Lab 5. Hover control

Do this before coming to the lab:

- **Review your notes on the dynamic model of the quadcopter, the sensors, and state estimation.**
- **Make sure your batteries are charged.**

Important notes:

- **Read the "Deliverables" section before starting work, so that you know what information you need to record (and which photos to take) as you are proceeding through the lab.**
- **This lab will involve free flight. You *must* fly only within the provided netting. *If you fly (at all) outside of the netting, you will be asked to leave – no exceptions.***
- **Wear your safety glasses throughout the lab. You are most likely to be injured by someone else's stupidity – even if you think you're acting safely, don't assume the same of others.**
- **Make sure your batteries are charged, and charge your spare battery while using the other, so that you are not slowed down by low batteries.**
- **You will likely not be able to finish the entire lab on the first day. Note that the delivery date for this lab is different from what you had for the other labs.**

Last update: 2017-10-30 at 13:02:40

This lab will be the most comprehensive so far, and we will take everything we have done so far, put it together, and make the vehicle hover. This will be complex, and involve multiple separate components; make sure you read the whole description before you get started. Note that the deliverables for this lab are substantially longer than for the previous labs, but that the deadline is shifted back.

The block diagram of the resulting system is shown in Figure 5.1.

From the previous labs, you should currently have a drone that is able to control its attitude, and produce a normalized thrust simultaneously (that is, a total thrust force normalized by mass).

We will begin this lab by implementing the state estimation for the vertical motion, and the horizontal velocity. Then we will implement the additional controllers.

5.1 Preparations

5.1.1 Optical flow and ranging board

You will receive an additional board, which you are to attach to your quadcopter. The board attaches to the bottom of the vehicle, and **it is crucial that you mount the board correctly, or risk damaging the board and your vehicle**. The board is mounted as shown in the figure below: important is that the arrow on the board lines up with the arrow on the quadcopter, as shown in the figure. The board contains both the distance sensor, and the optical flow sensor.

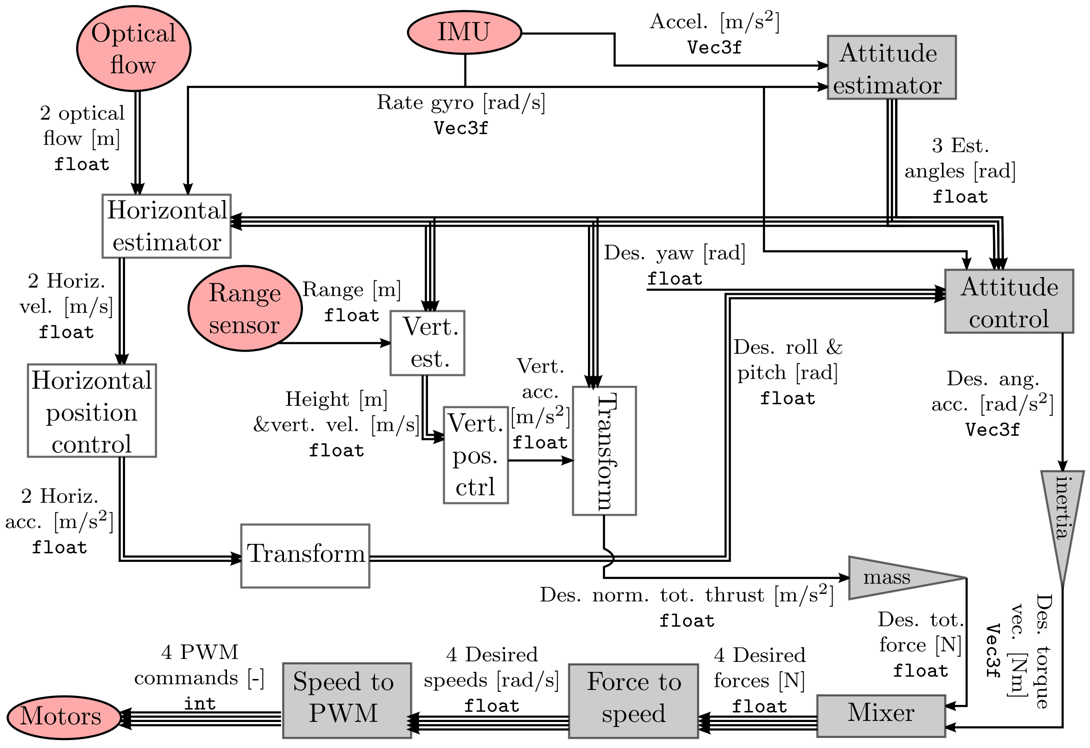
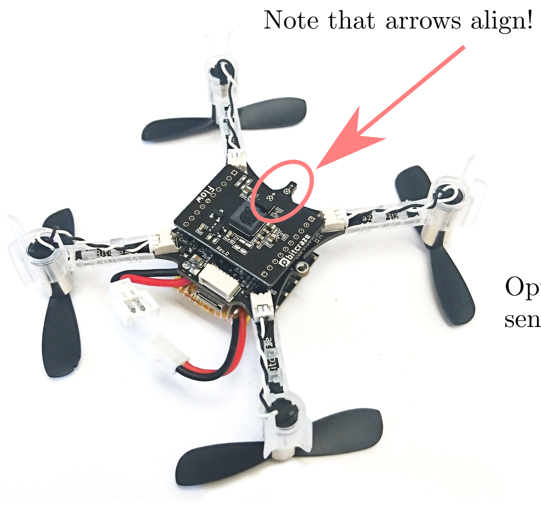
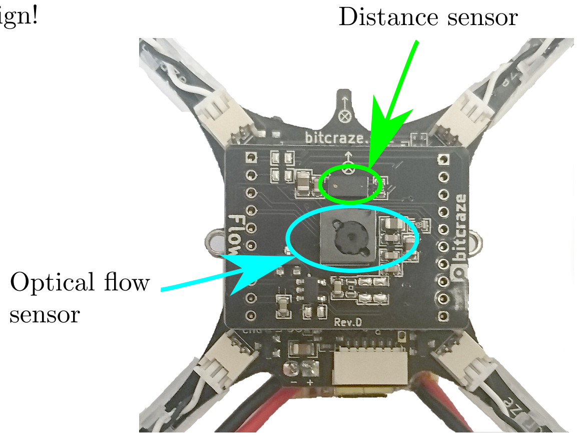

