Building a controller

(Note: Most of the code implelentations are Inspired by the full 3d controll module and python examples)

1. Body rate and roll/pitch control (scenario 2):
   A. Implement calculating the motor commands given commanded thrust and moments in C++:

Given the drag/thrust ratio(kappa), and using the angular velocities matrix while keeping in mind the thrust sign is reversed in the C++ implementation, with the following relations:

1) $k_{appa} = \frac{\tau}{F}$
2) $F = \frac{\tau}{k_{appa}}$
3) $F = \frac{\tau \times \sqrt{2}}{l}$

```
    float f_c = collThrustCmd;
    float f_p = momentCmd.x * (sqrt(2) / L);
    float f_q = momentCmd.y * (sqrt(2) / L);
    float f_r = momentCmd.z / kappa;

    cmd.desiredThrustsN[0] = (f_c + f_p + f_q + -f_r) / 4.f;      // front left
    cmd.desiredThrustsN[1] = (f_c + -f_p + f_q + f_r) / 4.f;     // front right
    cmd.desiredThrustsN[2] = (f_c + f_p + -f_q + f_r) / 4.f;    // rear left
    cmd.desiredThrustsN[3] = (f_c + -f_p + -f_q + -f_r) / 4.f; // rear right
```

   B. Implemented body rate control in C++:

Using moment formula $M = I * \psi$ which is moment of inertia times angular acceleration. Replacing ψ with pqrError times the proportional constant kpPQR.

```
V3F pqrError = pqrCmd - pqr;

    momentCmd.x = Ixx * kpPQR.x * pqrError.x;
    momentCmd.y = Iyy * kpPQR.y * pqrError.y;
    momentCmd.z = Izz * kpPQR.z * pqrError.z;
```

C. Implement roll pitch control in C++:

Using the following formulas and the provided rotation matrix: (also see the red comments in the code below)

$$\dot{b}_c^x = k_p\,(b_c^x - b_a^x)$$

$$\dot{b}_c^y = k_p\,(b_c^y - b_a^y)$$

$$\begin{pmatrix} p_c \\ q_c \end{pmatrix} = \frac{1}{R_{33}} \begin{pmatrix} R_{21} & -R_{11} \\ R_{22} & -R_{12} \end{pmatrix} \times \begin{pmatrix} \dot{b}_c^x \\ \dot{b}_c^y \end{pmatrix}$$

$b_c^x = \ddot{x}_{command}/c$ $\quad$ $b_c^y = \ddot{y}_{command}/c$ $\quad$ $a = F/m$ $\quad$ $b_a^x = R_{13}$ $\quad$ $b_a^y = R_{23}$

```
    float b_c_x_dot;
    float b_c_y_dot;

    float b_c_x;
    float b_c_y;



    float Row_1[2] = { R(1,0) , -R(0,0) };
    float Row_2[2] = { R(1,1) , -R(0,1) };

    float c_acc = -collThrustCmd / mass; //collective acceleration, thrust is minus "up"

    b_c_x = accelCmd.x / c_acc;         // calculating roll  around x axis
    b_c_y = accelCmd.y / c_acc;         // calculating pitch around y axis


    b_c_x = CONSTRAIN(b_c_x, - maxTiltAngle, maxTiltAngle); // limiting roll angle
    b_c_y = CONSTRAIN(b_c_y, - maxTiltAngle, maxTiltAngle); // limiting pitch angle


    b_c_x_dot = kpBank * (b_c_x - R(0, 2));
    b_c_y_dot = kpBank * (b_c_y - R(1, 2));



    for (int i = 0; i < 2; i++) {            //using for loop to multiply by 1/R33
        Row_1[i] = Row_1[i] / R(2, 2);
        Row_2[i] = Row_2[i] / R(2, 2);

    }

    pqrCmd.x = Row_1[0] * b_c_x_dot + Row_1[1] * b_c_y_dot; // calculating p_c
    pqrCmd.y = Row_2[0] * b_c_x_dot + Row_2[1] * b_c_y_dot; // calculating q_c

    pqrCmd.z = 0;                             // z component remain zero in the matrix
```

2. Position/velocity and yaw angle control (Scenario 3):
   A. Implement lateral position control in C++:

(See the red comments in the code bellow)

```
velCmd += kpPosXY * (posCmd - pos);  // accumulating velocity component
velCmd.x = CONSTRAIN(velCmd.x, -maxSpeedXY, maxSpeedXY);// limiting maximum x velocity
velCmd.y = CONSTRAIN(velCmd.y, -maxSpeedXY, maxSpeedXY);// limiting maximum y velocity

accelCmd += kpVelXY * (velCmd - vel) + accelCmdFF;// accumulating acceleration component
accelCmd.x = CONSTRAIN(accelCmd.x, -maxAccelXY, maxAccelXY);// limiting maximum x acceleration
accelCmd.y = CONSTRAIN(accelCmd.y, -maxAccelXY, maxAccelXY);// limiting maximum y acceleration
```

   B. Implement altitude controller in C++:

Implementing the following formulas: (See the red comments in the code bellow)

$$\overline{u1} = k_{p-z}(z_t - z_a) + k_{d-z}(\dot{z}_t - \dot{z}_a) + k_{i-z}(dt(z_t - z_a)) + \ddot{z}_t$$

$$c = ((\overline{u1} - g) * mass)/b^z$$

```
float thrust = 0;
float u_1_bar = 0;
float g = 9.8;
float b_z = R(2, 2);

velZCmd = CONSTRAIN(velZCmd, -maxAscentRate, maxDescentRate); // limiting the max commanded velocity
float pos_err = (posZCmd - posZ);
float p_term = kpPosZ * pos_err;

float i_err = pos_err * dt;
integratedAltitudeError += i_err; // accumulating integrator error



integratedAltitudeError = CONSTRAIN(integratedAltitudeError, -.27, .27); // limiting the max
integrator error

float i_term = KiPosZ * integratedAltitudeError;

float d_term = kpVelZ * (velZCmd - velZ);


u_1_bar = p_term + i_term + d_term + accelZCmd;



thrust = ((g - u_1_bar) * mass) / b_z;
```

C. Implement yaw control in C++: (See the red comments in the code bellow)

```
    float err = yawRateCmd - yaw; // calculating yaw error
     // calculate rate command
    yawRateCmd = kpYaw * err;      // yaw proportional term
```

3. Non-idealities and robustness (scenario 4):

Implementing the following formulas and adding an integrator term (See the red comments in the code bellow)

$$\overline{u1} = k_{p-z}(z_t - z_a) + k_{d-z}(\dot{z}_t - \dot{z}_a) + k_{i-z}(dt(z_t - z_a)) + \ddot{z}_t$$
$$c = ((\overline{u1} - g) * mass)/b^z$$

```
float thrust = 0;
float u_1_bar = 0;
float g = 9.8;
float b_z = R(2, 2);

velZCmd = CONSTRAIN(velZCmd, -maxAscentRate, maxDescentRate); // limiting the max commanded velocity
float pos_err = (posZCmd - posZ);
float p_term = kpPosZ * pos_err;

float i_err = pos_err * dt;
integratedAltitudeError += i_err; // accumulating integrator error



integratedAltitudeError = CONSTRAIN(integratedAltitudeError, -.27, .27); // limiting the max
integrator error

float i_term = KiPosZ * integratedAltitudeError;

float d_term = kpVelZ * (velZCmd - velZ);


u_1_bar = p_term + i_term + d_term + accelZCmd;



thrust = ((g - u_1_bar) * mass) / b_z;
```