To find the trust commands for the four rotors, I solved the equations below for F1, F2, F3 and F4

$$\tau_x = (F_1 + F_4 - F_2 - F_3)l$$
$$\tau_y = (F_1 + F_2 - F_3 - F_4)l$$
$$\tau_z = \tau_1 + \tau_2 + \tau_3 + \tau_4$$
Where $\tau_1 = k_m\omega_1^2$, $\tau_2 = -k_m\omega_2^2$, $\tau_3 = k_m\omega_3^2$, $\tau_4 = -k_m\omega_4^2$.

$$F_1 = k_f\omega_1^2,\ F_2 = k_f\omega_2^2,\ F_3 = k_f\omega_3^2,\ F_4 = k_f\omega_4^2.$$

$$F_{total} = F_1 + F_2 + F_3 + F_4.$$

Note that the intro would fail with this code, since I lose the variable 'mass'. The line 79 can be ‘uncommented out’ and used to pass the intro (here the trust is ‘thrust produced’ minus the gravitational force, note; the mass need to be adjusted in QuadControlParams.txt).
The BodyRateController is a simple P-controller with a parameter kpPQR (V3F vector). The code uses the inertia around the different axes.
I tuned kpPQR to 95, 95, 5. kpPQR.x and kpPQR.y worked in a range of values, while tuning kpPQR.z had no effect.
For the implementation of RollPitchControl() (a P-controller on two of the RotationMatrix elements), I used the formulas;

$$\dot{b}_c^x = k_p(b_c^x - b_a^x)$$
$$\dot{b}_c^y = k_p(b_c^y - b_a^y)$$
where $b_a^x = R_{13}$ and $b_a^y = R_{23}$.

and

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

I tried tuning kpBank to pass Scenario 2, but failed. Consulting the provided python code I see extra code to introduce some constraints. I transposed that code to c++ (line 154-158).

Scenario 2 now had a pass for a range of kpBank. At this point I choose the midway; kpBank = 20.
The LaterPositionControl function takes in NE position and velocity and generates a commanded acceleration in the shape of V3F vectors. To implement the code for a PD-controller I used the following math equations;

$$\ddot{x}_{\text{command}} = cb_c^x$$
$$\ddot{x}_{\text{command}} = k_p^x(x_t - x_a) + k_d^x(\dot{x}_t - \dot{x}_a) + \ddot{x}_t$$
$$b_c^x = \ddot{x}_{\text{command}}/c$$

And to implement the code for a PD-controller in AltitudeControl, I used the following math equations;

$$c = (\bar{u}_1 - g)/b^z$$

and

$$\bar{u}_1 = k_{p-z}(z_t - z_a) + k_{d-z}(\dot{z}_t - \dot{z}_a) + \ddot{z}_t$$

Then I tuned kpPosXY and kpPosZ, and also kpVelXY and kpVelZ, using the four to one rule as a guide line.
The YawControl() function is a P-controller. The parameter used in this controller (kpQPR.z) I adjusted from 5 to 7.
To pass scenario 4 I added the ‘I’ term to the controller making AltitudeControl() a PID-controller.
I had to tune kpPQR.z to 9, and kiPosZ to 80 to make it work.
And finally scenario 5 passed also!