Projects with Pluto

Getting started with your Pluto nano drone

Table of contents

1. Introduction

PlutoX aims at providing an ultimate user experience in terms of flight experience as well as tinkering. These two experiences are must have for innovation. We believe that tinkerers have great ideas. Some of which can even change the world. We at Drona Aviation strive to help such ideas become reality. Our product PlutoX is a great prototyping tool for implementing your ideas.

Tinkering is an interesting hobby and the development tool plays a very important role in making it an engrossing experience. A basic development tool will provide a platform where the tinkerer has to begin from scratch every time and gradually start implementing their ideas, which surely is time consuming. However, if a development tool provides an effortless platform for the tinkerers to directly develop their ideas on, without the need of worrying about the rest of the system, it will definitely be more productive.

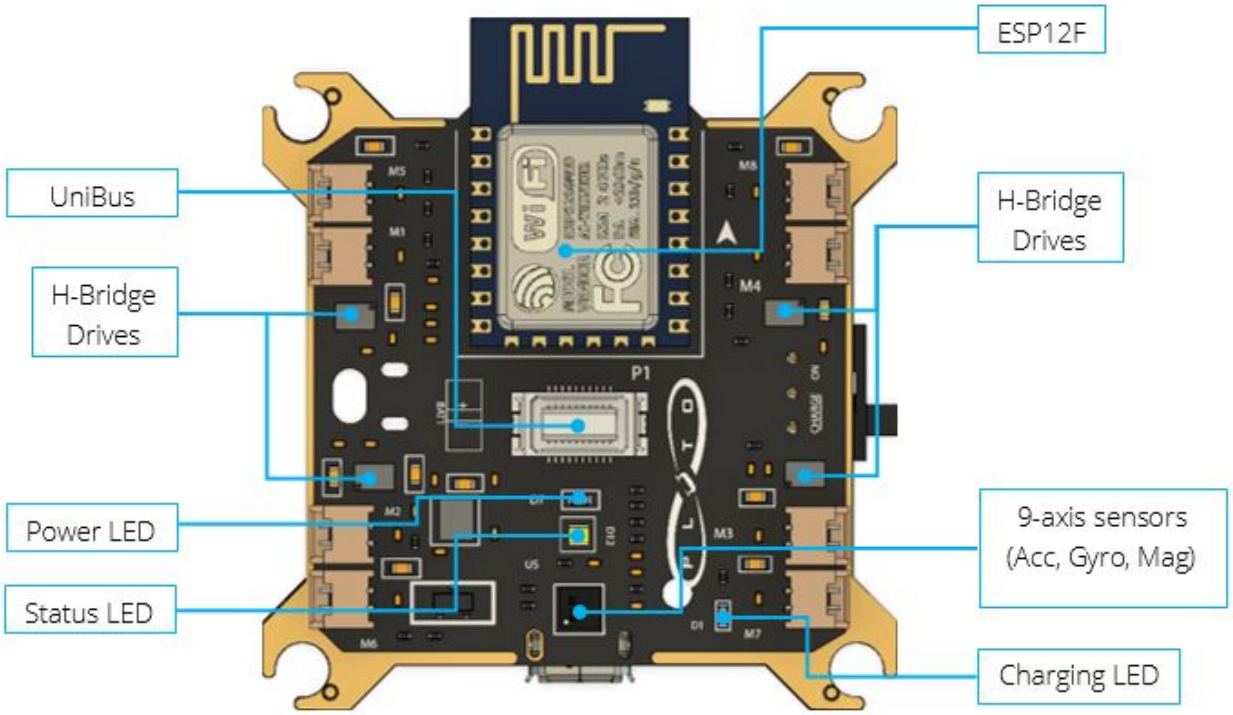
With this perspective, we do not want our tinkerers to waste their energy in figuring out how to get the drone to fly but to focus only on what they want to do and achieve with the drone . For this purpose, we have done all the work required for PlutoX to fly and hence, tinkerers do not need to worry about that. Even without any tinkering, a user can easily fly PlutoX. The entire system has been designed in such an elegant way that the tinkerers can directly start implementing their ideas by coding into the platform provided to them.

2. The Platform

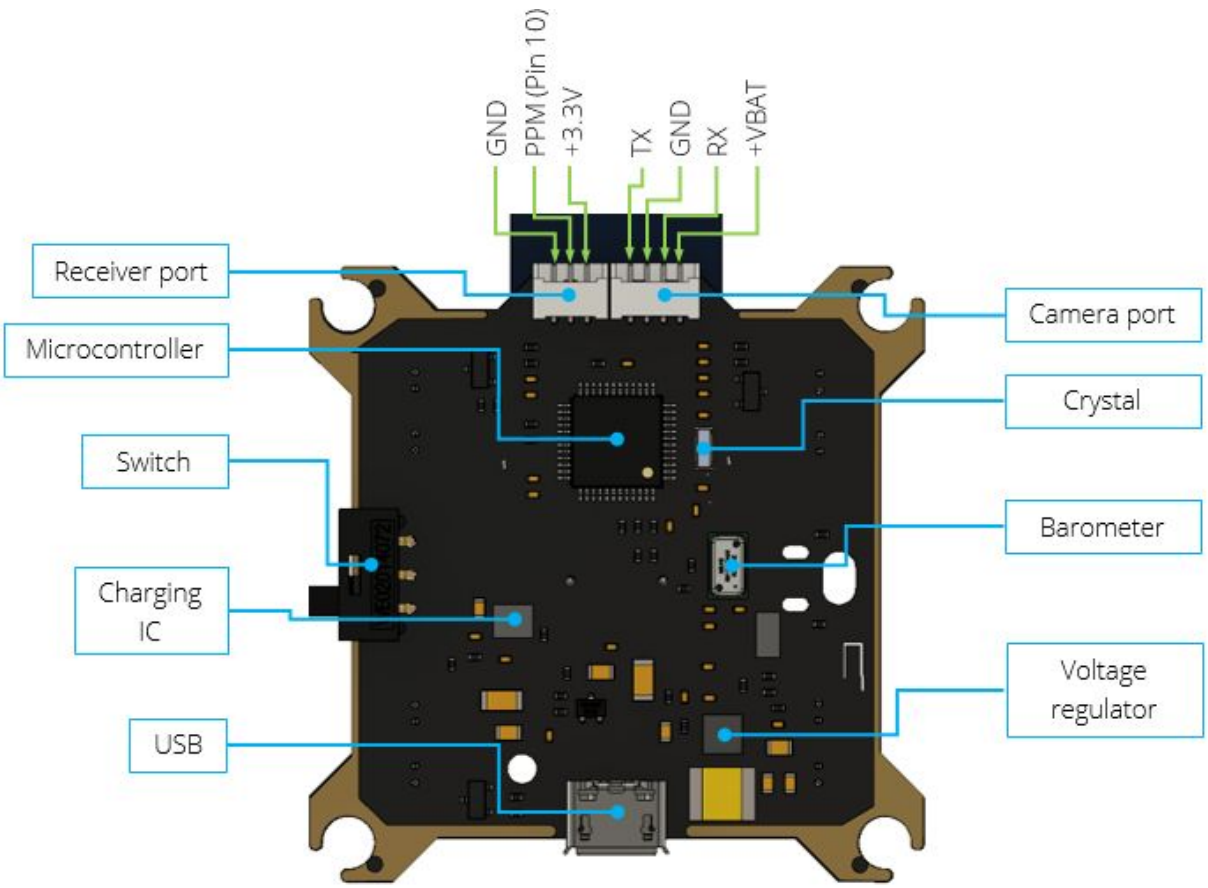
PlutoX, has three essential components: Hardware, Software and Design. All three parts are essential for a seamless flight. The platform allows you to change all three components to suit your needs.

2.1 PrimusX - The hardware

Let us understand the hardware that goes into PlutoX. By hardware, we essentially refer to the electronics. The flight controller of PlutoX is called PrimusX. Primus X comes loaded with many features. Some of the important components are highlighted in the below diagram. These components are explained ahead.



Top side



Bottom Side

Microcontroller: PrimusX has a 32 bit ARM Cortex-M4 processor. It has 256Kb of flash. Approximately 178Kb is available for user code and 78kb is used by drone's firmware.

Wireless module: PrimusX uses ESP12F module for wireless communication. It generates hotspot for connecting your computer and your phone. This helps in wirelessly flashing your code onto PlutoX and also for flying it with your phone using Pluto Controller app.

Inertial Measurement Unit (IMU): PrimusX has a 9-axis IMU with three inbuilt core sensors - Rate-gyro, Accelerometer and Magnetometer. These three sensors are crucial as they provide the information regarding PlutoX's current orientation, velocity and acceleration. PrimusX also comes with a Barometer sensor located at the bottom side. It measures pressure along with the temperature. The working of each of these sensors is explained in detail in their respective projects.

Motor drivers: PrimusX comes with 8 motor drivers M1 to M8. Four of these drivers, M1 - M4 are reversible drivers (H-bridge) and the other four, M5 - M8 are unidirectional drivers. All the motor ports are labelled on the Primus X board.

LEDs: PrimusX contains three sets of LEDs:

1. Power LED: Which indicates whether PlutoX is powered ON or not.
2. Status LEDs: These are the typical RGB LEDs which are useful for indicating status as per the requirements.
3. Charging LED: This turns on when PlutoX is charging. When the charging is finished, the LED dims.

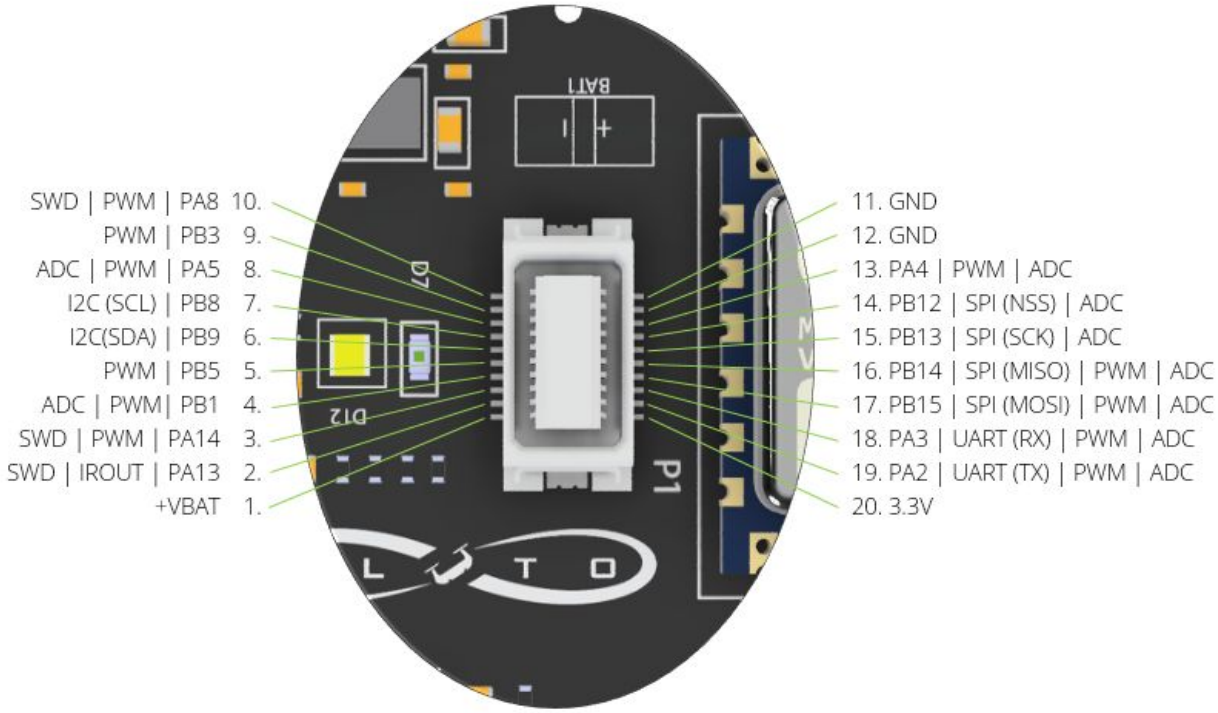
USB port: PrimusX is mounted with a USB port for charging the battery. It can also be used for DFU based flashing.

Switch: At the back side of Primus X is the ON-OFF slide switch.

Camera port: PlutoX comes with a camera module which contains its own WiFi and UART for communication with the microcontroller. The ESP12F also contains its WiFi and communicates with the microcontroller by UART. This could cause trouble in communication with two channels communicating simultaneously. When PlutoX is switched ON, the ESP12F therefore checks for any connection on the camera port. If the port is not connected with the camera module, the ESP module works as usual. If the camera port is connected, the ESP module detects it and shuts itself down, with the camera module taking over all the tasks carried out by the ESP12F module.

RC/LED port: This port is under development. An external RC can be connected to PlutoX. This port can be used to connect the receiver of the controller. One way communication is possible through this port. This port can also be used to connect addressable LEDs.

Unibus: PlutoX is all about tinkering. For this purpose, PrimusX comes with a 20 pin unibus for integrating external hardware like sensors.

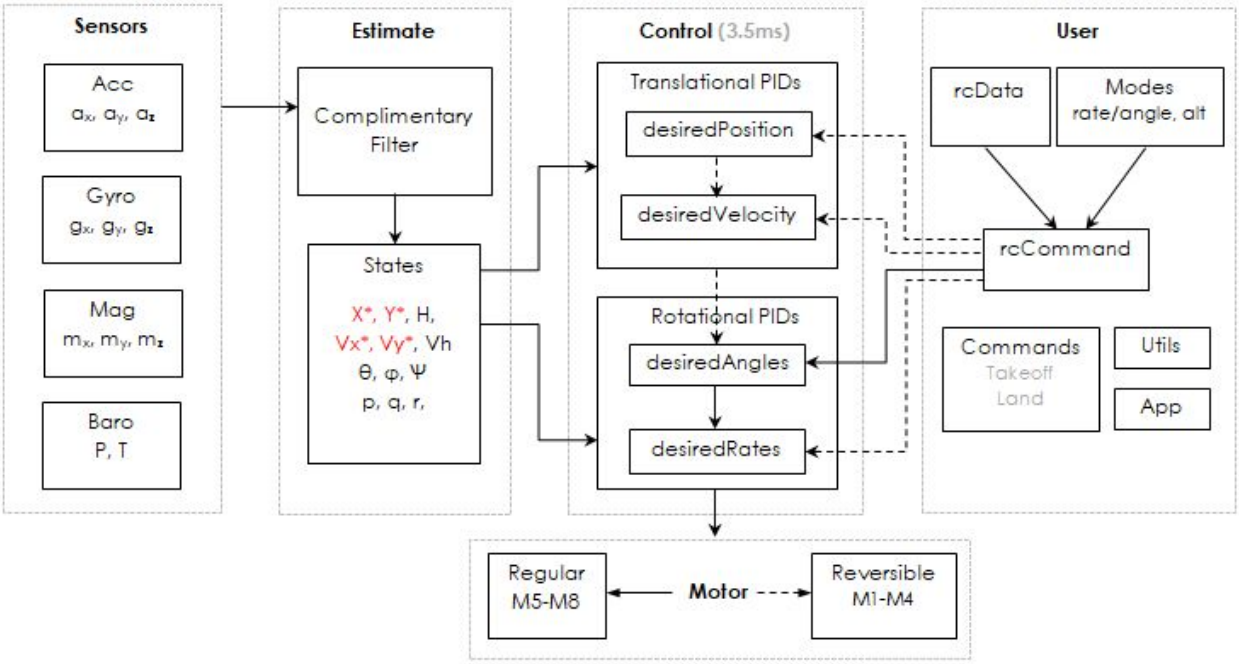


Unibus

2.2 Magis - The software

All the hardware is of no use unless coupled with the software, and this combination together makes PlutoX extraordinary. It was mentioned earlier that the tinkerers do not have to worry about programming PlutoX for ordinary flights. PlutoX runs on Magis, which provides stable flight.

