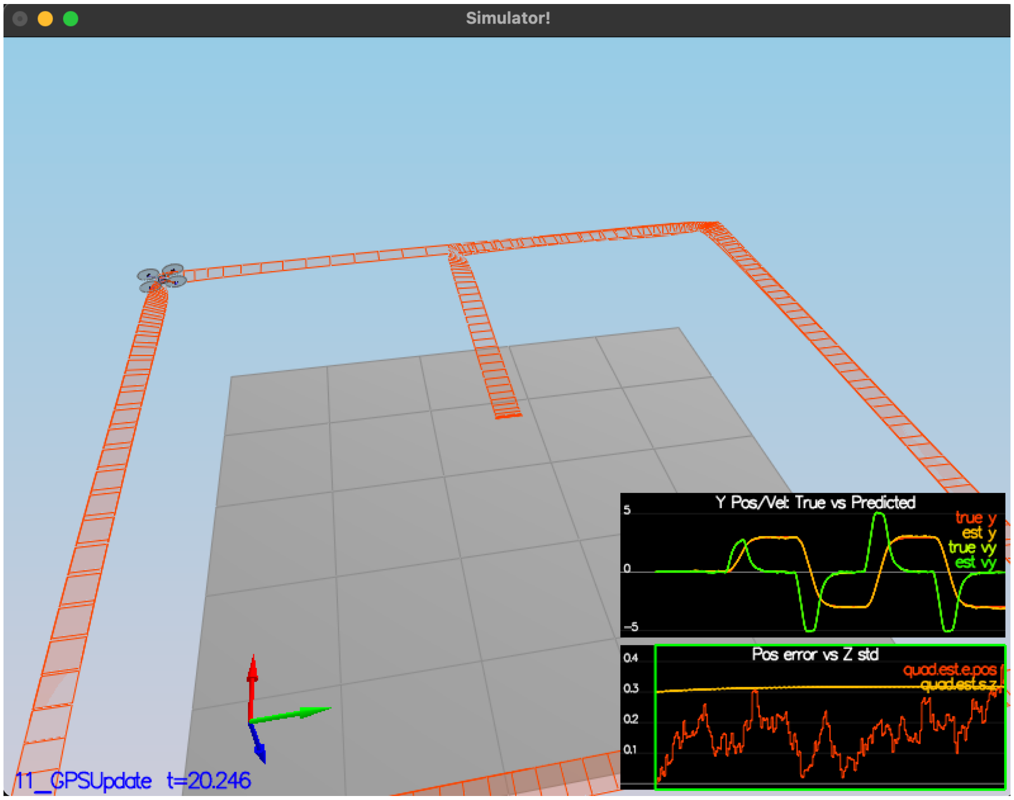
UpadteFromGPS function is written based on the measurement model below:

$$h(x_t) = \begin{bmatrix} x_{t,x} \\ x_{t,y} \\ x_{t,z} \\ x_{t,\dot{x}} \\ x_{t,\dot{y}} \\ x_{t,\dot{z}} \end{bmatrix}$$

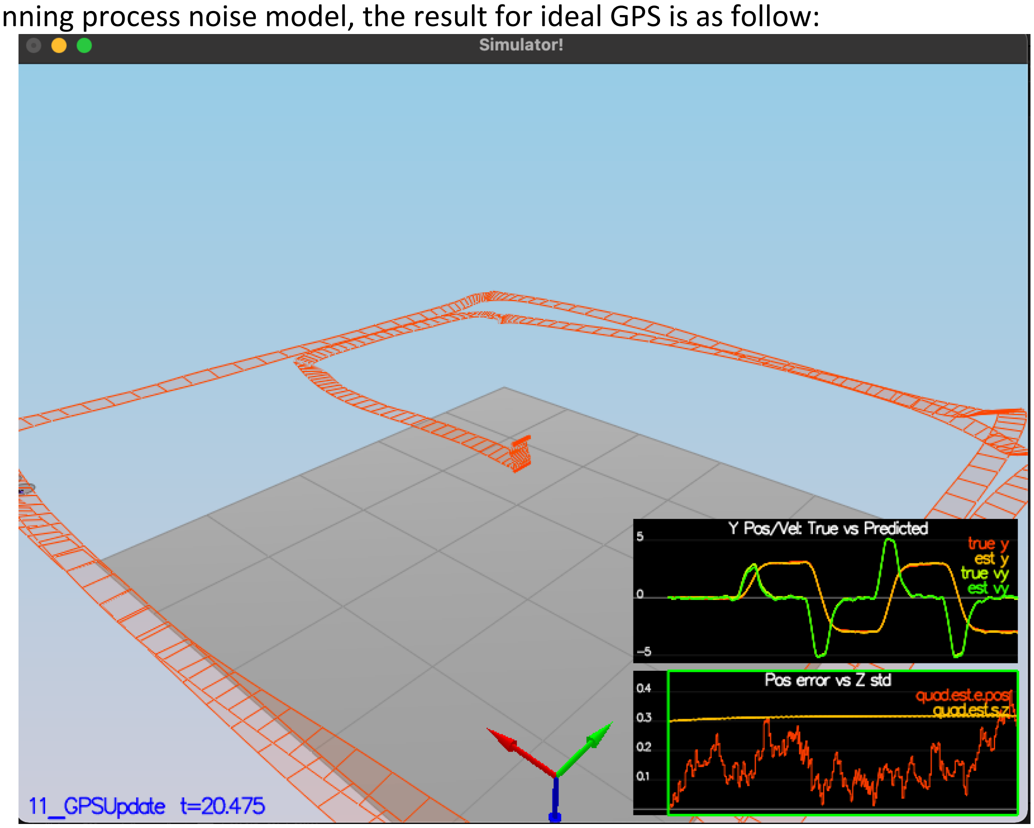
$$h'(x_t) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

```
void QuadEstimatorEKF::UpdateFromGPS(V3F pos, V3F vel)
{
  VectorXf z(6), zFromX(6);
  z(0) = pos.x;
  z(1) = pos.y;
  z(2) = pos.z;
  z(3) = vel.x;
  z(4) = vel.y;
  z(5) = vel.z;

  MatrixXf hPrime(6, QUAD_EKF_NUM_STATES);
  //hPrime.setZero();

  // GPS UPDATE
  // Hints:
  //  - The GPS measurement covariance is available in member variable R_GPS
  //  - this is a very simple update
  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
  hPrime.setIdentity();

  zFromX(0) = ekfState(0);
  zFromX(1) = ekfState(1);
  zFromX(2) = ekfState(2);
  zFromX(3) = ekfState(3);
  zFromX(4) = ekfState(4);
  zFromX(5) = ekfState(5);
  ////////////////////////////// END STUDENT CODE /////////////////////////////

  Update(z, hPrime, R_GPS, zFromX);
}
```

Results in prediction as shown below for ideal GPS:



After tunning process noise model, the result for ideal GPS is as follow:



Adding the controller design from previous project: