Engineering 100 Sections 400 Lab 4: Introduction to Autonomous Flight Control

In this lab you will begin work on enabling your quadcopter to traverse an obstacle course using a sensor enclosure containing four horizontal (side-mounted) infrared (LiDAR) sensors[1] and a small Arduino microcontroller.[2] The microcontroller is then used to interface with the sensors, and send their measured distances to the BealgeBone. You will learn how to create your own flight mode for controlling the movement of the quadcopter. Your task for the final project is to move the quadcopter to clear obstacles detected by the horizontal sensors. Therefore, after creating a new flight mode you will implement and test simple PID controllers for stabilizing the quadcopter within a certain distance from a wall, which you can use to make the quadcopter move along walls to avoid obstacles.

There are no pre-lab or post-lab assignments. You will have two weeks to complete the lab but you should try to finish as soon as practical so you can transition to the course project.

1 Creating a new flight mode

This section shows you how to add a new flight mode to ArduPilot, and how to fly using your new mode in Mission Planner. This allows you to define your own flight controller for adding functionality to the quadcopter, such as the ability to traverse an obstacle course.

1.1 Downloading ArduPilot's code

To add your new autonomous flight mode, you need to download and alter ArduPilot's source code, then compile and upload the modified code to the quadcopter. To do this, you will need to use a specific version of Ubuntu Linux which has all the necessary dependencies and tools needed to cross-compile the ArduPilot code for the BeagleBone. To run this version of Linux we have created a virutal machine that can be run from nearly any operating system. Please download the .ova file from Canvas which contains the image for this virtual machine. This is a large file and if you are having trouble downloading it you can request a USB stick from the lab staff with this already loaded.. We recommend that at least two team members go through this process of setting up the Linux environment on their laptops because this is the format in which you will be editing and compiling the code for your final project.

There are several options for running Ubuntu virtual machines in Windows or MacOS, but the virtual machine we have provided is configured for an application called Virtual Box. Virtual Box is a virtual machine manager, or 'hypervisor.' It is free open-source software maintained by Oracle Corp. and is available for Windows, MacOS, and most Linux distributions. On the VirtualBox downloads page, download the Oracle VM VirtualBox Extension Pack as well as the regular VirtualBox download for whichever operating system you are working on. Once it finishes downloading,

[1] `https://www.pololu.com/product/3415`
[2] `https://www.pololu.com/product/3101`

launch it and select “file → import appliance” and point it to the .ova file you downloaded from Canvas. You can then run the virtual machine by selecting it and clicking “start”.

As a starting point for your project, you will use a fork of the ArduPilot code hosted on the University’s GitLab servers[3]. This repository contains some modification to the default code in order to help you start work on the project. You will need to clone this git repository by opening the Linux terminal and typing the command, “git clone https://gitlab.eecs.umich.edu/engr100-400/ardupilot.git”. You may need to manually type this in, as copy and paste does not always paste certain characters properly. If you have not used GitLab at the University of Michigan yet, your credentials may be refused when you attempt a command-line download. In this case, go the the UMich GitLab web homepage and sign in (https://gitlab.eecs.umich.edu/users/sign_in). That should activate your account.

Checkpoint 1. *(5 points) Download and extract ArduPilot’s source code on your machine. Ask your instructor to double check your setup before continuing.*

1.2 Adding a flight mode in ArduPilot

Note that the following changes have already been made on the modified firmware hosted on the university’s GitLab server. Read this section to better understand how a flight mode works, and parts of the code that you need to implement for your projects.

Please start in the `ArduCopter` directory of ArduPilot’s source code. This directory contains the code required for setting up and controlling the quadcopter. ArduPilot has 18 built-in flight modes, defined in the `defines.h` header file. So far, you have mainly used AltHold (for keeping the quadcopter at a certain altitude) and Land modes. To define a new flight mode, we have added an extra value to the `control_mode_t` enumeration in this file.

```
// Auto Pilot Modes enumeration
enum control_mode_t {
    STABILIZE =     0,  // manual airframe angle with manual throttle
    ACRO =          1,  // manual body-frame angular rate with manual throttle
    ALT_HOLD =      2,  // manual airframe angle with automatic throttle
    ...
    GUIDED_NOGPS = 20,  // guided mode but only accepts attitude and altitude
    AUTONOMOUS =  100,  // autonomous flight mode for ENGR100-Drone
};
```

The last (`AUTONOMOUS`) entry in the above enumeration defines a new flight mode which will be used for implemeneting the controller for traversing the obstacle course.

