This Program is written by Abuhanif Bhuiyan for the completion of
the Project - 3 of the udacity FCND program.Submitted on May 30, 2018

This is the writeup for the C++ Controls project.

Below are the notes on each of the required sections for this project.

GenerateMotorCommands

Used the arm length parameter L, and the drag/thrust ratio kappa. We can get the moment of each motors from the equations below:

$$\begin{pmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{pmatrix} = T^{-1} \begin{pmatrix} c_c \\ p_c \\ q_c \\ r_c \end{pmatrix}$$

and

$$\begin{pmatrix} 1 & 1 & 1 & -1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 \end{pmatrix} \times 0.25$$

From class lessons I derived the GeneratMotorCommands using the following C++ code.

```
  float c_cmd = collThrustCmd;
  float p_cmd = momentCmd[0] / L;
  float q_cmd = momentCmd[1] / L;
  float r_cmd = momentCmd[2] / kappa;

  cmd.desiredThrustsN[0] = (c_cmd + p_cmd + q_cmd - r_cmd) / 4.f; //
front left
  cmd.desiredThrustsN[1] = (c_cmd - p_cmd + q_cmd + r_cmd) / 4.f; //
front right
  cmd.desiredThrustsN[2] = (c_cmd + p_cmd - q_cmd + r_cmd) / 4.f; // rear
left
  cmd.desiredThrustsN[3] = (c_cmd - p_cmd - q_cmd - r_cmd) / 4.f; // rear
right
```

Body Rate Control

Body Rate Control is derived by multiplying the Inertia (x, y, z) components * the PQRError * kpPQR varaible.
And that yields the momentCmd.

Implemented in the project by the following codes:

```
  momentCmd = Inertia * kpPQR * (pqrCmd - pqr);
```

RollPitchControl,

These controls are very similar to what we did in Exercise 5 of the lesson.

The roll-pitch controller is a P controller responsible for commanding the roll and pitch rates in the body frame. First, it sets the desired rate of change of the given matrix elements using a P controller. The given values can be converted into the angular velocities into the body frame by the matrix multiplication. Implemented by the following codes:

```
  // Current attitude
  const float b_x_a = R(0, 2);
  const float b_y_a = R(1, 2);

  // Target attitude
  const float thrust_acceleration = -collThrustCmd / mass;
  const float b_x_c = accelCmd.x / (thrust_acceleration);
  const float b_y_c = accelCmd.y / (thrust_acceleration);

  // Commanded rates in world frame
  const float b_x_c_dot = kpBank * (b_x_c - b_x_a);
  const float b_y_c_dot = kpBank * (b_y_c - b_y_a);

  // Roll and pitch rates
  const float r_33_inv = 1.0F / R(2, 2);
  pqrCmd.x = r_33_inv * (R(1, 0)*b_x_c_dot - R(0, 0)*b_y_c_dot);
  pqrCmd.y = r_33_inv * (R(1, 1)*b_x_c_dot - R(0, 1)*b_y_c_dot);
  pqrCmd.z = 0.0F;  // yaw controller set in YawControl
```

altitudeControl

The controller used both the down position and the down velocity to command thrust. The output value is indeed thrust and the drone's mass also accounted for. The thrust includes the non-linear effects from non-zero roll/pitch angles. Implemented by the following codes:

```
  float z_target = posZCmd;
  float z_dot_target = velZCmd;
  float z_actual = posZ;
  float z_dot_actual = velZ;
  float z_dot_dot_target = accelZCmd;

  float u_bar_1 = (kpPosZ * (z_target - z_actual)) + (kpVelZ *
(z_dot_target - z_dot_actual)) + (KiPosZ * integratedAltitudeError) +
z_dot_dot_target;
  float b = R(2, 2);
  float c = (u_bar_1 - CONST_GRAVITY) / b;
  float c_constrain = CONSTRAIN(c, -maxAscentRate / dt, maxAscentRate /
dt);
  thrust = -c_constrain * mass;
```

The altitude controller also contains an integrator to handle the weight non-idealities presented in scenario 4 using the following code:

```
// Integrator
```

```
integratedAltitudeError = integratedAltitudeError + ((z_target -
z_actual) * dt);
```

LateralPositionControl

Here, instead of doing calculations for each x, y component I used the V3F directly. It is an efficient way to calculate it.
I constrained velocity using maxSpeedXY. Used the following codes to achieve that.

```
 // Constrain desired velo
  velCmd.x = CONSTRAIN(velCmd.x, -maxSpeedXY, maxSpeedXY);
  velCmd.y = CONSTRAIN(velCmd.y, -maxSpeedXY, maxSpeedXY);

  // Compute PD controller + feedforward
  const V3F error = posCmd - pos;
  const V3F error_dot = velCmd - vel;

  accelCmd = kpPosXY * error + kpVelXY * error_dot + accelCmd;

  // Constrain desired accell
  accelCmd.x = CONSTRAIN(accelCmd.x, -maxAccelXY, maxAccelXY);
  accelCmd.y = CONSTRAIN(accelCmd.y, -maxAccelXY, maxAccelXY);
  accelCmd.z = 0.0F;
```

YawControl

Important thing in the yaw controller implementation is the implementation of the fmod command to keep the value within the boundary limit. The Yawcontroller is implemented in the program by the following code:

```
yawRateCmd = kpYaw * (fmod(yawCmd - yaw, 2 * M_PI));
```

Comments
My C++ controller is successfully able to fly the provided test trajectory and visually passes inspection of the scenarios leading up to the test trajectory.