Many tinkerers want to understand the working of Magis and the processes taking place in the system which make it so stable while flying. This can be understood with the help of the block diagram below explaining the structure of the firmware. There are five main blocks which work together to make PlutoX fly. The functioning of each of these blocks should be understood by a tinkerer as it will provide a base for working with PlutoX and also help in programming new features on it.



Sensors Block: In order to control any system, it is required to know the current condition of that system. Condition could refer to a number of variables that a system can have. The smallest set of variables which completely describe a system are called as state variables. The value of these state variables can be obtained by using sensors.

All the four primary sensors - Accelerometer, Rate Gyro, Magnetometer and Barometer continuously supply values of the current state variables of PlutoX. Accelerometer provides the values of perfect acceleration along all three directions. Rate Gyro provides values of angular rate about three body axes. Magnetometer provides data of magnetic fields for all three axes. Barometer provides the instantaneous pressure and temperature readings.

It is important to know that all the data provided by each of these sensors is very noisy. Such raw data can not be directly utilised in estimation of the states of the drone. The data provided in Sensor Block is forwarded to Estimate Block where further processes take place.

Thus, the Sensor Block will contain all the raw data provided by each sensor. Including this block as a header while coding gives access to all the data contained in this block. So, for example, the current reading of temperature or pressure can be

accessed by using “Sensor.h” header. (To know about ‘header’ please refer to “3.2.1 Header”)

Estimate Block: The data received from the Sensors Block is raw, containing a lot of noise. This noise is filtered out by the complementary filter and the filtered data is used for estimating the values of state variables. The values of certain state variables are not obtained directly through the sensors. These values are then estimated in this block by using other state variables’ estimations as they are interdependent.

Using the values obtained from the four sensors currently mounted on Primus X, it estimates the values of 8 out of 12 state variables. These are: Vertical distance/altitude/height (H), vertical velocity (V_h), Roll (Θ), Pitch (Φ), Yaw (ψ), Angular Velocities along the X axis (p), Y axis (q) and the Z axis (r).

Estimate block contains all the filtered data regarding the current angle, rate, position and velocity, which is then fed to the Control Block. The access to all the filtered data can be gained by using the header “Estimate.h”. This data can be utilised in programming new features. For example, suppose a feature where in PlutoX should not tilt beyond a specific angle of roll. The limiting angle of roll can be mentioned based on choice but it will have to be compared with the current angle of roll. In order to get the current angle of roll, the data can be accessed from Estimate Block and that will be done by using the header “Estimate.h”.

User Block: This block represents the input provided by the user or the pilot. A pilot can use Pluto Controller app for flying PlutoX. Using the app, the pilot will be controlling the pitch, roll, yaw and thrust movements of PlutoX. Let us understand the purpose of this block by taking an example of pitch. Suppose, the pilot wants to pitch forward. So the pilot moves the right joystick in front direction. The magnitude of the forward movement of the joystick will be taken as the data for rcData. This data is used to generate rcCommand. It is the modified rcData after applying low pass filters, exponential, etc. rcData is converted into user friendly range in rcCommand. rcCommand is further given as input data to the Control Block and the further process takes place over there.

It is interesting to notice that the user block contains different types of flying modes, such as rate mode, angle mode, altitude hold mode, etc. Depending on the type of mode the user selects, rcCommand will be generated. For example, if the user has selected angle mode, then the rcData while being generated into rcCommand takes

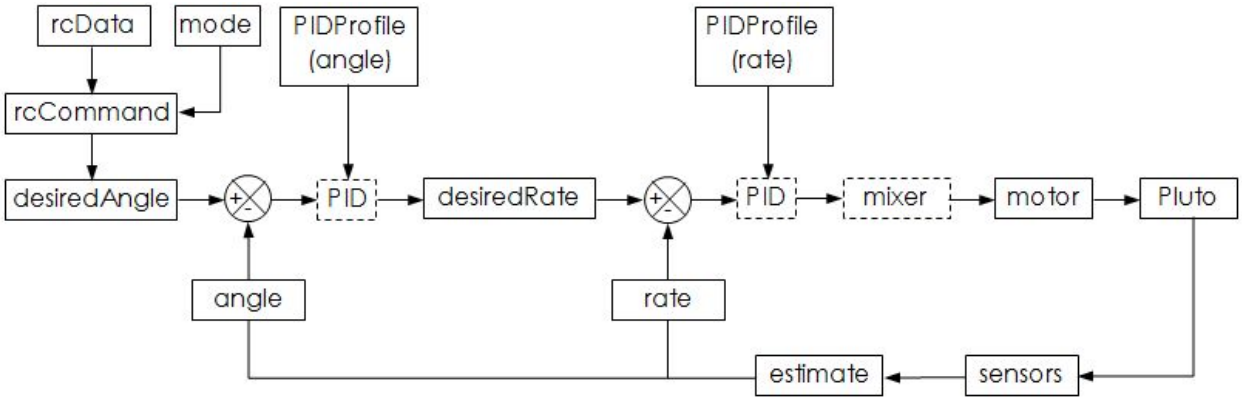
into account that the rcCommand has to be the angle that the user wants to change to or the desired angle. If the selected mode is rate mode, then while generating rcCommand from rcData, it is taken into account that rcCommand has to be the rate at which user wants to change the angle or the desired rate.

User block contains data regarding the input that the user gives through remote control/app (rcData), the data that is forwarded to the Control Block (rcCommand), data regarding the flight modes, current flight status. This block can give access to data regarding all above. Hence, in order to check whether PlutoX is flying or crashed, to change the flight mode, to set the rcCommand data for roll, pitch, yaw and thrust, etc. one can use the header for this block, “User.h”

The User Block also contains Utils Block. Utils Block basically consists of all debug APIs like LEDs, print, graph. The access to data regarding any of these can be obtained by using the header “Utils.h”. For example, to use red LED as an indication for rolling right, it can be programmed to turn ON when PlutoX rolls right.

Control Block: Control Block, as the name suggests, controls the movement of the drone across all the axes, both translational and rotational. This is achieved by using PIDs. This block receives input data from Estimate Block and from User Block. The Estimate Block provides the current values of the state variables of PlutoX. The User Block provides the desired values of the state variables as received from the user. In order to understand how Control Block works, let us take an example.

Imagine a user wants to move PlutoX in the forward direction i.e. to pitch forward. When it is stable in the air before providing any pitch input, the Estimate Block provides the particular values of state variables at that instant. As soon as the user pitches forward, the rcData generates rcCommand considering the flight mode, and this rcCommand generated is forwarded to the specific part in the Control Block. Control Block generates the desired position at which PlutoX has to be taken to, the desired translational velocity at which PlutoX should move, the desired angle at which PlutoX should incline at and the desired rate at which PlutoX will change its angles. It should be noted that the ‘desired’ values are generated by considering the current values of the state variables and input from the user, i.e. essentially it generates the correction to the current error. So if PlutoX is pitching forward at 10° and the rcCommand is to tilt it to 30°, then the desired angle value would be 20° and based on the desired angle, it will further generate the desired rate of change of angle to reach there. It can be understood by the following control loop diagram, which takes into account angle mode as the flight mode.



It is interesting to notice that the Control Block eventually generates the desired rate as the output of the block. So if the user is flying in rate mode, then the rcCommand will be given directly to the desired rate block.

Thus, Control Block contains data regarding the control of PlutoX. It will give access to the desired angle, desired rate, desired position, desired velocity, control limit and PID systems. One can access their values or set values for them depending on the requirements. In order to access them use “Control.h” header while programming.

Motor Block: The desired rate data generated from Control Block is provided to the Motor Block. This block controls the power to the motors. Depending on the desired rate of change of angle, the power to the motors is changed. If higher rate is required then the power of motors is increased and the power is decreased if the required rate is lower.

There are eight motor drivers, M1-M4 being the reversible motor drivers and M5-M8 being the unidirectional motor drivers. By default, only M5-M8 drivers are functional. In order to use M1-M4 drivers, they have to be initiated while programming, which will be explained further in the book.

“Motor.h” header will give access to all the motors, which can be used to:

- Initiate motors
- Set the direction (in case of reversible drives)
- Read power
- Set power

To summarise, the entire firmware of PlutoX is divided into blocks which carry their own set of tasks. The Sensors Block provides the raw data of the state variables, which is filtered in Estimate Block to make it usable, the User Block provides the input taken from the user, Control Block generates the desired rate from the input and the Motor Block controls the power to the motors. The entire firmware works continuously for a stable flight of PlutoX.

3. Ease of tinkering with PlutoX

The idea behind this specific design of firmware of PlutoX is to make it as easy as possible to tinker with it. In order to understand why it is so simple to tinker with it, there are certain points that a tinkerer needs to understand.

3.1 No programming for normal flights

Firstly, a tinkerer has to understand that Magis continuously runs inside the hardware of PlutoX. Even while programming PlutoX for tinkering, the core software which results in the seamless flight still continues to run. Thus, the tinkerer does not need to program the drone again or does not need to flash Magis again after testing. The entire tinkering program will run simultaneously along with Magis.

3.2 Tinkering Loop

While programming an idea, tinkerer will not require to make changes in Magis. A separate loop has been provided for the tinkerers to program their idea. This loop is called just before the Control loop. The default frequency of calling is 3.5ms, which can be changed if needed. The entire program can be coded in this loop by the tinkerer and when the loop is called, the code will run. Let us understand the loop in a better way.

This is the structure of the loop:

```
/* Do not remove the include below*/
#include "PlutoPilot.h"

/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{

/* Add your hardware initialization code here*/

}


/*The function is called once before plutoPilot when you activate Developer Mode*/
```

```
void onLoopStart()
{

/* do your one time tasks here*/

}

/*The loop function is called in an endless loop*/
void plutoLoop()
{

/*Add your repeated code here*/

}


/*The function is called once after plutoPilot when you deactivate Developer Mode*/
void onLoopFinish()
{

/* do your cleanup tasks here*/

}
```

The loop basically consists of five parts. Understanding each of these parts will provide a baseline for programming.

3.2.1 Header

This is the part of the code where the headers are mentioned. These headers are selected depending on the data required. As mentioned earlier, Magis contains data under headers like Sensor.h, Estimate.h, User.h, Utils.h, Control.h and Motor.h

```
#include "PlutoPilot.h"
#include "Estimate.h"
#include "Utils.h"
```

3.2.2 void plutoInit ( )

This function is immediately executed after PlutoX powers up. It is used to initialize any spare hardware of PlutoX which is not initialized during normal flights. For example: motors M1 - M4 are not initialized during normal flights. If these motors are to be used for tinkering, then they have to be initialized as follows:

```
void plutoInit()
{
	Motor.initReversibleMotors();
}
```

3.2.3 void onLoopStart ( )

This function is executed once, after the user turns the developer mode on. It is used to perform tasks which are performed only once, such as initializing some variable or changing the default behaviour of a part of the system. For example, turning the default LED behaviour OFF does not need to be performed in iterations. Performing it only once in the beginning is enough. It can be done as follows:

```
void onLoopStart()
{
	LED.flightStatus(DEACTIVATE); /*Disable default Led behaviour*/
}
```

3.2.4 void plutoLoop ( )

This is the function that is executed along with PlutoX's internal primary stabilization code. By default, this loop runs every 3.5 ms which can be changed with the use of APIs. The core iterating logic of the program is coded in this function.

3.2.5 void onLoopFinish ( )

This function is executed once after the user turns the developer mode OFF. It is useful to undo the tasks done in "On Loop Start" part to restore the defaults of PlutoX. For example, default flight status of LED can be restored over here as follows:

```
void onLoopFinish()
{
        LED.flightStatus(ACTIVATE); /*Enable the default LED behaviour*/
}
```

3.3 Application Programming Interface (API)

APIs are essentially functions which help in performing complex tasks much easily. Consider a case where PlutoX has crashed. After a crash, it is essential to automatically disarm PlutoX otherwise it could be dangerous if the motors and propellers are still running. To disarm automatically, the system requires to check whether PlutoX has crashed or not. In order to check this condition, one needs to first understand the dynamics of a drone during a crash. While crashing, the drone experiences more acceleration than normal. That value of acceleration can be identified using accelerometer sensor. After performing a number of crashes, it was concluded that the value can generally go up to 2g. Also, the angles of roll and pitch would change to a value at which it will become difficult to fly the drone. Hence the next step would be to identify these angles. The values of these angles are set at 70°. Hence, if either of these conditions is satisfied, then PlutoX should disarm itself. In order to do that, the system has to constantly check the values of acceleration, roll and pitch. And when the condition is satisfied, the task of disarming will take place.

While tinkering, if the tinkerer requires to check whether PlutoX has crashed or not, they do not need to do all this work. Magis provides the tinkerers with APIs to access all the data from the firmware, making the entire process of programming easier. Tinkerers can simply use 'FlightStatus.check' API to check if PlutoX has crashed or not.

The entire list of the APIs can be accessed using the following link:

LIST OF APIs

4. Projects

The primary aim behind this project book is to help in giving the tinkerers a kick start to the universe of PlutoX. The entire information prior to this was to give an idea about PlutoX, its working and its capabilities. Using the platform provided through PlutoX, one can develop a number of applications across a variety of areas. The book further provides a set of projects that will help in building a practical approach towards identifying a problem and solving it by programming the ideas on PlutoX.