Next we define the functions for initializing and running the Autonomous flight mode by adding the following function definitions to the `Copter.h` header file.

```
// autonomous flight mode for ENGR100-Drone
bool autonomous_init(bool ignore_checks);
void autonomous_run();
bool autonomous_controller(float &target_climb_rate,
                           float &target_roll, float &target_pitch,
                           float &target_yaw_rate);
```

The above three functions are used to define the new flight mode, as described below.

[3]https://gitlab.eecs.umich.edu/engr100-400/ardupilot.git

- `autonomous_init`: This function is used for initializing the controller once the user switches to this flight mode. You generally do not need to alter this function.
- `autonomous_run`: This function defines the main loop for the flight mode; it is executed at a frequency of 400 Hz. It contains safety checks, the take-off procedure, and code for controlling the quacopter's movement (by setting motor speeds to achieve the desired throttle, pitch, roll, and yaw angles) during flight. You generally do not need to alter this function.
- `autonomous_controller`: Called from inside `autonomous_run`, this function is used for setting the target vertical climb rate, roll, pitch, and yaw rate (angular velocity for yaw) during flight. The `autonomous_run` function then converts these desired values to individual motor speeds, which will then be converted to PWM signals and fed to the ESCs for controlling motors. You can implement the algorithm for your final project inside this function.

As a starting point for the `autonomous_init` and `autonomous_run` functions, we use the implementations for `althold_init` and `althold_update` from `control_althold.cpp` and put them in a sketch named `control_autonomous.cpp`. We then add the `autonomous_controller` function for controlling the quadcopter's movement during flight.

We also need to tell ArduPilot to run these function when the user switches to this flight mode. This can be done by adding a case to the `switch` statement in the following functions under `flight_mode.cpp`: `set_mode` (switching to a new flight mode), `update_flight_mode` (for running the main loop of the new flight mode), and `print_flight_mode` (for printing the name of the flight mode). As stated above, these changes have already been made for you.

The new flight mode is now a duplicate of AltHold. AltHold takes the target roll, pitch, yaw rate, and vertical climb rate of the quadcopter from the pilot. To make this an automated flight controller, you must generate these values in software. As described above, this is done using the `autonomous_controller` function. This function is repeatedly called inside the `autonomous_run` function (i.e., at a frequency of 400 Hz), assuming that the motors are armed and spinning, and the quadcopter is not in the process of taking off.

```
case AltHold_Flying:
    // compute the target climb rate, roll, pitch and yaw rate
    // land if autonomous_controller returns false
    if (! autonomous_controller(target_climb_rate, target_roll, target_pitch,
    target_yaw_rate)) {
        // switch to land mode
        set_mode(LAND, MODE_REASON_MISSION_END);
        break;
    }
```

The above code sets the target climb rate (in cm/s), lean angles (in centi-degrees), and rotation speed (in centi-degrees per second) for controlling the quadcopter's motion, and provides them to the main loop. These values are then converted to individual motors speeds through ArduPilot's internal controllers. The desired roll and pitch angles are further adjusted by ArduPilot's obstacle avoidance to prevent crashes, and that obstacle avoidance has a higher priority than your code.

The `autonomous_controller` function also returns a boolean value. A *false* value indicates the end of the mission, immediately putting the quadcopter in "Land" mode and disarming the motors. You can use this functionality to land your quadcopter once you have reached the designated landing zone in the obstacle course.

In summary, you will need to implement your controller for the final project inside the `autonomous_controller` function, inside the `ArduCopter/control_autonomous.cpp` file under ArduPilot's source code. This function is executed at a frequency of 400 Hz (400 times per second), and generates the following parameters for controlling the quadcopter's movement.

- `target_climb_rate`: The desired climb rate in centimeters per second. Unless you want to control the vertical movement of your quadcopter for your team-defined innovation, set this value to zero. Note that the quadcopter automatically climbs to the altitude specified by the `PILOT_TKOFF_ALT` parameter in Mission Planner after take-off, and maintains that altitude until the `autonomous_controller` function returns false, telling the quadcopter to land.
- `target_pitch`: The desired pitch angle in centi-degrees for moving forward/backward. Note that a negative value causes the quadcopter to lean/move forward.
- `target_roll`: The desired roll angle in centi-degrees for moving sideways. Note that a positive value causes the quadcopter to lean/move right.
- `target_yaw_rate`: The desired yaw rate (i.e., angular velocity for yaw), in centi-degrees per second. You generally want to set this value to zero to keep constant heading, and move the quadcopter by changing `target_pitch` and `target_roll`.