Optical flow
IMU
Accel. [m/s^2] Vec3f
Attitude estimator
Rate gyro [rad/s] Vec3f
2 optical flow [m] float
3 Est. angles [rad] float
Horizontal estimator
Des. yaw [rad] float
Range sensor
Range [m] float
Vert. est.
Attitude control
2 Horiz. vel. [m/s] float
Horizontal position control
Height [m] &vert. vel. [m/s] float
Vert. pos. ctrl
Vert. acc. [m/s^2] float
Transform
Des. roll & pitch [rad] float
Des. ang. acc. [rad/s^2] Vec3f
inertia
2 Horiz. acc. [m/s^2] float
Transform
mass
Des. norm. tot. thrust [m/s^2] float
Des. tot. force [N] float
Des. torque vec. [Nm] Vec3f
4 PWM commands [-] int
Speed to PWM
4 Desired speeds [rad/s] float
Force to speed
4 Desired forces [N] float
Mixer
Motors

Figure 5.1: System block diagram. The shaded boxes are components that we built in the prior labs, and hardware is shown as shaded ellipses. The triangular elements represent simple multiplications, and unshaded components are the focus of this lab (and are more complex algorithms).





We will first confirm that the sensors are, in fact, working. Add the following lines to your `PrintStatus()` function in `UserCode.cpp`:

```
//in PrintStatus():
printf("Last range = %6.3fm, ", \
  double(lastMainLoopInputs.heightSensor.value));
printf("Last flow: x=%6.3f, y=%6.3f\n", \
  double(lastMainLoopInputs.opticalFlowSensor.value_x), \
  double(lastMainLoopInputs.opticalFlowSensor.value_y));
```

Flash this to the vehicle, and connect over USB. Type `quad status`, and check the output. Run this repeatedly, and confirm that the range makes sense (note that the sensor returns approximately 8m if the measurement is out of range). Also check the flow sensor – hold the vehicle stationary at a height of $\sim$ 20cm above the table surface, and check `quad status`. The sensor should output approximately zero in all directions. Now move the vehicle slowly along the vehicle's $\underline{1}^B$ direction, being careful not to rotate it: you should have a flow value in `x`, but none (or only a small value) in `y`. Repeat this for the $\underline{2}^B$ direction.