5. Prerequisites

Before beginning with the projects, please make sure to fulfill the prerequisites for a smooth learning and tinkering experience.

5.1 Cygnus IDE

In order to work with PlutoX, install Cygnus IDE where programs will be developed and flashed on PlutoX. Please follow the steps as mentioned to complete this prerequisite:

5.1.1 Installation

Step 1: Download the Cygnus IDE from the following link:

Cygnus IDE

Step 2: Extract the downloaded Cygnus. Make sure that the installation path does not contain any spaces. For example:

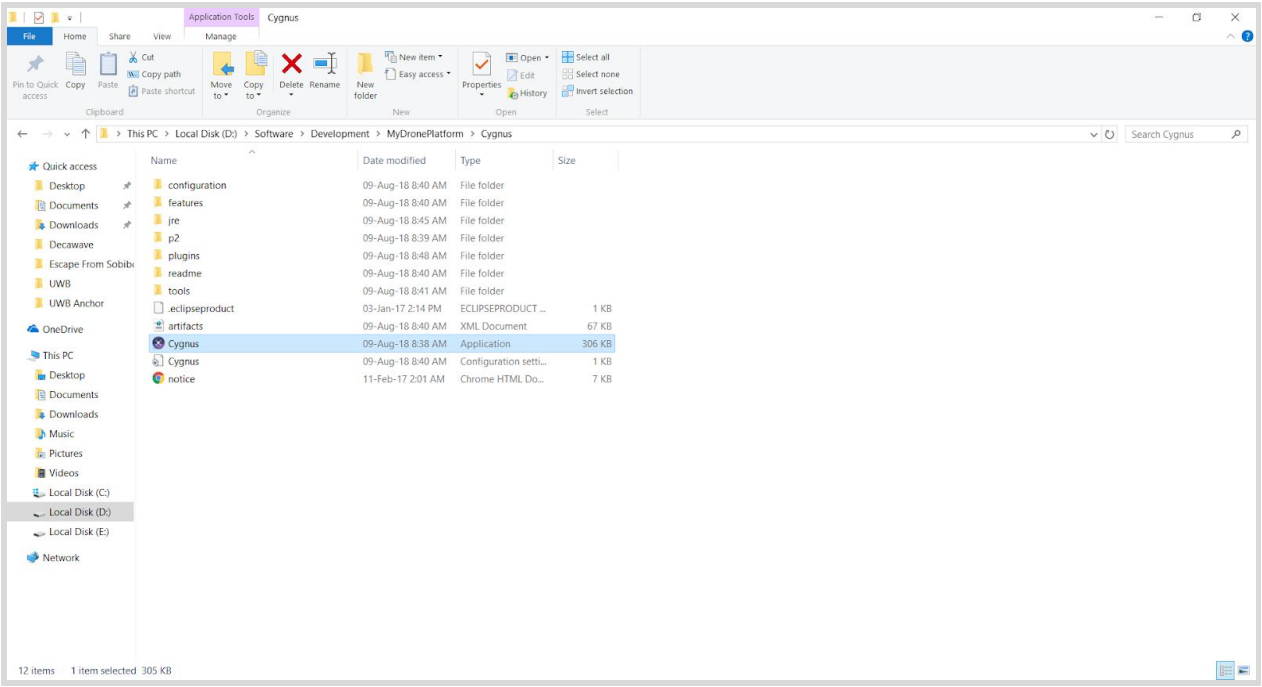
"D:\Software\Development\My Drone Platform\cygnus"

The above path will be invalid as it contains spaces in "My Drone Platform".

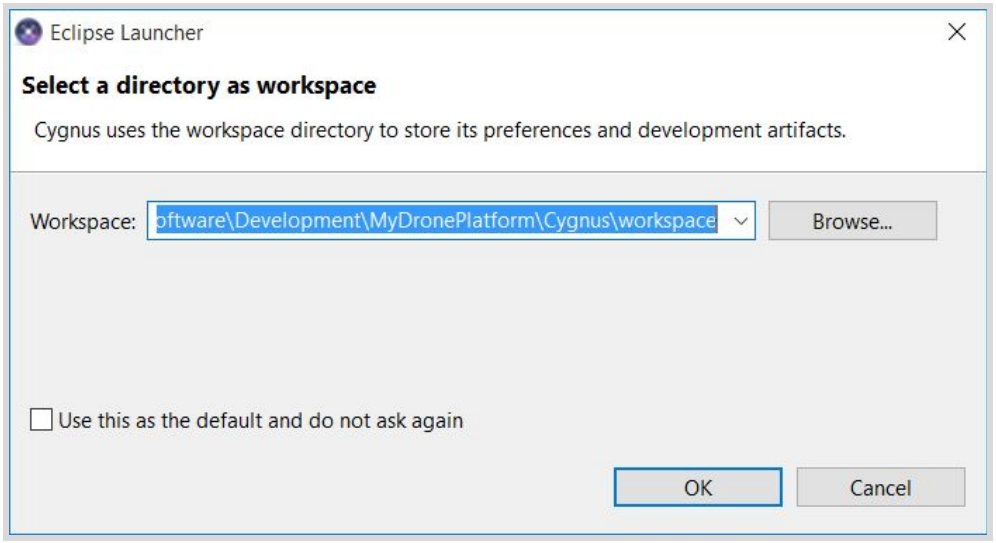
"D:\Software\Development\My-Drone-Platform\cygnus"
"D:\Software\Development\MyDronePlatform\cygnus"

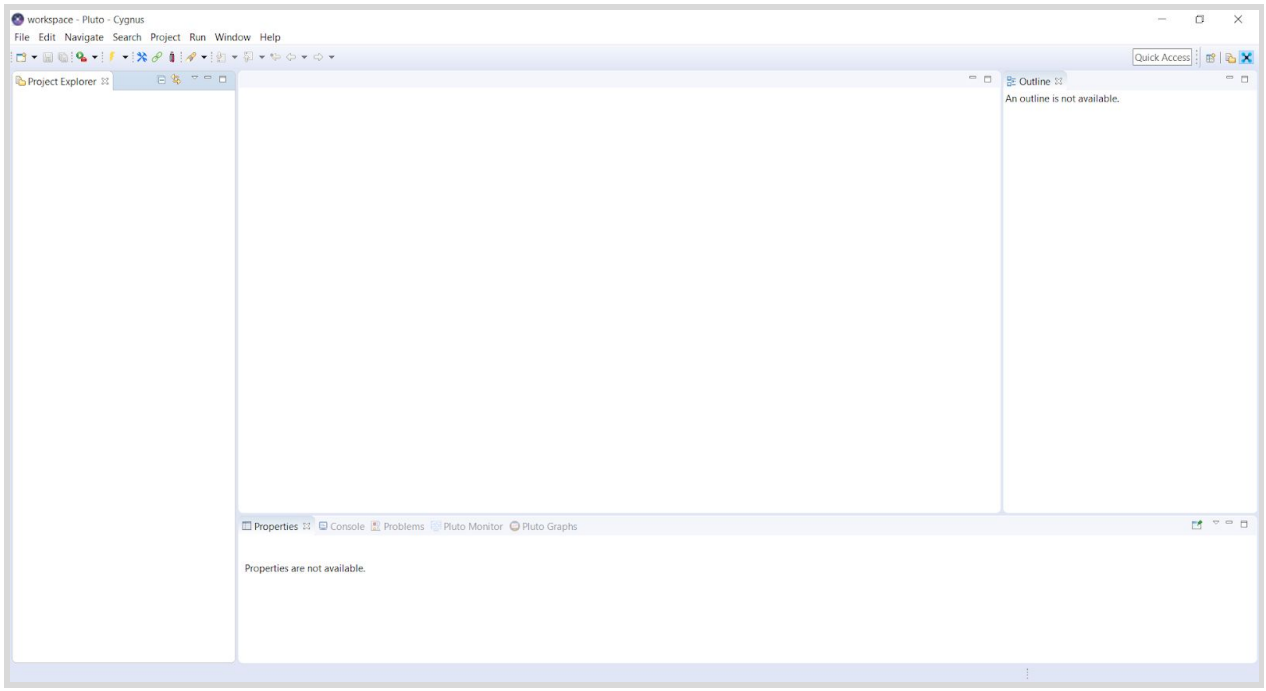
Both the above paths are valid as they do not contain any spaces. This is required for the Cygnus compiler to work properly.

Step 3: Run “Cygnus.exe”. Make sure it has an executable permission.



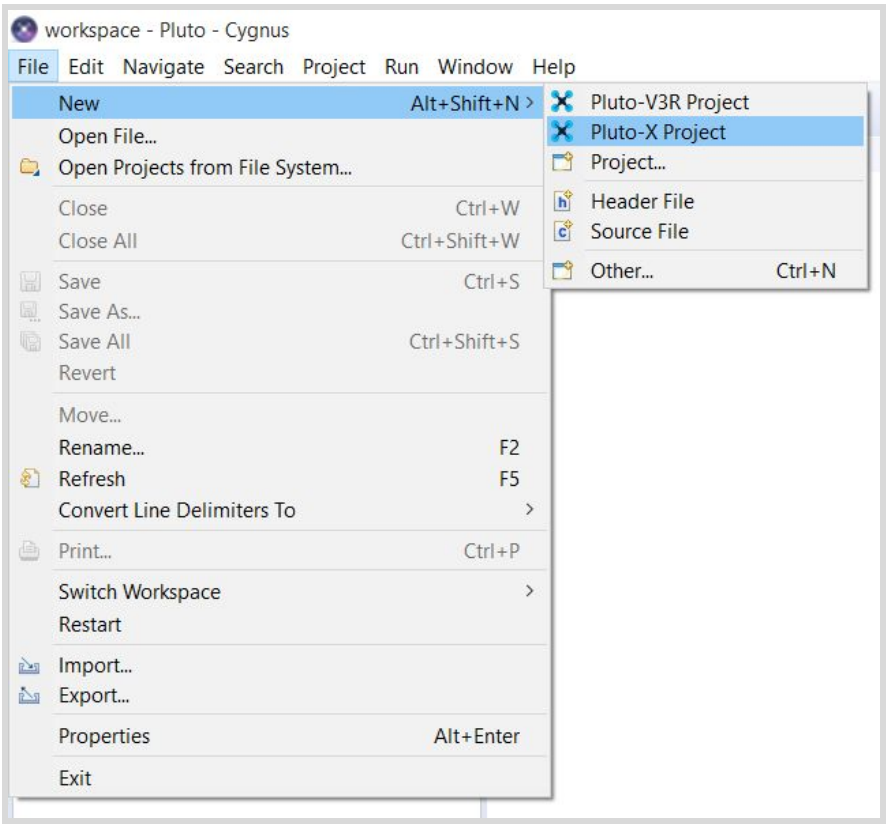
Step 4: Select the Workspace. Default workspace is created in the Cygnus folder itself. If you want to change the path of workspace you are free to do so. However, we recommend you to continue with the default workspace itself.



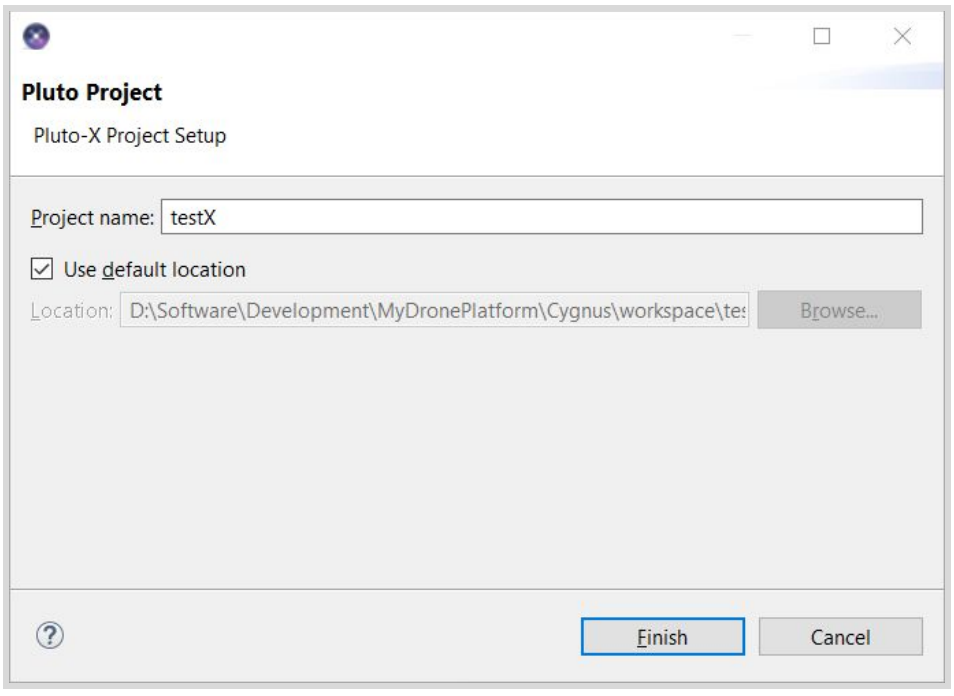


5.1.2 Verification

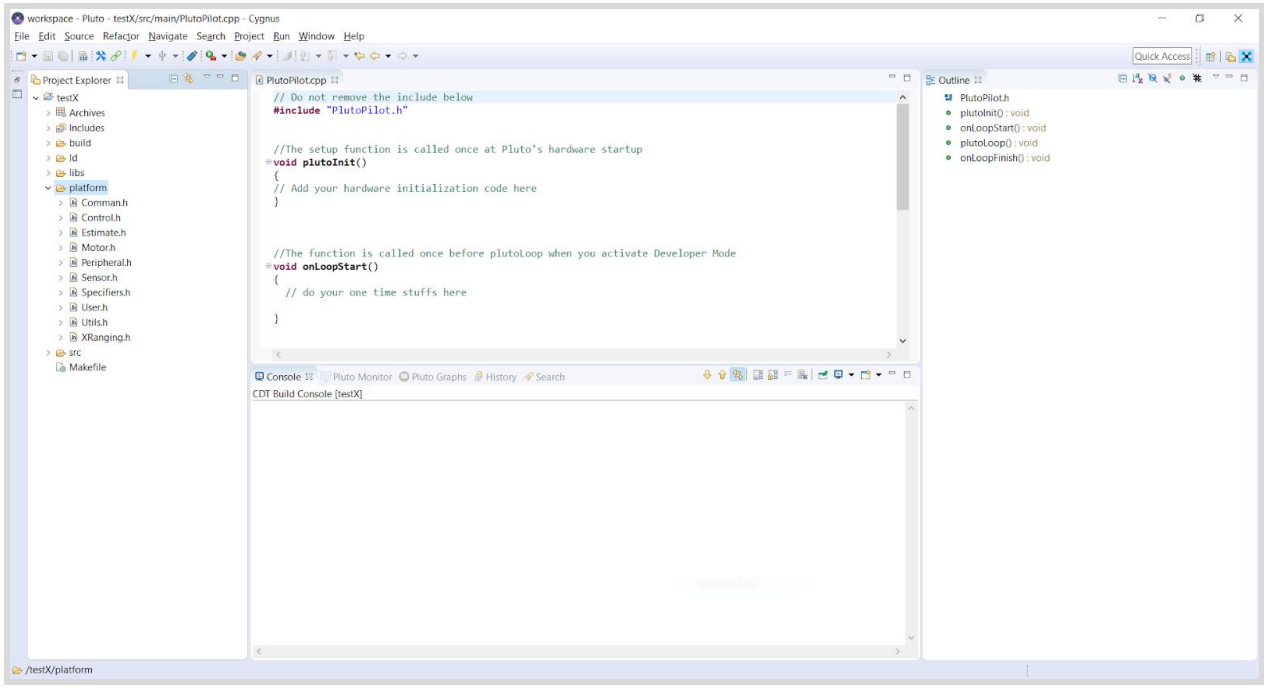
Step 1: To verify if installation is successful, create new test project. Go to File -> New -> PlutoX Project