Note that you have access to the following variables inside `control_autonomous.cpp`:

- `rangefinder_alt`: The altitude, in meters, from the downward facing sonar sensor.
- `dist_forward`: Distance, in meters, read by the forward horizontal sensor.
- `dist_right`: Distance, in meters, read by the right horizontal sensor.
- `dist_backward`: Distance, in meters, read by the backward horizontal sensor.
- `dist_left`: Distance, in meters, read by the left horizontal sensor.

Checkpoint 2. *(5 points) Read the implementation of the the autonomous_ controller function and discuss it with your instructor. Make sure you understand each piece of code in the above function, and how they can be used to automatically controll the quacopter.*

1.3 Compiling ArduPilot

Now that you understand the new flight mode, you need to compile and upload ArduPilot's code to your quadcopter. Open a Linux terminal, and configure the source code by typing the following commands into the terminal. You may need to run the first command (`./configure`) twice if the terminal output asks you to do so. This first compile will take a few minutes, but subsequent builds after editing your code will only take a few seconds. This is because it only needs to recompile the part that you editing, while leaving most of the rest of the compiled code the same.

```
cd ardupilot
./configure
./compile
```

Connect to the BeagleBone using WiFi, and then **stop the ArduPilot service by pressing the `PAU` button on the BeagleBone board**[4]. Note that the button may be hard to press due to the mounted sensor enclosure. Be careful not to accidently press the `RST` or `POW` buttons, as this will restart (turn off) the main board.

You will now copy the compiled executable and upload it onto the board, by typing the following command in a terminal (the password is `engr100`).

[4]This is in bold because students very often miss it.

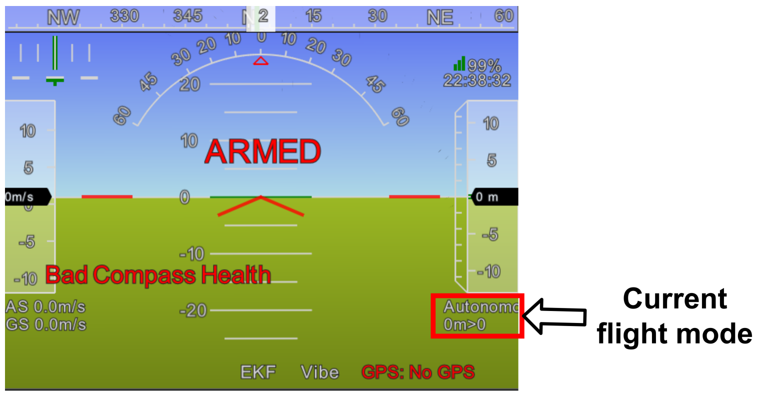

Figure 1: Finding the current flight mode in Mission Planner.

```
./upload
```

Sometimes the upload process will fail. If this occurs, in Virtual Box open files and go to /home/engr100400/.ssh. There should be a file called "Known Hosts". Empty this file and retry the upload.

Once the upload is done, press the `MOD` button to restart the ArduPilot service. Congratulations! You have succesfully compiled and uploaded ArduPilot to your quadcopter.

Note that you need to recompile and upload the compiled binary to the quadcopter (by typing the above two commands) every time you make changes to your code.

Checkpoint 3. *(5 points) Ask your instructor to check that you have successfully compiled and uploaded ArduPilot's code to the quadcopter.*

1.4 Setting up Mission Planner

At this point if you have been using any other version of Mission Planner than 1.3.58 or if you accidentally allowed your original install of 1.3.58 to upgrade on start-up, you need to go back and re-install that version from the Canvas file repository because these additional files may not be compatible with other versions.

After compiling the ArduPilot binary and uploading it to the quadcopter, you also need to configure Mission Planner to recognize this new flight mode. This can be done by modifying the two XML files below, replacing them those provided with the lab handout.

- `C:\Program Files\Mission Planner\ParameterMetaDataBackup.xml`, or if the directory does not exist `C:\Program Files (x86)\Mission Planner\ParameterMetaDataBackup.xml`
- `Documents\Mission Planner\ParameterMetaData.xml`.

