C++ Tips and Tricks

Here are some tips and tricks you may find useful while writing your C++ code. Keeping in mind that the target platform for this code is a microcontroller, there are some additional factors you should consider:

- Use floats, don't use doubles (in practice, floats will run much faster on many microcontrollers than doubles)
- Please don't add additional library dependencies or allocate memory or use fancy STL features (you shouldn't need to), if you keep your dependencies and code complexity down, the code you implement should easily port to real hardware. A real embedded environment is often stripped down to bare essentials, and even basic C++ STL features like `<string>` or `<map>` are often missing.

Functions you may find useful:

- Lots of the code uses a data type called V3F, this is a convenience class for working with vectors of 3 floats. For example, position/velocity/acceleration in 3D, body rates, etc.
- `CONSTRAIN()` is a function that allows you to constrain your value within a set of bounds. This is very helpful to have an upper and lower limit on a command (e.g. limiting a drone's bank angle).

Parameter Tuning Tips and Tricks

Tuning the controller can be tricky. The parameters for your controller are defined in a file called `QuadControlParams.txt`. In that file you'll see the following initial values for the parameters

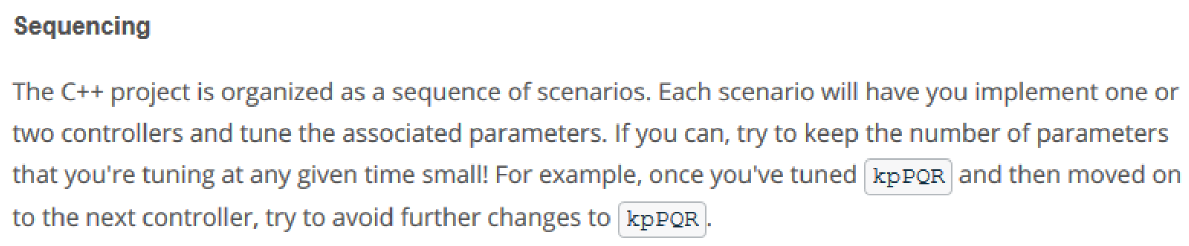
```
# Position control gains
kpPosXY = 1
kpPosZ = 1
KiPosZ = 20

# Velocity control gains
kpVelXY = 4
kpVelZ = 4

# Angle control gains
kpBank = 5
kpYaw = 1

# Angle rate gains
kpPQR = 23, 23, 5
```

Approximate parameter values

These initial values are all **too low**. To get our "solution" controller properly tuned we needed values that were between 2 and 4 times larger than the values shown here.

Approximate parameter ratios

Sergei Lupashin put together a one-page document deriving the ratio of velocity gains to position gains for a "critically damped" system (where the damping ratio is 1). In that document he shows that the ratio Kv/Kp=4 for such a system.



Sequencing

The C++ project is organized as a sequence of scenarios. Each scenario will have you implement one or two controllers and tune the associated parameters. If you can, try to keep the number of parameters that you're tuning at any given time small! For example, once you've tuned `kpPQR` and then moved on to the next controller, try to avoid further changes to `kpPQR`.

Dronecode provides some good guidelines on PID tuning.

Acknowledgement

The C++ simulator and project scenarios were largely designed and built by Fotokite. Big thanks to the Fotokite team (and Sergei Lupashin in particular) for their great work!