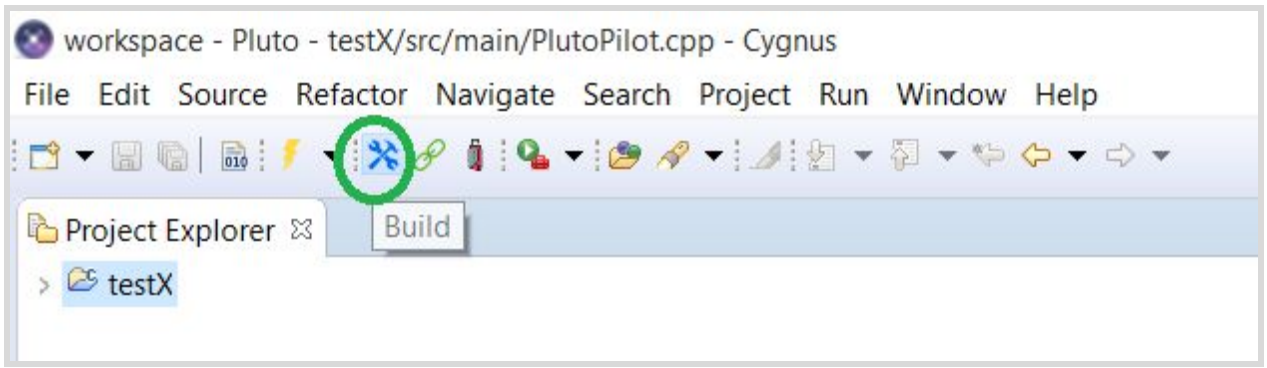
Step 2: A PlutoX project setup pop up will appear. Name the project as “testX” and after naming, press Finish. A “PlutoPilot” file will open in the editor.



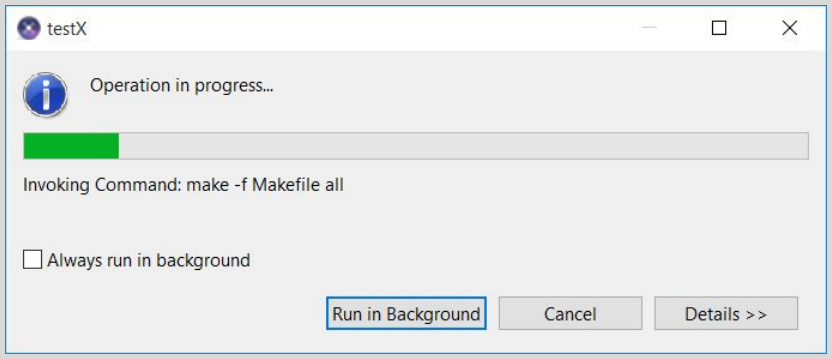
Step 3: To make sure that the project is created successfully, open “platform” folder and check if API header files are present.



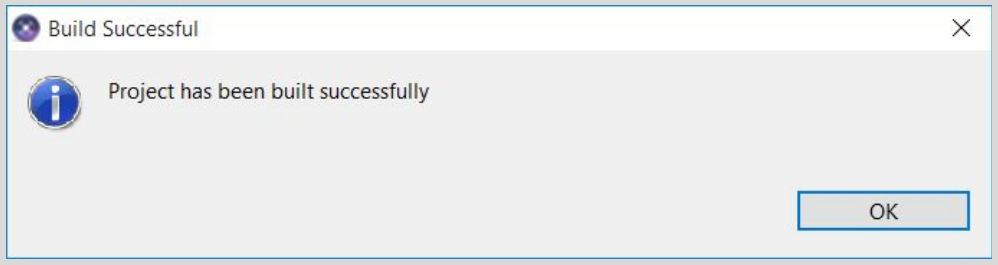
Step 4: Once project is created successfully, build it. Select the “Build” command from toolbar. Build command will build the project that you have selected or it will build the project of current file opened in editor by default.



Cygnus will start building the project.

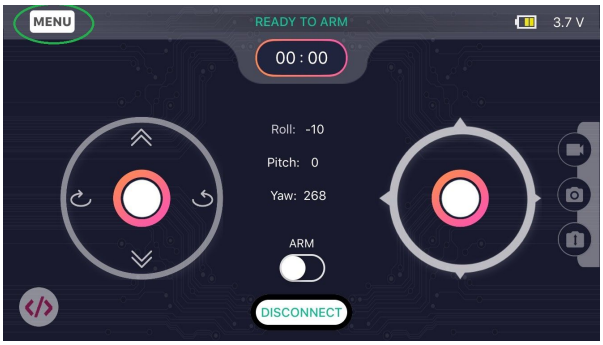
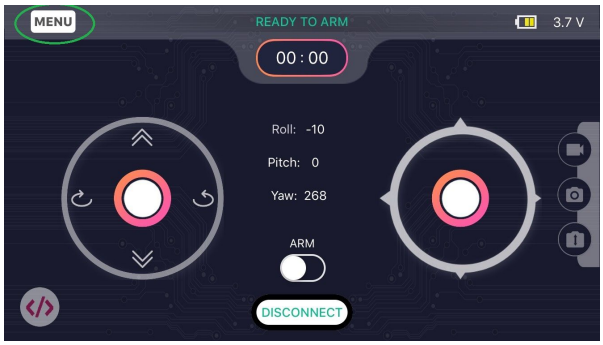


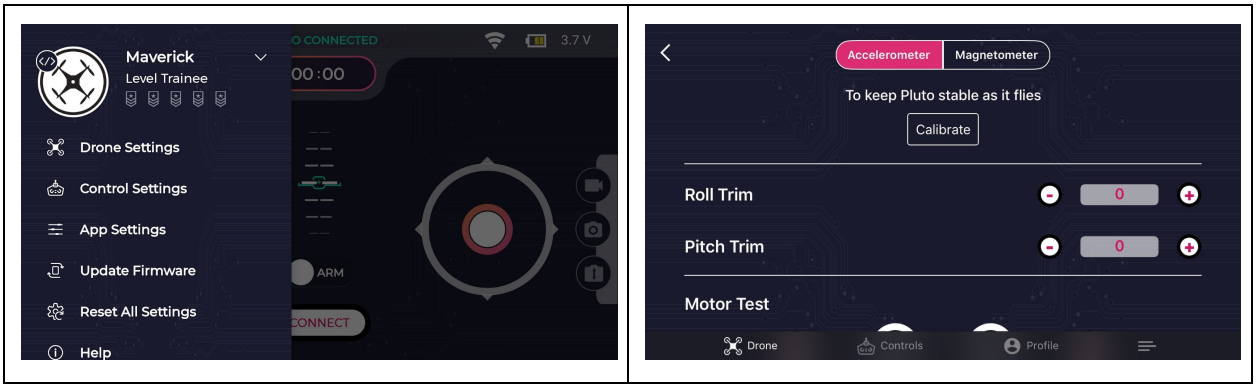
A pop up saying “Project has been built successfully” message should appear after the build process. This message verifies that Cygnus is installed and configured successfully.

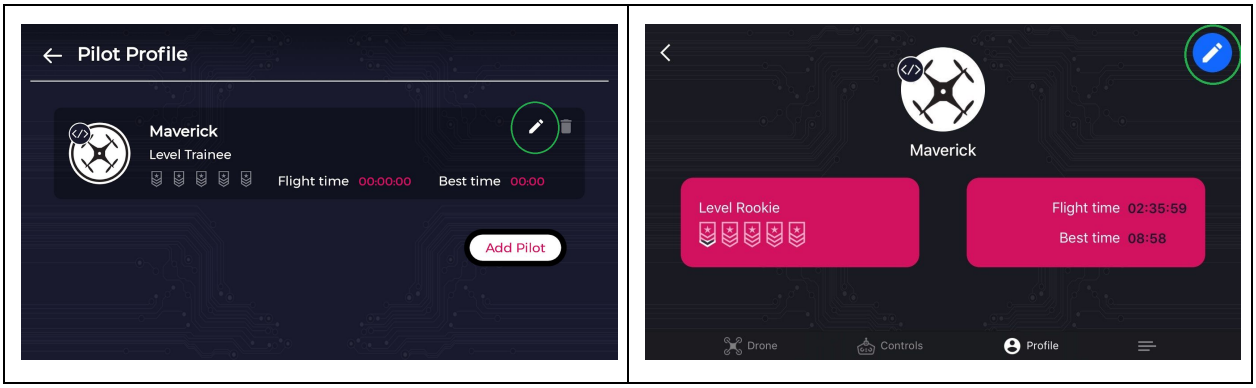


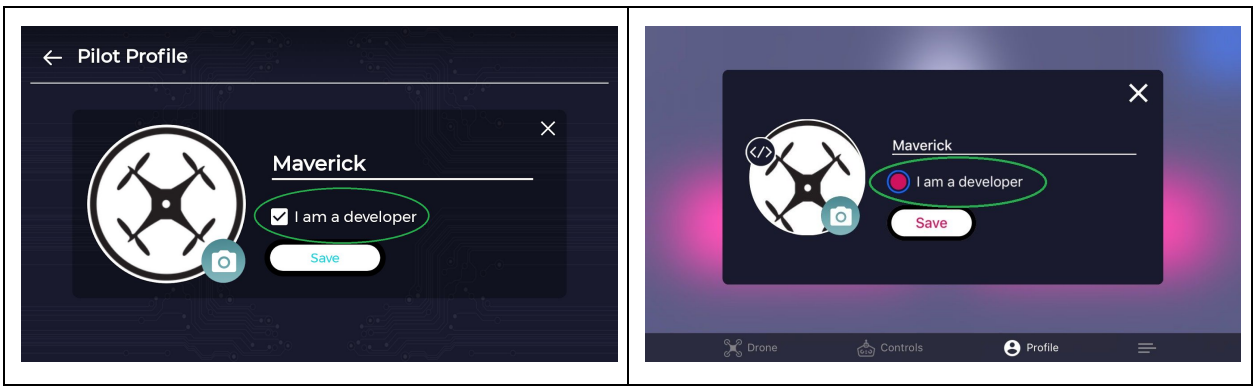
5.2 Developer Mode

Another prerequisite is to select “I am a developer” option in the profile settings in the Pluto Controller app. If the option “I am a developer” is not selected in the profile then the ‘Developer Mode’ button will not appear on the main screen. To enable Developer Mode button on the main screen, go to menu, select your profile and edit to check “I am a developer” option

Android	IOS
	

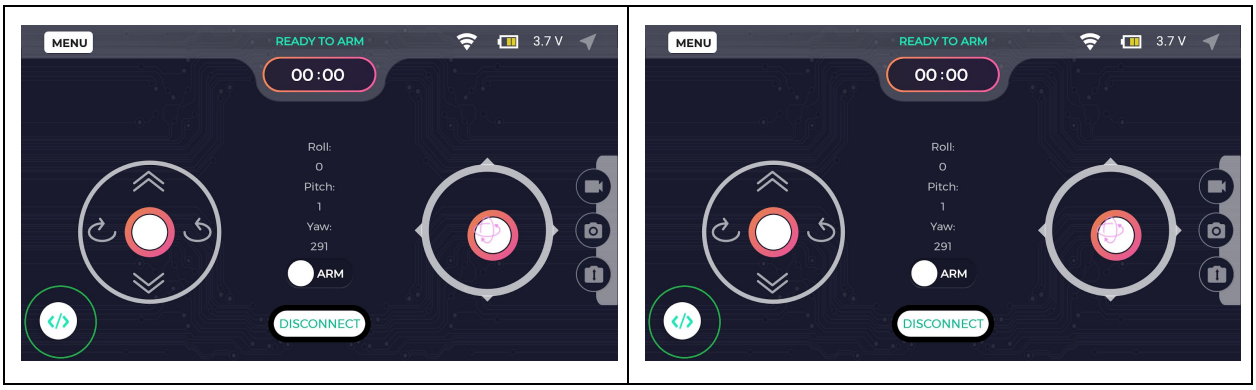




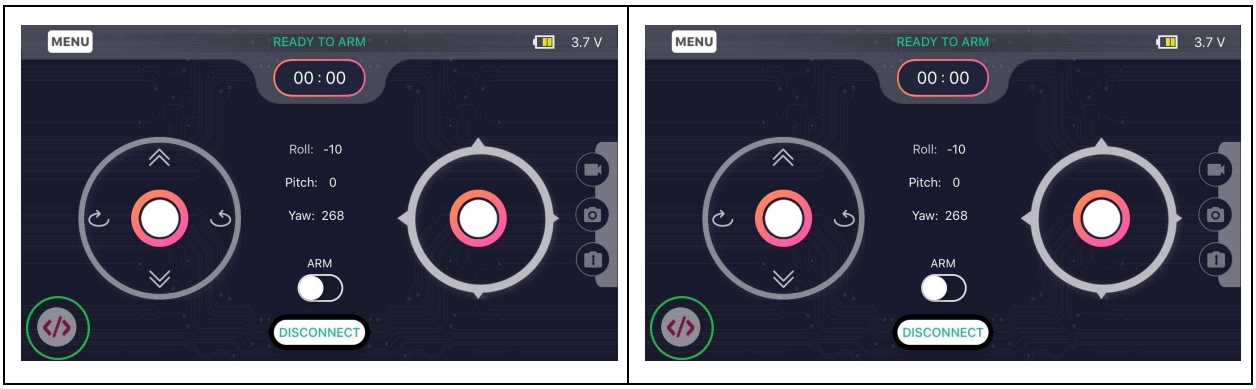


After selecting the “I am a developer” option, you can see Developer Mode button on your main screen.

