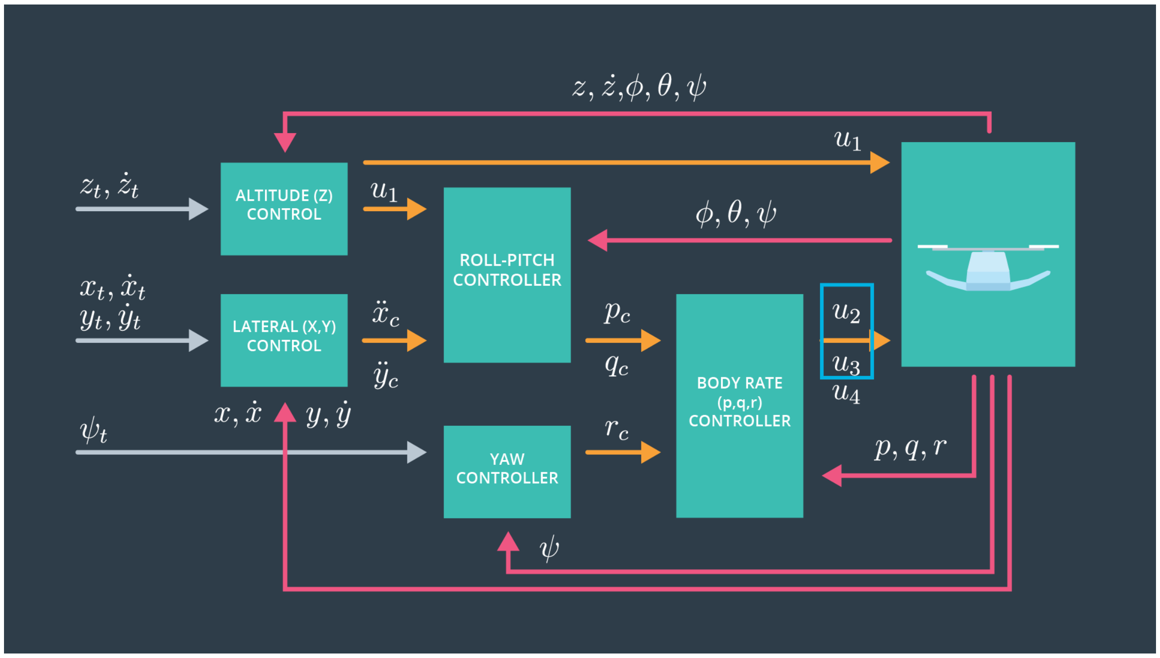

1: **Implemented body rate control in C++. :** The controller should be a proportional controller on body rates to commanded moments. The controller should take into account the moments of inertia of the drone when calculating the commanded moments.

Solution:

The body rate controller takes as input the pc,qc,rc input commands and returns the desired moments comand (3 rotational moment commands). It uses the p,q,r as gains.

Considering this, pc error was calculated as pError, qc error was calculated as qError, rc error was calulated as rError which were used to calulate moment in X,Y,Z directions as momentCmd.x, momentCmd.y, momentCmd.z together forming momentCmd.

```
float pError = pqrCmd[0] - pqr[0];

float qError = pqrCmd[1] - pqr[1];

float rError = pqrCmd[2] - pqr[2];


momentCmd.x = kpPQR[0] * pError * Ixx;

momentCmd.y = kpPQR[1] * qError * Iyy;

momentCmd.z = kpPQR[2] * rError * Izz;
```

2. **Implement roll pitch control in C++.:** The controller should use the acceleration and thrust commands, in addition to the vehicle attitude to output a body rate command. The controller should account for the non-linear transformation from local accelerations to body rates. Note that the drone's mass should be accounted for when calculating the target angles.

Solution:

It takes a thrust command, x and y accelerations and the attitude of the drone (φ,ψ,θ) and outputs the p and q commands. p_c, q_c. As you can see from the implementation the mass of the drone is accounted when calculating the target angles.

```
if ( collThrustCmd > 0 ) {

    float c = - collThrustCmd / mass;

    float b_x_cmd = CONSTRAIN(accelCmd.x / c, -maxTiltAngle, maxTiltAngle);

    float b_x_err = b_x_cmd - R(0,2);

    float b_x_p_term = kpBank * b_x_err;


    float b_y_cmd = CONSTRAIN(accelCmd.y / c, -maxTiltAngle, maxTiltAngle);

    float b_y_err = b_y_cmd - R(1,2);

    float b_y_p_term = kpBank * b_y_err;


    pqrCmd.x = (R(1,0) * b_x_p_term - R(0,0) * b_y_p_term) / R(2,2);

    pqrCmd.y = (R(1,1) * b_x_p_term - R(0,1) * b_y_p_term) / R(2,2);

  } else {

    pqrCmd.x = 0.0;

    pqrCmd.y = 0.0;

  }


  pqrCmd.z = 0;
```

3: **Implement altitude controller in C++.:** The controller should use both the down position and the down velocity to command thrust. Ensure that the output value is indeed thrust (the drone's mass needs to be accounted for) and that the thrust includes the non-linear effects from non-zero roll/pitch angles.