Now open Mission Planner, and go to `Initial Setup` → `Mandatory Hardware` → `Flight Modes`. Confirm that you can see the `Autonomous` entry at the end of the drop-down lists. Choose `Autonomous` for all items (Flight Mode 1 through 6), and click `Save Modes`. You can test the new flight mode by using the `Autonomous.py` script included in the lab files. This script is the same as the script that you have used for flying the quadcopter so far, except that it uses the Autonomous flight mode instead of AltHold. Test the script without propellers. Mission Planner displays the current flight mode in the HUD area under the `Flight Data` screen, as illustrated in Figure 1; make sure that it shows *Autonomous* after clicking takeoff.

Checkpoint 4. *(5 points) Demo the Autonomous flight mode to your instructor.*

2 Implementing your autonomous controller

In this section, we will walk you through a number of tools that can help with implementing the algorithm for your project.

2.1 Sending logging messages to Mission Planner

We will first show you how to send logging messages from ArduPilot to Mission Planner, in order to keep track of your code when flying. Add the following piece of code at the end of the `autonomous_controller` function, before the return statement.

```
// send logging messages to Mission Planner once every second
static int counter = 0;
if (counter++ > 400) {
    gcs_send_text(MAV_SEVERITY_INFO, "Autnomous flight mode for X");
    counter = 0;
}
```

In the above code, the `gcs_send_text` function sends a piece of text ("Autnomous flight mode for X") to Mission Planner. Replace "X" with the name of your group. Note that since your code is cycling at 400 Hz, you should avoid sending a message at every cycle which might starve other tasks of processor time. Therefore in the above code, we have used a static variable to only send messages once every 400 cycles (i.e., once every second); you can change the number 400 to send messages more or less frequently.

Recompile and upload your code to the quadcopter, and make sure that you can see the logging messages by going to `Flight Data` → `Messages` after clicking takeoff.

Checkpoint 5. *(5 points) Ask your instructor to check your logging messages in ArduPilot.*

This method provides a powerful tool for debugging your code. As an examples you can further add version numbers (i.e., "Autonomous flight mode for X, version Y") to make sure that the quadcopter is running your most recent code. You also add more logging messages inside the `autonomous_controller` function to track/debug different parts of your code. We encourage you to make use of these messages as you work on your project.

2.2 Implementing PID controllers for roll and pitch

Now that you have learned how to compile and upload ArduPilot to the quadcopter, you need to implement an algorithm for traversing the obstacle course. To help you with this task, we will show you how to implement PID controllers for adjusting the roll and pitch angles of the quadcopter, applying what you learned in lectures for controlling the quadcopter's movement.

You have access to two PID controllers in your code, i.e., for setting the target roll and pitch of the quadcopter. The corresponding gains for these controllers can be tuned using `PITCH_{P,I,D}`, `ROLL_{P,I,D}` from Mission Planner. Additionally, the `PITCH_FILT` and `ROLL_FILT` parameters can specify a cutoff frequency for filtering the input (i.e., error) to the controller; this can help smoothen the response of the controller by filtering high frequency noise present in sensor readings. `PITCH_IMAX` and `ROLL_IMAX` set the maximum integral term that is produced by the controller; you can use these parameters to prevent the integral term from growing too large when the error remains large for a long duration.

Assume that you want the quadcopter to hover at a constant distance from a wall. As an example, you can use such a controller to keep the quadcopter a fixed distance from a wall, while moving left/right to find a path forward. In this example, the error of the controller is the difference between the target distance, and the distance measured by the front sensor. You can provide this error to the PID controller for pitch, and use its output to set the desired pitch angle. In the `autonomous_controller` function remove the following line:

```
target_pitch = 0.0f;
```

and replace it with the following two lines:

```
g.pid_pitch.set_input_filter_all(10*(0.5f) - dist_forward);
target_pitch = 100.0f * g.pid_pitch.get_pid();
```

In the above example we want to keep the quadcopter 0.5 meters from the wall. The first line sets the input (i.e., the error) to the PID controller, and the second line computes the output of the controller (i.e., the target pitch angle). Note that we are subtracting the forward distance from 0.5 meters, since a negative error (i.e., larger distance from the wall) results in setting the pitch to a negative value, which in turn causes the quadcopter to correct the error by moving forward. We are also assuming the output of the controller is the target pitch angle in degrees, we then multiply that number by 100 to compute `target_pitch`, which should be specified in centi-degrees. The distance is also set to 5.0 instead of 0.5 because we are multiplying all the distances by a factor of 10 to improve resolution when distance is represented as an integer.