Fade in (green) indicates that the Developer Mode is ON.



Fade out (red) indicates that the Developer Mode is OFF.



After successfully completing all the prerequisites, begin with the projects designed for PlutoX. Happy Learning!

PROJECT 1: DEBUG APIs (Pluto1.2 & PlutoX)

Objectives

- Learning the usage of Cygnus and its APIs.
- What is debugging.
- How to use debug APIs like LED, print.

Problem statement

To detect the movement of PlutoX in vertical direction (upwards and downwards) by using LEDs on board and printing the velocity values on the Pluto Monitor.

Explanation

Debugging is the process of locating and removing the bugs in the software or hardware. In case of PlutoX, tinkerers would program the ideas and flash them on the drone. In certain cases, the code might not work as expected or might show some errors. In such cases, debugging APIs can be used to identify the bug and solve them.

The debugging APIs include LEDs, print and graph. LEDs can be used for visual confirmation of a task, print can be used to display the values on the screen of the computer and graph can be used for plotting the graph of values on the screen. Using these APIs, it becomes easier to understand the issue with the written program and to debug it.

This project aims at understanding a basic application of debug APIs.

Approaching the problem

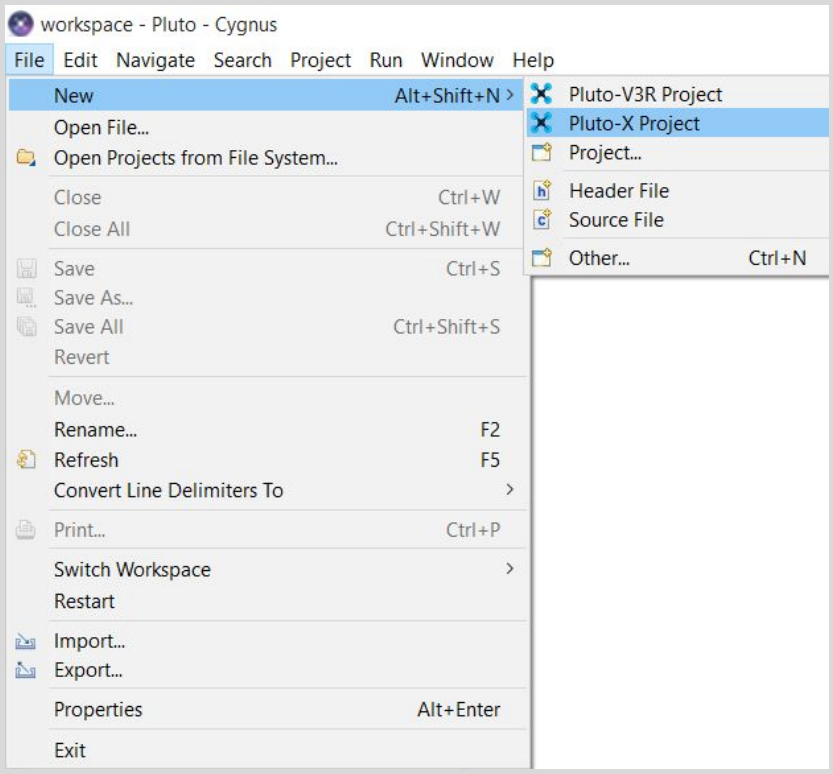
When PlutoX moves upwards, the vertical velocity, i.e. velocity in z axis will be positive and when it moves downwards, the same velocity will be negative. The values of velocity can be obtained from sensors. This velocity data provided by the sensors can be used to execute this project.

When the value of velocity is positive, the red LED can be turned ON and when the value of velocity is negative, the green LED can be turned ON. This will be a visual

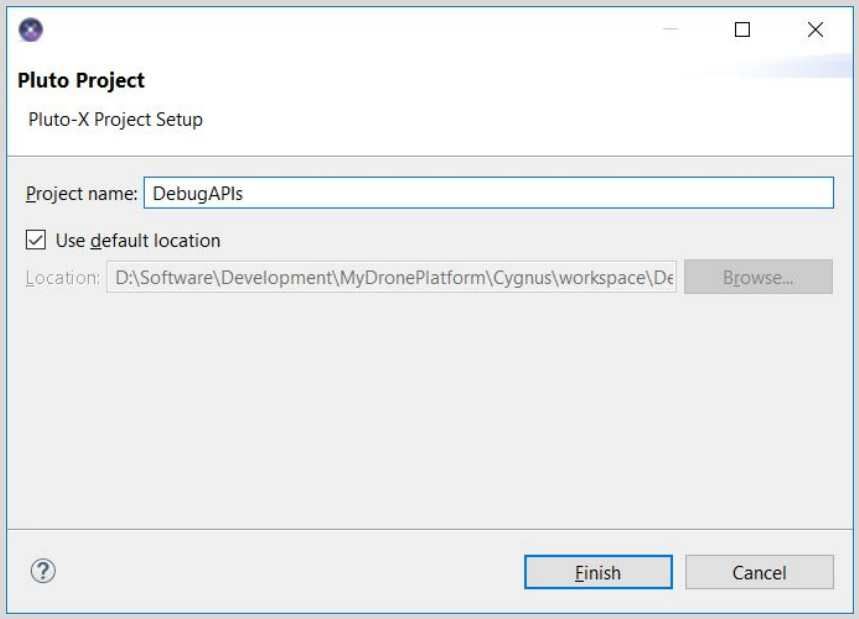
confirmation of the movement of PlutoX in the vertical direction. The velocity data can also be printed on Pluto Monitor and can also be plotted on a graph.

Creating a new project

- Select a new Pluto-X Project



- Name the project as “DebugAPIs”



Starting with the coding:

Headers

This project requires the velocity data in the z axis. This data can be obtained from the header “Estimate.h”. To access LEDs, print and graph APIs, use “Utils.h” header. Include these headers in the code.

```
#include "PlutoPilot.h"
#include "Estimate.h"      /*gives access to drone rates, angles, velocities and positions*/
#include "Utils.h"         /*gives access to LED, Graphs and Print*/
```

void plutoInit ( )

No spare hardware to be initialized.

void onLoopStart ( )

LEDs have a default programmed behaviour in Magis. In order to program the LEDs as per the requirement of the project, it is mandatory to disable the default behaviour as soon as the developer mode is turned ON. This can be done by using the ‘LED.flightStatus’ API.

```
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable default Led behaviour*/
}
```

void plutoLoop ( )

The logic is based on the condition whether the velocity in the z axis is positive or negative. This can be done by using **if else** condition. If the velocity is positive, then turn the red LED ON and turn the green LED OFF or else (if the velocity is negative), turn the green LED ON and turn the red LED OFF. For changing the status of LEDs, use 'LED.set(LED, STATE)' API.

Printing the current value of the velocity in the z axis and plotting the corresponding graph has to be done irrespective of the condition. Hence, this part can be coded outside the **if else** loop. It is done by using 'Monitor.println(tag, number)' and 'Graph.red(value, precision)' API.

```
void plutoLoop()
{
/*Add your repeated code here*/
        if(Velocity.get(Z) > 0)        /*If the drone is moving upwards (Velocity in the
Z axis will be positive)*/
        {
                LED.set(RED, ON);
                LED.set(GREEN, OFF);
        }
        else    /*If the drone is moving downwards*/
        {
                LED.set(RED, OFF);
                LED.set(GREEN, ON);
        }

        Monitor.println("Velocity Z: ", Velocity.get(Z));
        Graph.red(Velocity.get(Z), 1);
}
```

void onLoopFinish ( )

In this project, the default LED behaviour was deactivated at the beginning of the loop and hence, after the developer mode is turned OFF, it has to be activated again. This is also done by the same 'LED.flightStatus' API.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);      /*Enable the default LED behaviour*/
}
```

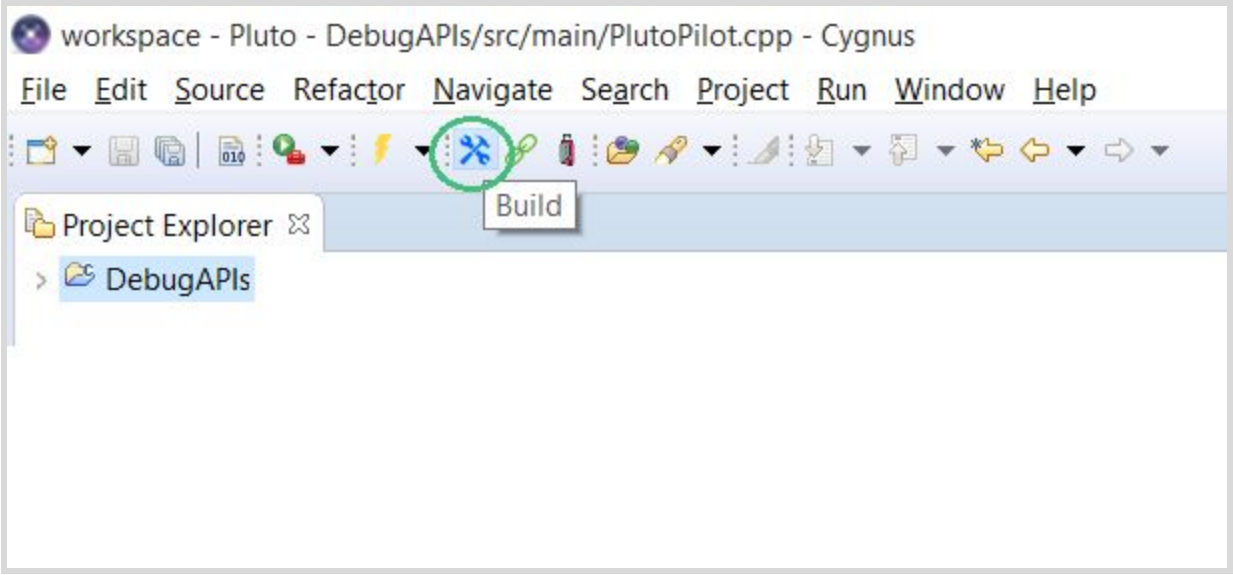
The entire code:

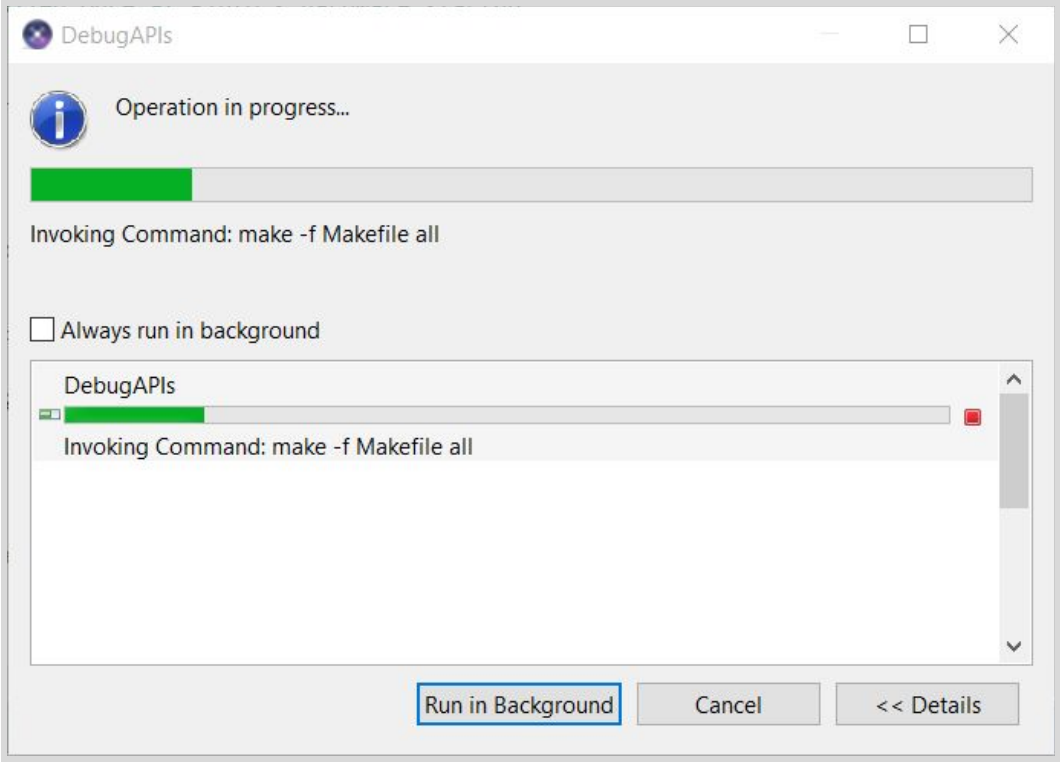
```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include "Estimate.h"       /*gives access to drone rates, angles, velocities and positions*/
#include "Utils.h"          /*gives access to LED, Graphs and Print*/


/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/
}


/*The function is called once before plutoPilot when you activate Developer Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable default Led behaviour*/
}

/*The loop function is called in an endless loop*/
void plutoLoop()
{
```

```
/*Add your repeated code here*/
        if(Velocity.get(Z) > 0)        /*If the drone is moving upwards(Velocity in the
Z axis will be positive)*/
        {
                LED.set(RED, ON);
                LED.set(GREEN, OFF);
        }
        else    /*If the drone is moving downwards*/
        {
                LED.set(RED, OFF);
                LED.set(GREEN, ON);
        }

        Monitor.println("Velocity Z: ", Velocity.get(Z));
        Graph.red(Velocity.get(Z), 1);
}