5.1.2 Stiffening the attitude control

As a first step, we will stiffen the attitude controller from the previous lab. We will change the time constant for the yaw rate, and the angles.

```
const float timeConst_rollRate = 0.04f;
const float timeConst_pitchRate = timeConst_rollRate;
const float timeConst_yawRate = 0.1f; //CHANGED!
const float timeConst_rollAngle = 0.12f; //CHANGED!
const float timeConst_pitchAngle = timeConst_rollAngle;
const float timeConst_yawAngle = 0.2f; //CHANGED!
```

5.1.3 Debug outputs over telemetry

You will rely on the telemetry to evaluate the system performance, and understand why things are going wrong. For the outputs 0,1, and 2 we want to send the angle estimates. Add the below, near the end of your MainLoop()

```
outVals.telemetryOutputs_plusMinus100[0] = estRoll;
outVals.telemetryOutputs_plusMinus100[1] = estPitch;
outVals.telemetryOutputs_plusMinus100[2] = estYaw;
```

Verify that you do not use any of the other `telemetryOutputs_plusMinus100` fields – we will populate them as we continue through the lab.

5.2 Vertical state estimator

We implement the state estimator as derived in the lecture. We will use the range sensor output, δ, and produce an estimate of the vehicle height $\hat{h}$ and vertical velocity $\hat{v}_3$. The lecture's estimator relied on generating a height pseudo-measurement from the range sensor and attitude estimate, as

$$h_{\text{meas}}(k) = \delta(k) \cos \hat{\theta}(k) \cos \hat{\phi}(k) \tag{5.1}$$

where k is the time step, and $\hat{\phi}(k)$, $\hat{\theta}(k)$ are estimates of the roll and pitch angle, respectively, at time k.

We also introduced the (rather ugly) vertical velocity pseudo-measurement:

$$v_{3,\text{meas}}(k) = \frac{h_{\text{meas}}(k) - h_{\text{meas}}(k-1)}{t_\delta(k) - t_\delta(k-1)} \tag{5.2}$$

where $t_\delta(k)$ is the time at which we got the distance measurement $\delta(k)$.

Implementing this estimator is slightly more complicated than the attitude estimator we implemented earlier, since we will not receive distance measurements at 500Hz, but only at a much lower rate. This means that we will run the estimator "predict" step at every cycle, but the "correct" step only occasionally; this will require some additional code. Furthermore, the distance sensor is somewhat "fragile", in the sense that it will return dubious (much too large) measurements occasionally, especially when operating over some surfaces – we will add some code to detect such outliers, and ignore them.

5.2.1 Implementation

We begin by creating state variables for the quantities we'd like to estimate; these variables must be created *outside* of `MainLoop` to keep their values (ideally, put them right after your estimation variables for the angles).

```
//outside of Mainloop()
float estHeight = 0;
float estVelocity_1 = 0;
float estVelocity_2 = 0;
float estVelocity_3 = 0;
```

We will also create some variables for storing the last height measurement, these are needed to create the velocity pseudo-measurement:

```
//outside of Mainloop()
float lastHeightMeas_meas = 0;
float lastHeightMeas_time = 0;
```

Finally, we add the estimates to the "debug outputs", so that we can analyze what the estimators did after a flight from the log file:

```
//in your main loop, near the end:
outVals.telemetryOutputs_plusMinus100[3] = estVelocity_1;
outVals.telemetryOutputs_plusMinus100[4] = estVelocity_2;
outVals.telemetryOutputs_plusMinus100[5] = estVelocity_3;
outVals.telemetryOutputs_plusMinus100[6] = estHeight;
```

We are now ready to implement the estimator. In your main loop, after the estimator for the roll, pitch, and yaw angles, implement the prediction step for the height estimator:

```
//In MainLoop():
//height estimator:
//prediction step:
estHeight = estHeight + estVelocity_3 * dt;
estVelocity_3 = estVelocity_3 + 0 * dt;  //assume constant (!)
```