Compile and upload the above code, and fly the quadcopter to check the performance of the controller. You can start with a value of 1.0 for the proportional gain (`PITCH_P`), setting the integral (`PITCH_I`) and derivative (`PITCH_D`) gains to zero. Further tune the controller by modifying the proportional gain, and adding derivative/integral gains.

You can similary use `g.pid_roll` in your code for setting the target roll angle. As an example, you can use this controller to keep the quadcopter centered between two side walls.

Checkpoint 6. *(5 points) Alter the code to implement a controller for centering the quadcopter between two side walls, while remaining at a constant distance from the front wall. Ask your instructor to check your code before continuing.*

Now you will fly the quadcopter with the above code (you can try using the same gains for the Roll conroller that you used for the pitch controller), and make sure that the quadcopter can remain relatively stable when flying inside an enclosed area.

You will need to turn the `AVOID_DIST_MAX` parameter to its minimum value: 2. This is because the built-in obstacle avoidance overrides the target pitch a. However, for this lab and the final project if this parameter is set too high, then it will constantly be avoiding obstacles and never using your code.

Remember that you should re-upload your Arduino code for the LiDAR. The current code on your drone's Arduino may have a different cutoff frequency (alpha value), or it may have completely different code from another team.

If your drone is consistently taking off in one direction and crashing, you may need to re-calibrate it. You can also test how well it is flying by running in with the previous Alt-Hold script to see if can maintain a stable flight without your edited code. A drone that consistently pulls in one direction running a vanilla 'alt-hold' script may also need ESC or motor calibration. Bring your quadcopter to your lab instructor to check this out.

Checkpoint 7. *(10 points) Demo a flight in an enclosed area with tuned PID parameters, and discuss your parameter choices with your instructor.*

In this exercise, we have shown you how to use feedback control to keep the quadcopter hovering inside a closed area. You can now combine these controllers to navigate inside the obstacle course, e.g., by following a front/side wall while moving sideways/forward to clear an obstacle. You can then implement a number of these maneuvers in sequence to go through the obstacle course. Note, however, that a robust controller has to be able to reliably detect state changes, i.e., when to move left/right/forward. Try to come up with a basic flow chart first, implement and test, and further refine your algorithm based on your observations.

2.3 Additional parameters

To allow you to tune the Autonomous flight mode from Mission Planner, we have added five additional parameters. As an example, you can use these parameters for setting the obstacle detection threshold. You can modify `E100_PARAM1` to `E100_PARAM5` to change these parameters in Mission Planner. The parameters can then be accessed using `g.e100_param1` to `g.e100_param5` inside your code. Begin by using one of these parameters for setting the distance to the front wall.

```
g.pid_pitch.set_input_filter_all(g.e100_param1 - dist_forward);
target_pitch = 100.0f * g.pid_pitch.get_pid();
```

You can now change this distance from Mission Planner by modifying `E100_PARAM1` without the need to recompile your code. We highly encourage you to make use of these parameters in your code, to help you tune your controller.

2.4 Parameter tunning

Note that you generally cannot keep the quadcopter perfectly stable when flying, as noise and delays (e.g., due to the low refresh rate of the horizontal sensors) in the controllers can result in undesirable movements. Your task is to obtain the smoothest response possible by tuning filters, PID controllers, and other parameters, some of which you should have already tuned in previous labs. Speed is desirable but collisions are undesirable. The tunable parameters are summarized below.

- *Take-off controllers*: `POS_Z_P`, `ACCEL_Z_P`, and `ACCEL_Z_I`. Tuned in Lab 2.
- *IMU filters*: `INS_GYRO_FILTER` and `INS_ACCEL_FILTER`. Tuned in Lab 3.
- *Obstacle avoidance*: `AVOID_DIST_MAX` and `AVOID_ANGLE_MAX`. Tuned in Lab 3.
- *Horizontal sensor filters*: `ALPHA` in the Arduino code for the sensor box. Tuned in Lab 3.
- *Pitch controller*: `PITCH_P`, `PITCH_I`, `PITCH_D`, `PITCH_FILT`, and `PITCH_IMAX`.
- *Roll controller*: `ROLL_P`, `ROLL_I`, `ROLL_D`, `ROLL_FILT`, and `ROLL_IMAX`.
- *Additional parameters*: `E100_PARAM1` to `E100_PARAM5`, if being used in your code.

That's it. Good luck to you in the final lab, and the project!