/*The function is called once after plutoPilot when you deactivate Developer
Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);      /*Enable the default LED behaviour*/
}
```

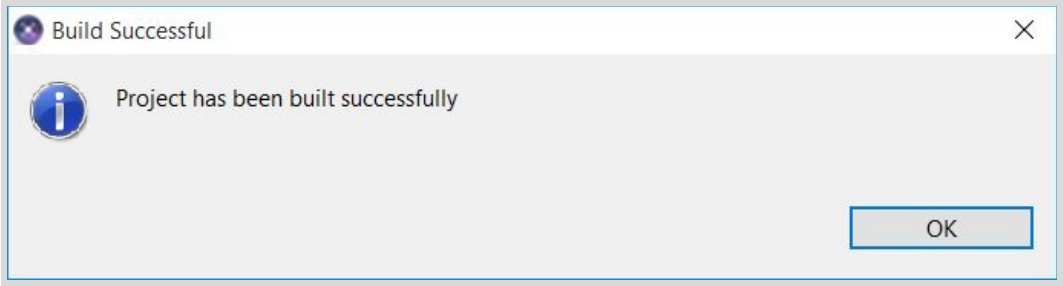
Building the project

After the completion of coding, build the project by selecting the Build command from the toolbar. Build command will build the selected project or it will build the current file opened in the editor by default.



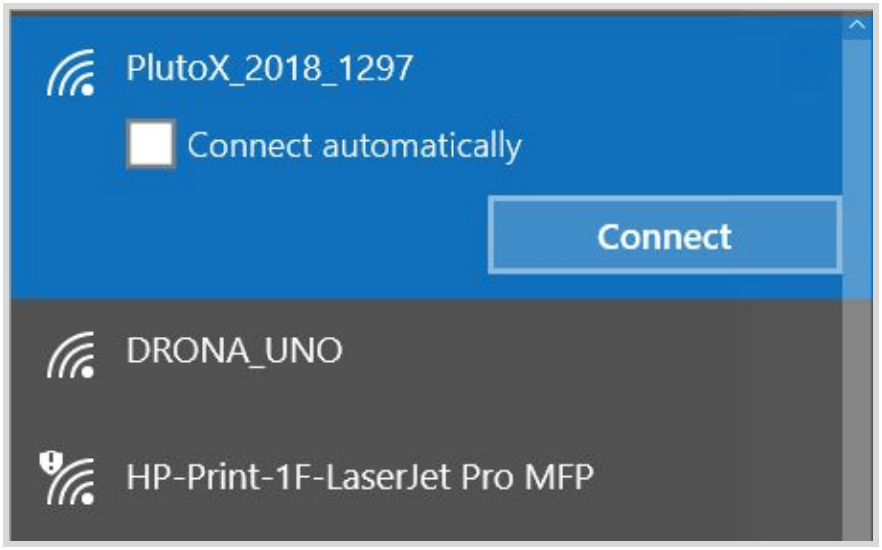


After selecting the Build command, Cygnus will start building the project. If there is some error in the code, it will be prompted on the screen. If the code is error free, cygnus will display “Project has been built successfully” message on the screen.



Flashing the code

After building the project, the next step is to flash it on PlutoX. For this, use the wireless flashing feature by connecting PlutoX's WiFi with the computer.



After connecting PlutoX's WiFi to the computer, select Flash command from toolbar. Flash command burns the code from the selected project or it burns the code from the project of current file opened in editor by default.

There are two types of flash available:

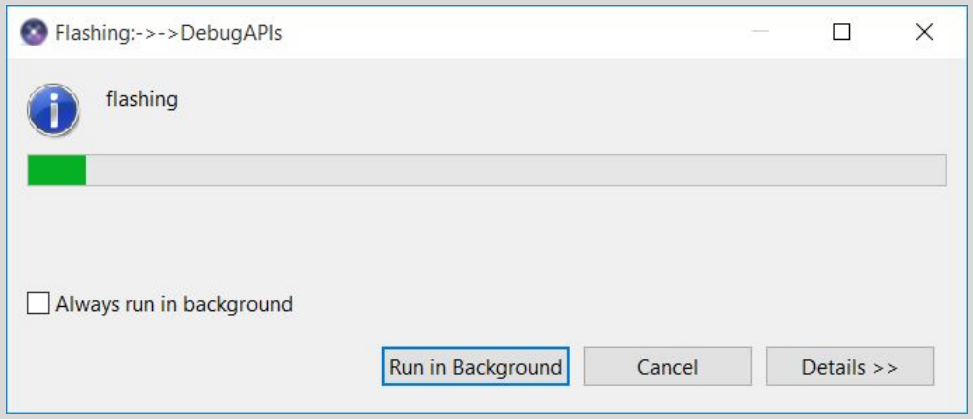
- Full Flash: Burn the code onto PlutoX by erasing ROM data like sensor calibration.
- Normal Flash: Burn the code onto PlutoX without erasing ROM data like sensor calibration.

It is recommended to use Full Flash when programming for the first time. Be sure to calibrate the accelerometer and magnetometer sensors after every Full Flash. If any

unusual behaviour of PlutoX is observed after Normal Flash, then prefer using Full Flash again.

NOTE: Do not connect PlutoX with the App during the process of flashing.

Cygnus will start flashing the code.

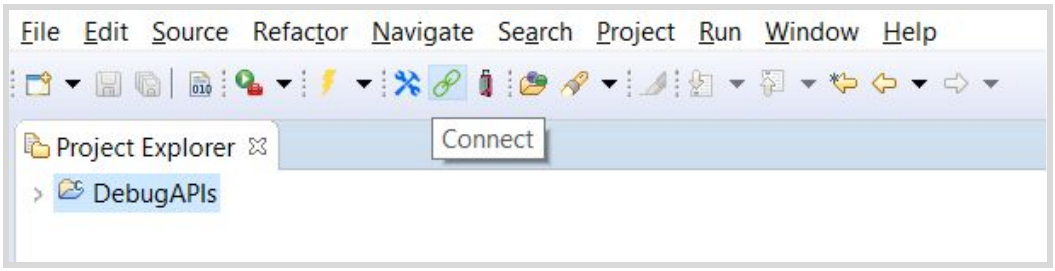


Cygnus will notify “Firmware Flashed Successfully”.

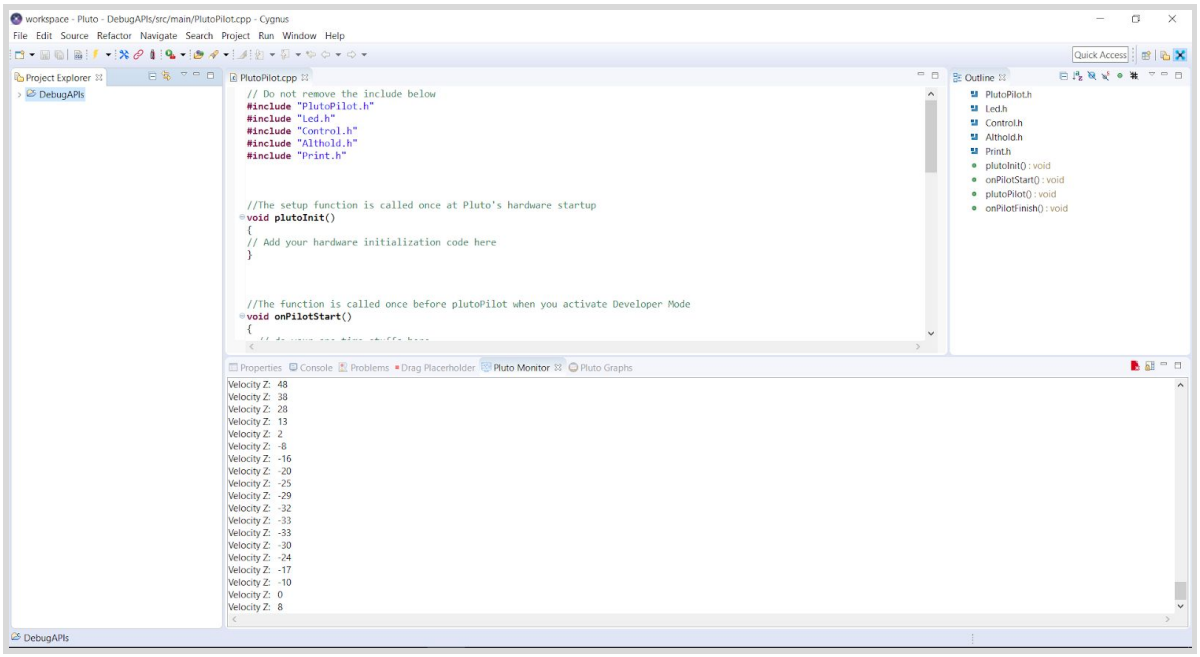
Running the project

To run the program on PlutoX after flashing it, activate Developer Mode using Pluto Controller on Android or iOS application.

In order to observe the velocity data on Pluto Monitor or on the graph, connect PlutoX with Cygnus using the connect button.



Now move the drone up and down by holding it in hand without actually flying it so that the velocity values across z axis can be observed.



Project Takeaway

This project helped in learning the use of LEDs for indicating the movement of PlutoX, monitoring the velocity values and plotting its graph as well. This can be used for debugging purposes in the future codes. For example, while programming the drone for performing a specific task such as rolling left and right based on certain input, program a specific LED for a particular movement. Suppose red LED is programmed with rolling right. During flight, if this task does not function properly, check whether the red LED is ON or not, which will indicate whether the system of PlutoX is rolling right or not. Correspondingly, the values for the same can be obtained on the monitor and can also be plotted on a graph for understanding the working and to rectify any bugs in the code i.e. debug the code.

Activity

This project was based on the movement of the drone in the vertical direction. When the drone moved upwards or downwards, the APIs confirmed the direction of movement by printing velocity values on the screen, plotting the graph, and by turning particular LEDs ON. When the drone is kept stationary, the value of velocity should ideally be zero. However, this is not always the case when it comes to a practical system.

During practical application, various parts of the system contain errors of different magnitudes. Due to these errors, there is a slight fluctuation in the output values. Hence, the value of vertical velocity will also fluctuate around the actual value. At rest, the value should be zero but there might be an error of ±5 or ±10.

Try to program the above project by taking into account this error range. Use different LEDs to confirm the movement of drone in upward direction, downwards direction and when at rest.

PROJECT 2: CHUCK TO ARM (Pluto1.2 & PlutoX)

Objectives

- The concept of free fall
- Working of an accelerometer

Problem statement

To arm PlutoX by tossing it in the air instead of arming it traditionally by keeping it on a flat surface.

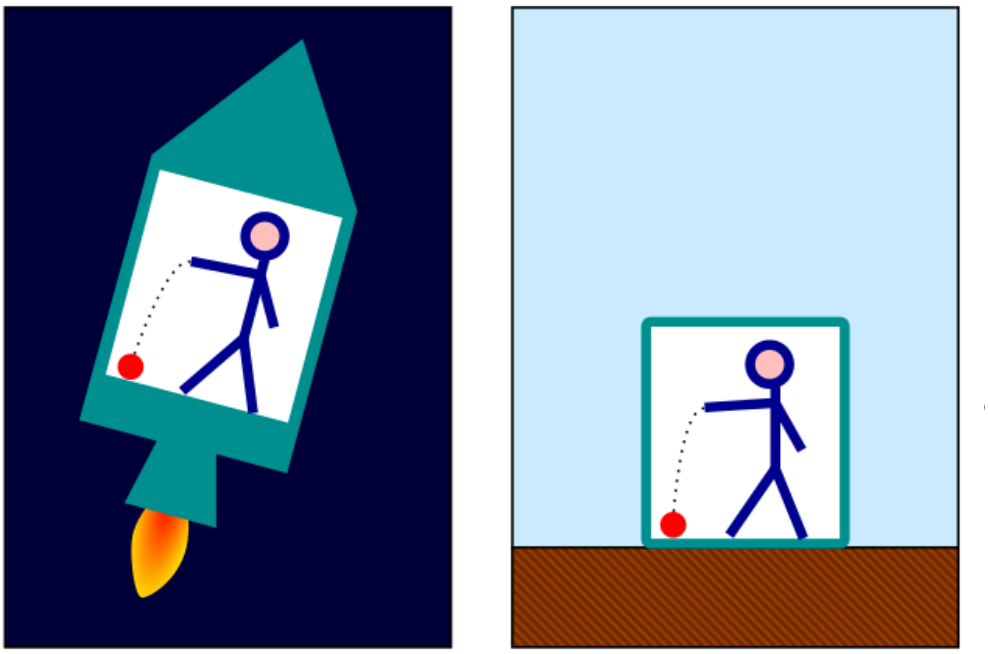
Explanation

Einstein's Happiest Thought: The Elevator Thought Experiment

In 1907, Albert Einstein came up with a thought experiment while observing some workers working at the top of a building near the patent office. He thought that if a worker fell from the top of a considerably tall building, he would not feel his weight. This thought gave birth to the famous Elevator Experiment.

Consider that you are in a closed elevator, with no contact with the outside world. You drop an apple inside the elevator. What would happen to the apple? Right, the apple would fall down on the floor. But more importantly, what must be the reason behind it? If your answer is gravity, then you are right and wrong, both.

The truth is that there is no way one can determine the actual reason behind it. The elevator could be here on Earth and it would be because of gravity that the apple falls down. However, theoretically, the elevator could also be in deep space far beyond the Earth or other bodies having strong gravitational pull and the apple could still fall down with the same rate as that on Earth, provided that the elevator is accelerating upwards by the same acceleration as the acceleration due to gravity on Earth, 9.81 m/s^2



Einstein's Elevator Thought Experiment[1]

Through this thought experiment, Einstein conveyed that gravity is nothing but an acceleration only. A body is accelerated towards Earth constantly by this acceleration. It is the floor or the ground which prevents it from falling further by providing an equal and opposite reaction force. Suppose a person weighing 100 kgs attempts to stand on a table made of cardboard kept on the ground. The cardboard will not be able to provide the required reaction force and hence will break, due to which the person will fall down i.e. accelerated downwards until he reaches to the ground. Thus, gravity is simply an acceleration which is acting continuously on us. This acceleration, if generated in deep space, would give us a feeling of gravity because we are aware of the concept of gravity.

Free fall

Now that we understand what gravity is, let us understand free fall.

Imagine that you are standing at the top of a building and you have a large sized watermelon and a tiny lemon. Now, you release both these objects from the top of the building towards the ground at the same time. Which of these two objects do you think will hit the ground first, assuming the wind resistance to be negligible?

[1] https:/*upload.wikimedia.org/wikipedia/commons/thumb/1/11/Elevator_gravity.svg/640px-Elevator_gravity.svg.png

If you guessed the watermelon, you clearly are not familiar with the concepts of free fall. When both the watermelon and lemon are dropped simultaneously, they are in free fall, since no other force other than the gravity acts on it. And for an object in free fall, the motion of the object can be easily predicted.

Assuming the mass of the object remains constant, and the size and speed of the object is not so small or so fast that it requires to consider relativistic effects, the motion of the object is described by Newton's second law of motion, force 'F' equals mass 'm' times acceleration 'a':

$$F = m * a$$

Solving for the acceleration of the object in terms of the net external force and the mass of the object:

$$a = F / m$$

For a free falling object, the net external force is just the weight of the object:

$$F = W$$

Substituting into the second law equation gives:

$$a = W / m = (m * g) / m = g$$

The acceleration of the object equals the gravitational acceleration. The mass, size, and shape of the object are not a factor in describing the motion of the object. So all objects, regardless of size or shape or weight, free fall with the same acceleration.

Thus both the watermelon and lemon fall on the ground at the same time.

This remarkable observation that all free falling objects fall with the same acceleration was first proposed by Galileo, nearly 400 years ago.

This video explains the above concept practically. Commander David Scott performed a live demonstration in which he dropped a heavy aluminium hammer and a falcon feather simultaneously on the surface of the moon. Because of the absence of the atmosphere on the moon, there is no air resistance and hence it provides the perfect environment to conduct the experiment. Check it out!

Hammer vs Feather - Physics on the Moon 

Remember: In Newtonian physics, a body is said to be under free fall if the only force acting on the body is gravity.

While in the process of tossing an object up in the air, there is a force acting on it because of the hand. However, as soon as the object leaves the hand and is tossed up, the only force acting on the object would be gravity. This means that even if a body moves upwards, and if there is no other force acting on it other than gravity, then it is under free fall. So do not let the name free 'fall' confuse you.

In conclusion, when an object is tossed in the air, it will be under free fall and hence the only force acting on it would be gravity and the acceleration would hence be the acceleration due to gravity.

Accelerometer

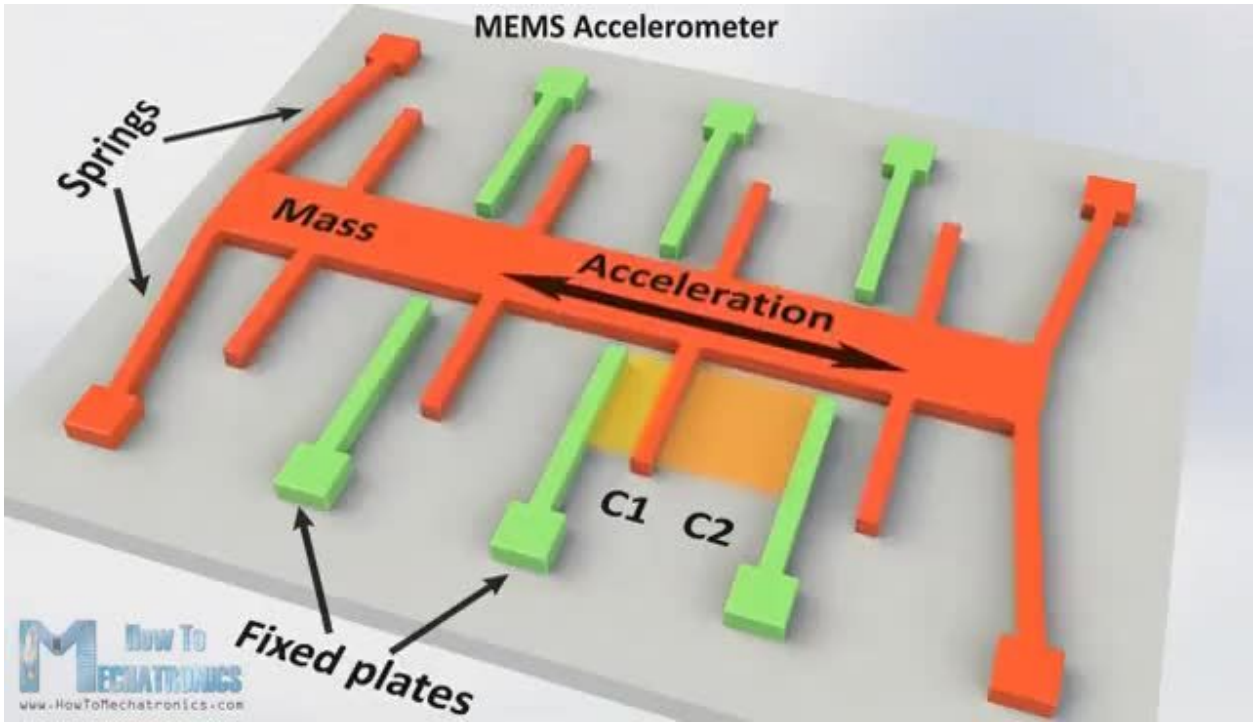
Accelerometer measures the value of Proper Acceleration. Proper acceleration of a body is defined as the acceleration of the body measured with respect to a freely falling object. It is the physical acceleration of the body.

It is now known that if an object is falling freely, it experiences acceleration due to gravity only. So it is safe to say that freely falling objects have downwards acceleration of g. Suppose an object A is freely falling. So it has a downwards acceleration of g. Another object nearby, object B is stationary. Since object B is not in motion, it must not have any acceleration as per Newton's Laws. While the object A is freely falling, if we observe object B from any point on object A, it would look like object B is moving upwards. This is a simple case of relative motion, which is often observed in daily life. However, the interesting point here is that since the object A is moving downwards with acceleration g, the object B will now appear to move upwards with the same acceleration g. This upwards g for a stationary object is called as perfect acceleration.

Thus, an accelerometer at rest will approximately measure 1g upwards. If an accelerometer is under free fall, its acceleration relative to a freely falling object is zero and hence it should measure zero g.

Working of an Accelerometer

Accelerometer used in drones are mems accelerometer. Mems stands for Micro Electro Mechanical Systems. A mems accelerometer is an electromechanical device used to measure acceleration forces. Such accelerations may be static, like the continuous acceleration of gravity or it can be dynamic, which occur due to movement or vibrations. The most famous types of accelerometer is the capacitive type. (high sensitivity high accuracy)



Working of a MEMS accelerometer[2]

A capacitor is made up of two conductive plates separated by a dielectric medium. In an Accelerometer, the movable plates (orange) and the fixed plates (green) act as the capacitor plates. The movable plates are supported by springs. When acceleration is applied, the proof mass moves accordingly. This movement of mass in turn moves the movable plates, which changes the distance between the movable and the fixed

2 https://www.digikey.pl/en/articles/techzone/2018/jan/apply-sensor-fusion-to-accelerometers-and-gyroscopes

plates. This change in distance produces a capacitance between the movable and the fixed plates. Capacitance is calculated as

$$C = \varepsilon A/d .$$

Where,

C = capacitance
ε = permittivity of dielectric
A = surface Area
d = distance between 2 plates

Thus changes occuring in 'd' can help to calculate capacitive changes. This variable is used in a circuit to ultimately deliver a voltage signal that is proportional to the acceleration.

Approaching the problem

The aim of this project is to arm PlutoX when it is tossed in the air. Looking at the entire process chronologically, the first step is to toss PlutoX in the air. When it is in the air, it will be under free fall as learnt earlier. And while under free fall, PlutoX should arm itself. Hence, the only requirement is to detect the moment when PlutoX is under free fall. This can be done with the help of Accelerometer. Accelerometer gives values for Proper Acceleration in each of the three axes and also the net acceleration downwards.

The logic would hence be, that when the net acceleration value reaches zero, the drone should arm. However, this basic logic will have certain flaws such as errors in the sensor. The flaw due to errors can be avoided by programming it to arm when the net acceleration value becomes less than 2 instead of reaching zero.

Another issue could occur because of crash. If PlutoX is crashing, it could be disoriented but it could still be under free fall and could get armed. This could prove to be dangerous. Hence, it should be programmed to arm only when it is not crashing.

To conclude, the required logic will be to arm the drone when the net acceleration value goes below 2 and PlutoX is not be crashed.

Creating a new project

- Create a new PlutoX Project.

- Name it as ‘Chuck to Arm’

Starting with the coding

Headers

The net acceleration value will be obtained from sensor block, arming the drone can be done from the user block and access to LEDs for debugging purposes will be obtained from Utils Block. Hence, we should include “Sensor.h”, “User.h” and “Utils.h” headers.