Next we do the measurement update. Note that the measurement correction is hidden behind two `if` statements: we only update if we have a new measurement, and if that measurement is reasonable.

```
//correction step, directly after the prediction step:
float const mixHeight = 0.3f;
if (in.heightSensor.updated) {
  //check that the measurement is reasonable
  if (in.heightSensor.value < 5.0f) {
    float hMeas = in.heightSensor.value * cosf(estRoll) * cosf(estPitch);
    estHeight = (1 - mixHeight) * estHeight + mixHeight * hMeas;

    float v3Meas = (hMeas - lastHeightMeas_meas)
        / (in.currentTime - lastHeightMeas_time);

    estVelocity_3 = (1 - mixHeight) * estVelocity_3 + mixHeight * v3Meas;
    //store this measurement for the next velocity update
    lastHeightMeas_meas = hMeas;
    lastHeightMeas_time = in.currentTime;
  }
}
```

Read through this code carefully, to make sure you understand what's happening. Note that we chose to use a mixing value of 0.3 for the height estimator. Note also that we use `cosf` to compute the cosine of the angles – this is to ensure that we get the `float` version of cosine, rather than the `double` (which would be substantially slower on the microcontroller). Eclipse may underline `cosf` with a red line – you may ignore this warning, the code should compile OK.

Complete Deliverable 2 to confirm that the estimator works as expected.

5.3 Horizontal state estimator

The horizontal state estimator is to estimate the vehicle's velocity in the $\underline{1^E}$ and $\underline{2^E}$ directions, using the optical flow sensor. Recall the model for the optical flow outputs, (σ_1, σ_2):

$$\begin{bmatrix}\sigma_1\\ \sigma_2\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\left(-\frac{1}{\left\|\underline{s_{FS}}\right\|}\left[\underline{v_S^E}\right]^S + \left[\underline{\Omega^{BE}}\right]^S\left[\underline{3^S}\right]^S\right) \tag{5.3}$$

In the class we expanded this, and came up with the two velocity pseudo-measurements, that rely on the optical flow (σ), rate gyroscope (γ), and range sensor (δ) measurements:

$$v_{1,\text{meas}} = \left(-\sigma_1(k) + \gamma_2(k)\right)\delta(k) \tag{5.4}$$
$$v_{2,\text{meas}} = \left(-\sigma_2(k) - \gamma_1(k)\right)\delta(k) \tag{5.5}$$

The estimator we create will be very simple: for the prediction step, we will assume that the velocity is simply constant:

```
//prediction
//(just assume velocity is constant):
estVelocity_1 = estVelocity_1 + 0 * dt;
estVelocity_2 = estVelocity_2 + 0 * dt;
```

The correction step will use the velocity pseudo-measurements. As with the range sensor, the optical flow does not give us a new measurement each cycle, so we implement some additional logic to only perform a correction if we have a new measurement. Note that we flip the sign of the measurement – this is because the sensor uses a different convention than we used in the notes.

```
//correction step:
float const mixHorizVel = 0.1f;
if (in.opticalFlowSensor.updated) {
  float sigma_1 = −in.opticalFlowSensor.value_x;
  float sigma_2 = −in.opticalFlowSensor.value_y;

  float div = (cosf(estRoll) * cosf(estPitch));
  if (div > 0.5f) {
    float deltaPredict = estHeight / div; //this is delta in the equation

    float v1Meas = (−sigma_1 + in.imuMeasurement.rateGyro.y) * deltaPredict;
    float v2Meas = (−sigma_2 − in.imuMeasurement.rateGyro.x) * deltaPredict;

    estVelocity_1 = (1 − mixHorizVel) * estVelocity_1 + mixHorizVel * v1Meas;
    estVelocity_2 = (1 − mixHorizVel) * estVelocity_2 + mixHorizVel * v2Meas;
  }
}
```

Again, carefully read through this code, and make sure that you understand it. Notice that we chose a mixing constant of 0.1. Also note that we compute a *predicted* range measurement $\delta(k)$ (`deltaPredict`), rather than use it directly from the range sensor. This is for two reasons: we know the range sensor occasionally returns very noisy values, which would corrupt our velocity estimate. Secondly, the range sensor measurement might be quite “old” when we have an optical flow measurement – now we have the advantage of using the prediction as well.

Complete Deliverable 3 to confirm that the estimator works as expected.

5.4 Horizontal control

In the previous lab, we implemented an attitude control scheme, which allowed us to control the vehicle roll, pitch and yaw angles. We will now extend this with velocity control – the resulting control structure is a cascaded controller for a triple integrator.