Additionally, the C++ altitude controller should contain an integrator to handle the weight non-idealities presented in scenario 4.

Solution :

The altitude controller's job is to see that the vehicle stayes close to the commanded set position and velocity by computing a thrust value. The output thrust is sent to the roll pitch controller. Because the commanded thrust is going to be shared across all dimensions. The portion that points in the x,y will determine acceleration in those directions.

```
float b_z = R(2, 2);

  float d_term = 0;

  velZCmd = -CONSTRAIN(-velZCmd, -maxDescentRate, maxAscentRate);

  float e = posZCmd - posZ;

  integratedAltitudeError += KiPosZ * e * dt;


  float u_bar_1 = kpPosZ * (posZCmd - posZ) + kpVelZ * (velZCmd - velZ) + accelZCmd +
integratedAltitudeError + d_term;

  float accelZ = (u_bar_1 - 9.81f) / b_z;

  if (accelZ > 0) {

      accelZ = 0;

  }


  thrust = -accelZ * mass;
```

4: **Implement lateral position control in C++. : The controller should use the local NE position and velocity to generate a commanded local acceleration.**

Solution:

The desired effect can be achived by a PD controller in the x and y directions, which will generate an acceleration in the x-y directions which will be then sent to the roll pitch controller. The negative sign shows that the position is in NE.

```
V3F kpPos;

  kpPos.x = kpPosXY;

  kpPos.y = kpPosXY;
  kpPos.z = 0.f;


  V3F kpVel;

  kpVel.x = kpVelXY;

  kpVel.y = kpVelXY;

  kpVel.z = 0.f;


  V3F capVelCmd;

  if ( velCmd.mag() > maxSpeedXY ) {

    capVelCmd = velCmd.norm() * maxSpeedXY;

  } else {

    capVelCmd = velCmd;

  }


  accelCmd = kpPos * ( posCmd - pos ) + kpVel * ( capVelCmd - vel ) + accelCmd;


  if ( accelCmd.mag() > maxAccelXY ) {

    accelCmd = accelCmd.norm() * maxAccelXY;

  }
```

5: **Implement yaw control in C++. : The controller can be a linear/proportional heading controller to yaw rate commands (non-linear transformation not required).**

Solution:

Yaw control is controller is imlemented via linear colntroller to control yaw.

```
 float yaw_error = yawRateCmd - yaw;

    yaw_error = fmodf(yaw_error, F_PI*2.f);


    if (yaw_error >F_PI){

        yaw_error = yaw_error - 2.0f*F_PI;
```

```
  } else if (yaw_error < -M_PI){
      yaw_error = yaw_error + 2.0f*F_PI;
  }
  yawRateCmd = kpYaw*yaw_error;
```

6: **Implement calculating the motor commands given commanded thrust and moments in C++.:** The thrust and moments should be converted to the appropriate 4 different desired thrust forces for the moments. Ensure that the dimensions of the drone are properly accounted for when calculating thrust from moments.

Solution:

If the thrust from each moter be denoted by mt1,mt2,mt3,mt4, moments in each direction (X,Y,Z) be represented by moment.x,moment.y and moment.z, collThrustCmd denote the collective thrust, kappa denote the drag/thrust ratio and l denote the drone arm length over square root of two, then,

mt1 + mt2 - mt3 - mt4 = momentCmd.x / l

mt1 + mt2 - mt3 - mt4 = momentCmd.y / l

mt1 - mt2 + mt3 - mt4 = momentCmd.z / kappa

mt1 + mt2 + mt3 + mt4 = collThrustCmd

l = L*sqrt(2) (Perpendicular distance to axises)

kappa = toruqe/trust

Thus, we calculate the desiredTrurtsN array as:

```
 float l = L / sqrtf(2.f);
 float a = momentCmd.x / l;
 float b = momentCmd.y / l;
 float c = - momentCmd.z / kappa;
 float d = collThrustCmd;


 cmd.desiredThrustsN[0] = (a + b + c+ d)/4.f;  // front left
 cmd.desiredThrustsN[1] = (-a + b- c+ d)/4.f; // front right
 cmd.desiredThrustsN[2] = (a - b- c + d)/4.f ; // rear left
 cmd.desiredThrustsN[3] = (-a - b + c + d)/4.f; // rear right
 cmd.desiredThrustsN[0] =  CONSTRAIN(cmd.desiredThrustsN[0],minMotorThrust,maxMotorThrust);

  cmd.desiredThrustsN[1] = CONSTRAIN(cmd.desiredThrustsN[1],minMotorThrust,maxMotorThrust);

  cmd.desiredThrustsN[2] = CONSTRAIN(cmd.desiredThrustsN[2],minMotorThrust,maxMotorThrust);

  cmd.desiredThrustsN[3] = CONSTRAIN(cmd.desiredThrustsN[3],minMotorThrust,maxMotorThrust);
```

7: **Your C++ controller is successfully able to fly the provided test trajectory and visually passes inspection of the scenarios leading up to the test trajectory. : Ensure that in each scenario the drone looks stable and performs the required task. Specifically check that the student's controller is able to handle the non-linearities of scenario 4 (all three drones in the scenario should be able to perform the required task with the same control gains used).**

Solution:

The implementation passes all senerios.