```
#include "PlutoPilot.h"
#include "Sensor.h"
#include "User.h"
#include "Utils.h"
```

void plutoInit ( )

No spare hardware needs to be initialized.

void onLoopStart ( )

As in Project 1, the default LED behaviour will be disabled to program them as per the requirements of the project.

```
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);     /*Disable default Led behavior*/
}
```

void plutoLoop ( )

The main user code will be programmed over here. The condition is that **if** the net acceleration value is approaching zero (less than 2) **and** PlutoX is not in crashed state, then PlutoX should arm itself. Both the conditions should be satisfied for it to arm. The net acceleration value will be obtained by using ‘Acceleration.getNetAcc’ API and the crash condition will be checked by using ‘FlightStatus.check’ API. To arm PlutoX, use ‘Command.arm ( )’ API. Set the red and green LEDs to turn ON when PlutoX arms itself.

```
void plutoLoop()
{
/*Add your repeated code here*/

        if(Acceleration.getNetAcc()<2&&(!FlightStatus.check(FS_CRASHED)))
/*Condition for free fall*/
        {
                Command.arm();  /*Arm the drone*/

                LED.set(RED, ON);
                LED.set(GREEN, ON);
        }
}
```

void onLoopFinish

When the developer mode is turned OFF, restore the default LED behaviour.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);  /*Enable the default LED behavior*/
}
```

The entire code

```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include "Sensor.h"
#include "User.h"
#include "Utils.h"


/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/

}
```

```
/*The function is called once before plutoLoop when you activate Developer Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

	LED.flightStatus(DEACTIVATE);	/*Disable default Led behavior*/
}


/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/

	if(Acceleration.getNetAcc()<2&&(!FlightStatus.check(FS_CRASHED)))
/*Condition for free fall*/
	{
		Command.arm();  /*Arm the drone*/

		LED.set(RED, ON);
		LED.set(GREEN, ON);
	}
}


/*The function is called once after plutoLoop when you deactivate Developer Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/

	LED.flightStatus(ACTIVATE);  /*Enable the default LED behavior*/
}
```

Running the project

- Build the project by using build command.
- Flash the code on PlutoX as in Project 1. Use Normal Flash since it is the second project. However, if any unusual behaviour is observed in PlutoX, then try Full Flash.

- Turn the developer mode ON.
- Toss PlutoX in the air to arm, i.e. chuck to arm PlutoX and fly!

Project Takeaway

This project focused on adding a new feature to PlutoX using which PlutoX can be armed by tossing it in the air. The first step in developing this feature was to analyze step by step the physics of tossing a body in the air. This gave an idea regarding the conditions and requirements for developing the feature. The condition was free fall and the requirement was to detect free fall using accelerometer values.

The next part was to develop the logic for the same and code it. Simple APIs were used to get the values of net acceleration and crash status of the flight. An API was used to arm the drone on satisfying the mentioned conditions. Knowledge from Project 1 was also used to detect the arming of PlutoX using LEDs.

It is important to understand that an Accelerometer is just a device that measures acceleration. It does not understand what gravity is. When PlutoX is kept on a flat surface, it measures approximately 1 g upwards, which is just a value of perfect acceleration for it at that instant.

In the elevator experiment, the person inside the elevator would not be able to differentiate between gravity and artificial acceleration. Accelerometer is like that person, unable to distinguish between gravity or any other acceleration.

So by understanding the physics of acceleration in various domains (like for chuck to arm used in free fall), one can develop different ideas using the Accelerometer on PlutoX.

Extra knowledge

The Earth pulls every object towards itself because of gravity. The value of this gravity is generally taken to be 9.81 m/s^2 However, this value changes from place to place since Earth’s gravitational force is diverging in nature. For any experiment to be conducted, a very small fraction of the Earth’s huge surface is used and hence the value of gravity can also be taken to be uniform for such a small place.

Brain teasers

Do you think there is gravity in space?
If you think there is no gravity in space, then you are wrong. A very small amount of gravity is present everywhere in space. This gravity is the reason why the Earth revolves around the Sun, Moon revolves around the Earth and other satellites revolve around the Earth.

The international space station (ISS) is at an altitude of 408km from the surface of the Earth. At that altitude, the gravity of Earth is still very strong. Thus, it is under free fall towards the Earth, constantly falling with acceleration g. But it also has a very high speed, about 27,600 km/hr because of which, it does not fall on Earth but overshoots it to orbit continuously.

The astronauts in the ISS are also under constant free fall towards the Earth. Free fall is very closely related to weightlessness. When a body is under free fall, the only force acting on it is gravity. Since there is no reaction force to counter 'g', the body will not feel any weight. Also, we know that free fall does not depend on the mass of the object. This means that the astronauts, the space station and all the equipments inside are all under free fall together and all of them fall with constant acceleration, which makes it look like they are floating.

PROJECT 3: FLY ACRO (Pluto1.2 & PlutoX)

Objectives

- Learning about gyroscopes
- Working of MEMS gyro sensor
- Different flight modes

Problem statement

Discovering different flight modes available in drones by using the basic knowledge of gyro sensors, and improving flying skills.

Explanation

In today's world we are surrounded by latest gadgets that have made our lives easier. One of the biggest boons of invention is the navigation system that has helped in getting rid of the fear of getting lost at unknown places. It has greatly helped in ship and aircraft navigation.

In as early as the 1700's, spinning devices were being used for sea navigation in foggy conditions. The more traditional spinning gyroscopes were invented in the early 1800's, and the French scientist Jean Bernard Leon Foucault coined the term gyroscope in 1852. In around 1916, the gyroscope found use in aircraft where it is still commonly used today. In the last ten to fifteen years, MEMS gyroscopes have been introduced and advancements have been made to create mass-produced successful products with several advantages over traditional macro-scale devices.

Gyroscopes, or gyros, are devices that can measure angular velocity or measure or maintain rotational motion. MEMS (microelectromechanical system) gyros are small, inexpensive sensors that measure angular velocity. The angular velocity measured has units degrees per second (°/s) or revolutions per second (RPS). Angular velocity is simply a measure of speed of rotation.

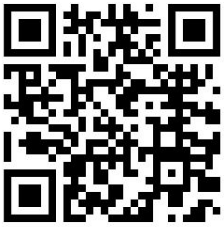
Working of a MEMS Gyro Sensor (Rate Gyro)
The working principle of a MEMS Gyro Sensor is different from the traditional Gyroscope. Traditional gyroscopes work on the principle of conservation of angular momentum whereas MEMS gyro sensors work on the principle of Coriolis Effect.

Coriolis effect occurs because of Coriolis Force, which is a pseudo force acting on a body in a rotating frame of reference.