The control structure is simple: we compute a desired horizontal acceleration, to oppose our current velocity. From this acceleration we can compute the pitch and roll angles. Specifically, we set:

$$a_{1,\text{des}} := -\frac{1}{\tau_v} v_1 \tag{5.6}$$

$$a_{2,\text{des}} := -\frac{1}{\tau_v} v_2 \tag{5.7}$$

and then

$$\phi_{\text{des}} := -\frac{1}{\| \underline{g} \|} a_{2,\text{des}} \tag{5.8}$$

$$\theta_{\text{des}} := \frac{1}{\| \underline{g} \|} a_{1,\text{des}} \tag{5.9}$$

Note the signs, and that we need a_2 for ϕ.

Implement this controller. First, we define the time constant; place this code where you defined the time constants for the attitude control:

```
//Near other time constant definitions:
const float timeConst_horizVel = 2.0f;
```

Note that we choose $\tau_v = 2\text{s}$.

We next compute the desired acceleration components. This should be done after the estimation is complete, but before the attitude control:

```
//In MainLoop, in control code:
float desAcc1 = -(1 / timeConst_horizVel) * estVelocity_1;
float desAcc2 = -(1 / timeConst_horizVel) * estVelocity_2;
```

Finally, we compute the desired angles from this, as below. Note that you have already defined these angles, in your code from the previous lab (where we simply set all angles to zero).

```
float desRoll = -desAcc2 / gravity;
float desPitch = desAcc1 / gravity;
float desYaw = 0;
```

Add the desired roll and pitch angles to the debug outputs:

```
outVals.telemetryOutputs_plusMinus100[7] = desRoll;
outVals.telemetryOutputs_plusMinus100[8] = desPitch;
```

5.5 Vertical control

For the vertical control, we recall that the quadcopter dynamics decoupled into four separate subsystems, with one system consisting of the states height s_3 and vertical velocity v_3, while the input was the change in total thrust normalized by mass $\Delta c_\Sigma / m^B = a_3$. This is a double integrator, which we control by choosing an input that makes the system behave like a damped second order system, with damping ratio ζ and natural frequency ω_n:

$$a_3 = -2\zeta\omega_n v_3 - \omega_n^2 \left(s_3 - s_{3,\text{des}}\right) \tag{5.10}$$

where $s_{3,\text{des}}$ is the desired height. We will choose a damping ratio of $\zeta = 0.7$, and natural frequency $\omega_n = 2\text{rad}\,\text{s}^{-1}$:

```
//Before MainLoop(), near where you defined the
// time constants for the attitude control
const float natFreq_height = 2.0f;
const float dampingRatio_height = 0.7f;
```

We will initially set the desired height to $s_{3,\text{des}} = 0.5\text{m}$, which is chosen low on purpose: we want to ensure that the range sensor gets good measurements, and that the vehicle is able to survive a crash.

```
//In MainLoop(), after all the estimation code:
const float desHeight = 0.5f;
const float desAcc3 = -2 * dampingRatio_height * natFreq_height
    * estVelocity_3
    - natFreq_height * natFreq_height * (estHeight - desHeight);
```

We use this to compute the total thrust to be commanded:

$$\frac{c_\Sigma}{m^B} = \frac{1}{\cos\hat{\theta}\cos\hat{\phi}}\left(\|\underline{g}\| + a_3\right) \tag{5.11}$$

Note that we add the nonlinear correction term, where we divide by the cosine of the angles. This ensures that we achieve our target vertical acceleration even if the vehicle is “far” from hover. This may be implemented as below, where we modify the `desNormalizedAcceleration` variable that you created in lab 4.

```
//note: this is the variable that you already introduced for
//   the previous lab. Note that gravity=+9.81m/s**2
float desNormalizedAcceleration = (gravity
    + desAcc3) / (cosf(estRoll) * cosf(estPitch));
```

Note that, to first order, dividing by the angles doesn't matter, since $\cos\hat{\theta} \approx 1$. However, the correction is easy, so we add it (and, it turns out to make an appreciable difference).

Add the desired normalized thrust to the debug outputs:

```
outVals.telemetryOutputs_plusMinus100[9] = desNormalizedAcceleration;
```

5.6 Flight test

This is related to Deliverable 4.

You have now implemented all the components so that the vehicle will hover. Place the vehicle inside the flight safety net, on the ground, with its power off. Now turn the vehicle on, and step away from it. *Note: don't turn the vehicle on, and then carry it in: the yaw estimate will then potentially be far from zero, which will cause unexpected behaviour after take off.* Making sure you are safely out of the way, let the vehicle take off from the ground, and observe its flight. Remember, to get the vehicle flying, you need to first arm it (hold down the start button), and then when you hold down the red button the motors are allowed to run. Ideally, you should see the vehicle take off, and slowly drift to the side. If it does this, you are almost done, and can move onto the next step, to improve the system performance.

If it does not fly well, you will need to debug the vehicle.

5.6.1 Debugging poor flight performance

If the performance is worse than you expect, you can debug it using data that you gathered in flight. Do the following: Make sure that you start the vehicle on the ground, and without disturbing the rate gyroscope calibration. Start the `RadioAndJoystick` program (so that you have a short, clean log file for this flight), and then let the vehicle take off. Immediately after the end of the flight, kill `RadioAndJoystick` (so that you have a short, clean log file for this flight). Copy the log file into its own, separate folder, and create a separate text file in that folder. In this text file, write down what you observed in the flight (e.g. the vehicle tilted to the left, and then crashed).

Center of mass error

If the vehicle always accelerates off in one direction on takeoff, you may have a center of mass error. In this case, your center of thrust goes through the vehicle's geometric center, and if the center of mass does not coincide with the geometric center the vehicle will experience a disturbance torque, which will cause the vehicle to accelerate off to one side. Most likely, this is due to the battery being too far forward/backward. You may adjust the position of the battery by hand, until the vehicle no longer accelerates off to the side.

Data-based debugging

If the vehicle is not always accelerating off in the same direction, you'll need to look at the data to understand the behavior. Open your log file, with your favorite tool (Python, Spreadsheets, or Matlab), and plot all of the state estimates over time. Also plot the IMU measurements. From the IMU measurements, identify when the vehicle took off, and when it crashed. Now check the state estimates for this period – do you notice any problems in the estimation?

If you think the estimates look reasonable, check the controller performance. Plot the vehicle's desired and estimated roll angle on the same figure, and check whether these agree reasonably well. Repeat this for the vehicle pitch and yaw.

Plot the vehicle's desired angular velocity, and the measured angular velocity from the rate gyroscope. Do these correspond well?

Note: you have two additional telemetry debug fields (numbers `10` and `11` of the values in `telemetryOutputs_plusMinus100`) to transmit additional data to your computer.

5.7 Improving the performance

Once you have the vehicle flying reasonably, you can attempt to improve the performance. Since the vehicle has no ability to measure its horizontal position, it *must* eventually drift out of the allowable flight space (and into the safety netting). The better your control and estimation works, the longer this will take.

If you have carefully followed all of the labs, characterized your system well, etc., your vehicle should do an OK job at hovering. You can, however, improve its performance. Some things you could try are listed below (note, these are not exhaustive). Also note, that these are not all mandatory – you may do some of these, or you may do modifications other than what is suggested.

NOTE: before you try anything wild, make a copy of your code, label it, so that it's easy for you to go back. This is important, because it is much easier to make something *worse* than it is to make it better.

- Improve your model by: reducing the center of mass error or compensating explicitly for a known center of mass error.
- Improve the controllers, by analyzing the chosen time constants and their interactions.
- Improve the estimators, by modifying the various mixing constants to better trade off noise and uncertainty.
- Replace the cascaded controllers with controller gains computed for the full system (using, e.g., pole placement, or LQR).
- Implement a position estimator, that integrates the velocity estimates; and perform feedback on this (note that, even if you did this perfectly, this will also, eventually, drift off due to noise).
- Improve the estimators: especially by improving the predictive step for the estimators (recall the very simple velocity prediction we make by default, where the velocity is held constant).

Clearly note what you did, and what its effects were, for Deliverable 5. You will be scored on scientific methodology: even if an idea doesn't help, it is useful to document:

1. what you tried,

2. what the expected effect was,
3. what the observed effect was,
4. (if applicable) potential reasons for a discrepancy between expected and observed,
5. final action (keep this, or don't keep this?).

5.8 Deliverables

You must submit electronically, via email, a PDF report describing what you did in this lab exercise. The report should be typed up, e.g. using Latex or a word processor; it should be complete, i.e. each deliverable should be a separate section, which is self-contained. Comment on your results: what was surprising, what is unexplained, etc.

Besides being technically correct, your report must be clear and concise. We will apply a "signal-to-noise ratio" penalty, if we feel that your report is not clearly written, neat, and concise. Any code snippets that you submit must be neat and legible, with comments describing the code.

There may be some homework questions posted with the laboratory exercises. You should complete these, and append their solutions to the report. You do not need to type up the solutions to the homework questions, you can simply scan them in and append them to your report.

Lab reports are due electronically at 14:00 on Wednesday 29 November *for all groups.*

Submit the reports with homework as a single PDF for your group, via email, making sure that you complete this checklist:

- Attach: single PDF (which contains group ID, and all member names)
- To: `me136_gsi@berkeley.edu`
- Subject (replace "XX" with your group ID): `[ME136 lab report] Group XX`
- CC: Put all group members in CC.

Late submissions: you will lose 20% if your submission is late by less than eight hours. For each additional hour, you lose an additional 10%.

For your report:

1. Describe your experimental set-up (with photographs as necessary). [5%]
2. For your height estimator, perform the following experiment to verify its performance. [10%]
   (a) Perform the following steps:
      i. Place the vehicle on the floor, and turn it on. Make sure there are no obstacles near the vehicle. Now start `RadioAndJoystick`, so you have a clean log file.
      ii. Being careful to keep the vehicle horizontal and not obstruct the distance sensor, pick it up, and lift it to approximately 0.5m above the surface.
      iii. Hold the vehicle there for approximately 5s.
      iv. Rotate the vehicle through a pitch and/or roll angle of approximately ± 30 degrees, and return it to a horizontal attitude (while keeping the height constant).
      v. Smoothly return the vehicle to the ground, keeping it approximately horizontal.

   (b) Make three plots, against time, of the following below quantities. On the graph, indicate where each phase of the experiment starts.
      - The roll and pitch estimates
      - The vertical velocity
      - The height estimate, and the measurements from the distance sensor.

   (c) Comment on the estimator performance.
3. For your horizontal estimator, perform the following experiment to verify its performance. [10%]
   (a) Perform the following steps:
      i. Place the vehicle on the floor, and turn it on. Make sure there are no obstacles near the vehicle. Now start `RadioAndJoystick`, so you have a clean log file.
      ii. Being careful to keep the vehicle horizontal and not obstruct the distance sensor, pick it up, and lift it to approximately 0.5m above the surface.
      iii. Hold the vehicle there for approximately 2s.
      iv. Keeping the vehicle orientation constant, walk forward (along $\underline{1}^B$) approximately 1, and then stop for 2s.
      v. Keeping the vehicle orientation constant, walk left (along $\underline{2}^B$) approximately 1, and then stop for 2s.

    vi. Rotate the vehicle through a pitch and/or roll angle of approximately ± 30 degrees, and return it to a horizontal attitude (while keeping the height constant).
    vii. Smoothly return the vehicle to the ground, keeping it approximately horizontal.

  (b) Make 4 plots, against time, of the following below quantities. On the graph, indicate where each phase of the experiment starts.
    - The rate gyroscope measurements
    - The output from the flow sensor
    - The estimated horizontal velocity
    - The height estimate

  (c) Comment on the estimator performance.

4. For your closed-loop hover flight [10%]

  (a) Perform the following steps:
    i. Place the vehicle on the floor, in the center of the flight space, and turn it on.
    ii. Let the vehicle take off, and fly as long as possible.

  (b) Plot, against time, the following below quantities.
    - The estimated angles, as well as the desired angles
    - The estimated vehicle velocity
    - The estimated height and the desired height
    - The total motor force (you'll have to implement the functions to transform the commands you receive over telemetry to [N], as you did in the C++ code).

  (c) Comment on the flight performance.

  (d) Clearly explain any changes you made to the "standard" control layout, as presented in the labs so far, to improve performance.

5. For your closed-loop hover flight analysis and modifications, do the below. Note that your hover performance will be evaluated experimentally (during the competition) in Lab 6, the description for which is available online. Here, we want to see your reasoning behind the modifications you make, and how you tested their efficacy. [15%]

  (a) Explain what modifications you made to the standard control/estimation architecture, and what its effects were.

  (b) Comment on the performance you could achieve: what do you believe is the main limitation that you face, to improving the performance?

   (c) Substantiate the above with clear experimental data.

6. Provide the full listing of your `MainLoop` function, and provide any other functions you introduced. Make sure that everything is well commented.

7. Complete the Lab 5 homework questions, posted separately on the class webpage [50%]