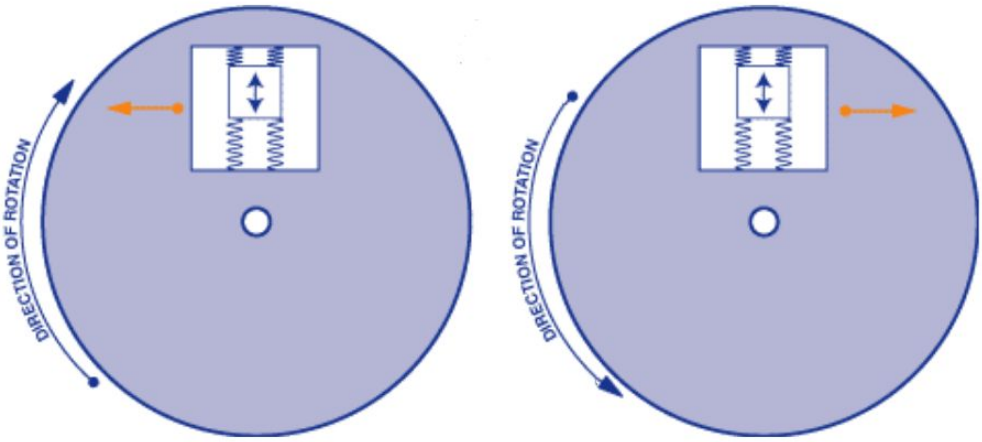
The Coriolis Force acts only in a rotating frame and is necessary to mathematically validate Newton's laws in a rotating frame of reference. When a body moves in a straight line in a rotating frame of reference, Coriolis Force acts on the body and makes it deviate from its original path. The direction of the deflection (left or right) depends on the direction of rotation of the rotating frame of reference. A body moving in anticlockwise frame of reference will be deflected towards the right of its direction of motion and a body moving in clockwise frame of reference will be deflected towards the left of its direction of motion. It is important to know that this deflection of a body will not be observed in an inertial frame of reference. In order to practically observe Coriolis Force and Coriolis Effect, refer to the following videos.

smartereveryday

Coriolis effect



The gyroscope sensor within the MEMS is very tiny (between 1 to 100 micrometers, the thickness of a human hair). It essentially contains a small mass resonating continuously. When the gyro is rotated, the small resonating mass is shifted to its left or right depending on the direction of rotation.



Working of a Gyro sensor[3]

The above diagram portrays two cases. In the first case, the direction of rotation is clockwise. When the mass moves upwards, it is deflected to its left. In the second case, the direction of rotation is anticlockwise. When the mass moves upwards in this case, it is deflected to its right. This movement is converted into very low-current electrical signals that can be amplified and read by a host microcontroller which is then used to calculate the angular velocity.

Importance of Gyro Sensors in drones

Gyro sensor is one of the most crucial sensors in drones. Gyro sensor detects the angular velocity across all three axes, i.e. it can detect the rate of change of angle in pitch, roll and yaw. The entire flight system depends on gyro sensor. It detects the current angles of orientation of the drone and also the angles required corresponding to the controller input. Since the Control Block eventually gives the output data in terms of desired rate, gyro sensors provide the required feedback on the corresponding inputs. It provides stability to drone during flight and prevents it from wobbling.

Flight Modes

Having the necessary knowledge regarding Gyro Sensors, let us dig deeper to understand the various flight modes available to fly a drone, which are all possible because of the gyro sensor.

Apart from the tinkering platform that PlutoX provides, PlutoX also offers a wonderful flying experience to all its users. People fly drones for a variety of purposes like

[3] https://learn.sparkfun.com/tutorials/gyroscope/all

hobby, aerial photography or to fulfill their childhood dream of flying. Different people have different flying skills, and hence would prefer different flying modes. Professional drone pilots love to fly drone in various flight modes to keep drone flying interesting and challenging.

Different flight modes available to fly the drones give pilots different types of assistance. When it comes to flying multirotors, generally there are two main flight modes used. One is self-level mode, and the other is acro mode.

Self Level Mode

Self-level mode is an assisted flight mode. When the roll and pitch stick are let go off, the drone returns to the neutral orientation. It's like there is an invisible hand bringing the drone to its neutral orientation at all times whenever it's not controlled by the pilot. In Betaflight and Cleanflight (Open-Source flight controller software), there are 2 different self-level modes: Angle mode and Horizon mode.

- Angle Mode

In Angle mode the stick controls the tilt angle of the drone on the pitch and roll axis. When the stick reaches its maximum position the drone will stop tilting further and hold there as it has reached the maximum tilt angle allowed. When the stick is released back to the centre, the angle of the drone will follow the stick back to horizontal level.

- Horizon mode

Similar to Angle mode, Horizon mode still works to keep the drone level when there are no inputs. But it also allows the pilot to do flips and rolls when the stick is at full deflection. However doing aerobatics in this manner feels more like a toy grade "push-button" flip system. This is where Rate mode comes in.

Rate mode / Acro mode

Rate mode is also known as Acro mode. In Acro mode, the pilot is controlling the drone's angular velocity of rotation with the stick, instead of the tilt angle. This means that if the pilot gives an input to pitch forward, the drone will pitch forward with the corresponding rate of change of angle, and if the pilot holds the stick at the same position, the drone will keep on pitching forward with the same rate instead of remaining at the same angle as in self-level mode. To return to its neutral orientation, the pilot would have to move the stick in the opposite direction.

Thus, Acro mode doesn't level the drone automatically, it will hold its roll and pitch orientation when the pilot lets go off the stick. Therefore the pilot would have to constantly make manual adjustments to keep the quadcopter from losing control and crashing into the ground.

To sum up the differences:

Angle / Horizon mode	Rate / Acro mode
● Suitable for beginners who are flying drone for the first time.	● For experienced pilots who want to challenge themselves.
● Once the pilot leaves the stick control, the drone automatically levels itself.	● To level the drone pilot needs to give opposite stick input.
● Pitch and roll inputs determine how far the drone will rotate on the given axis.	● Pitch and roll inputs determine how fast the drone rotates on the axis.
● In angle mode the pilot can only do basic maneuvers.	● In acro mode the pilot can perform acrobatics such as flips and rolls.
● It uses both Gyro and Accelerometer sensors.	● It uses only the Gyro sensor.
● When the pilot releases the stick back to centre, the angle of drone will follow the angle of the stick which creates some oscillations.	● Flight performance is more stable with less oscillations due to the fact that the accelerometer is disabled.

Approaching the problem

In this project, the tinkerers should try Acro mode or Rate mode. In order to do this, use a simple API 'FlightMode.set( )' to set the flight mode to the Rate mode. Different flight modes like Angle, Rate, Maghold, Althold, Throttle or Headfree mode can be set using the same API.

Creating a new project

- Create a new PlutoX project
- Name it “FlyAcro”

Starting with the coding.

Headers

To change the flight mode would mean to access the User Block, hence include “User.h” header. In order to use LEDs for indication purpose, include “Utils.h” header.

```
#include "PlutoPilot.h"
#include "User.h"
#include "Utils.h"
```

void plutoInit ( )

No spare hardware is required to be initialized.

void onLoopStart ( )

Deactivate the default LED behaviour as done in the earlier project.

```
void onLoopStart()
{
/*do your one time tasks here*/
        LED.flightStatus(DEACTIVATE); /*Disable default Led behavior*/
}
```

void plutoLoop ( )

The only task to be done in this project is to activate the Rate mode using the API “FlightMode.set ()”. Also, turn the red and blue LEDs ON and green LED OFF along with it.

```
void plutoLoop()
{
```

```
/*Add your repeated code here*/
        FlightMode.set(RATE); /*Change flight mode to Rate mode*/

        LED.set(RED, ON);
        LED.set(BLUE, ON);
        LED.set(GREEN, OFF);
}
```

void onLoopFinish

Restore the default LED behaviour.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/
        LED.flightStatus(ACTIVATE); /*Enable default Led behavior*/
}
```

The entire code

```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include "User.h"
#include "Utils.h"

/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/
}


/*The function is called once before plutoLoop when you activate Developer
Mode*/
void onLoopStart()
{
/*do your one time tasks here*/
        LED.flightStatus(DEACTIVATE); /*Disable default Led behavior*/
}
```

```
/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/
        FlightMode.set(RATE); /*Change flight mode to Rate mode*/

        LED.set(RED, ON);
        LED.set(BLUE, ON);
        LED.set(GREEN, OFF);
}


/*The function is called once after plutoLoop when you deactivate Developer
Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/
        LED.flightStatus(ACTIVATE); /*Enable default Led behavior*/
}
```

Running the project

- Build the project
- Flash the code on PlutoX
- Connect PlutoX with the Pluto Controller app and turn the Developer Mode ON
- Fly PlutoX in Rate Mode!

NOTE: Tinkerers are advised to fly PlutoX in an open room/space are while trying out Rate Mode. A small enclosed space will not give the best experience.

Project Takeaway

This project helped in understanding the working of a gyro sensor and its use in flight through Rate Mode. Flight mode can be changed by using the API “FlightMode.set()”. Using this project, a pilot can switch between the flying modes without having to disarm the drone by simply using the Developer Mode option. This can be useful during drone racing or for maneuvering.

PROJECT 4: PHONE CLONE (Pluto1.2 & PlutoX)

Objectives

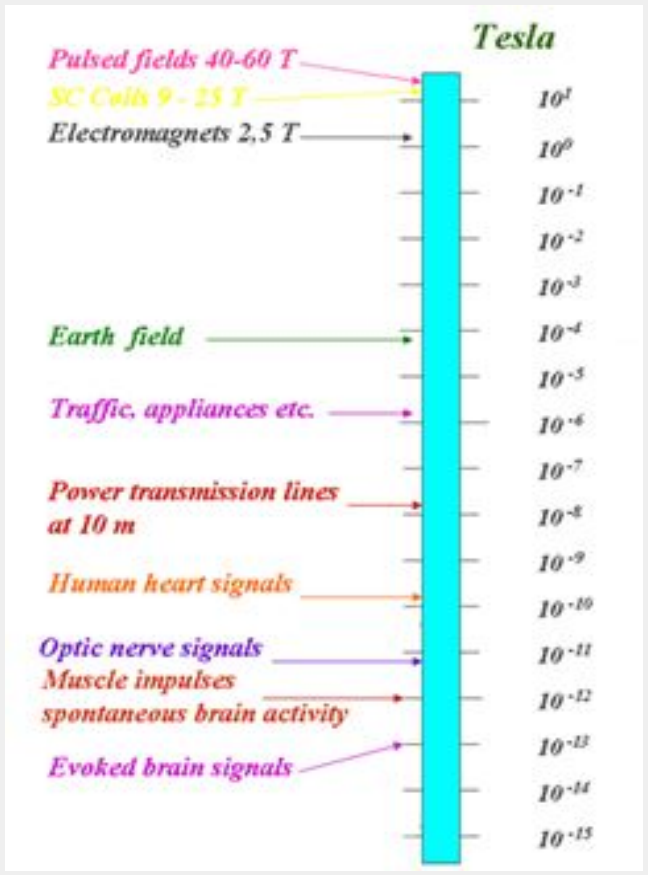
- Earth's Magnetic Field
- Working of a Magnetometer
- Learning Error Correction

Problem statement

To control the heading of PlutoX using the heading of the phone.

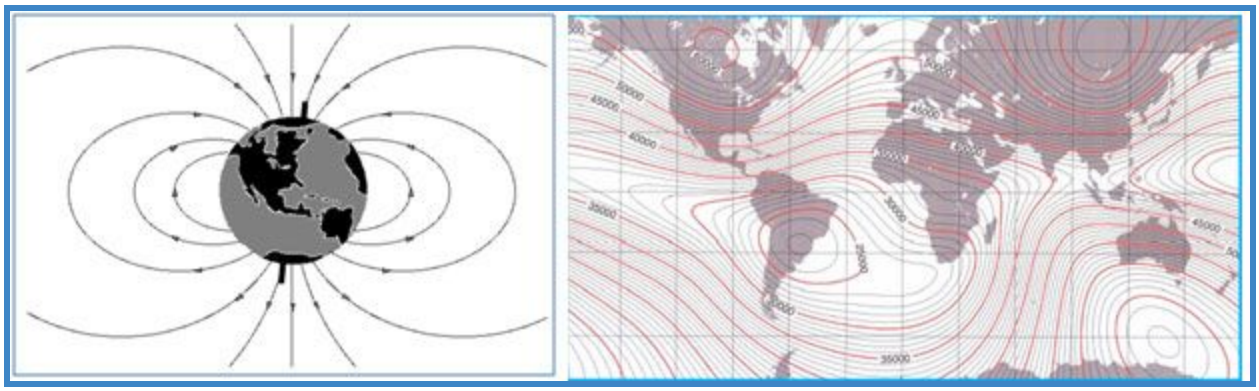
Explanation

A tinkerer must be having knowledge about the basic concepts such as magnets create magnetic fields; Earth has its own magnetic field; current flowing through a wire also generates magnetic field. But the tinkerer might not aware that magnetic fields are generated by our hearts and brains as well. However, what differs between the magnetic fields generated by a magnet and that by brain and heart is the magnitude of magnetic field. Following figure shows the magnitude of magnetic fields generated by various sources.



Magnetic fields generated by various sources

Magnetic fields are all around us. Be it electrical appliances like phones, computers or transmission lines. One of the most important magnetic fields around us is the Earth's magnetic field. It is relatively much smaller and varies around the globe. It is highest at the poles, about 60,000 nT and lowest at the equator, about 30,000 nT.



Measuring the magnetic field is of interest in various scientific purposes. However, the utmost important use of measuring magnetic field has been for navigation purposes. Since olden ages, people have used compass for navigation on ships. The compass was one of the earliest types of Magnetometer.

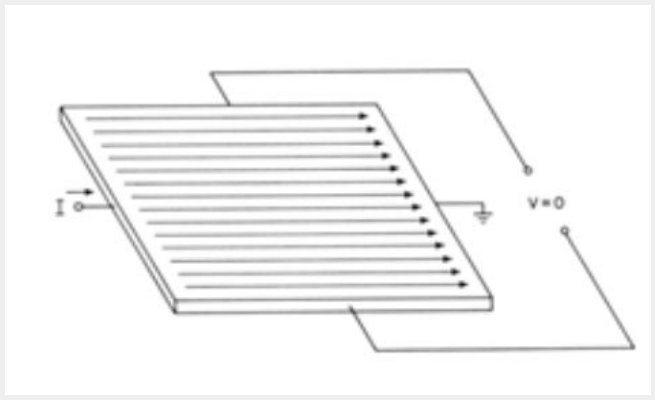
Magnetometers refer to sensors used for sensing magnetic fields or to systems which measure magnetic field using one or more sensors. Since magnetic flux density in the air is directly proportional to magnetic field strength, a magnetometer is capable of detecting fluctuations in the Earth's field.

Magnetometers have different working principles, such as rotating coil, Hall effect, fluxgate, etc. The most common type of magnetometer is the Hall Effect Magnetometer. The next part explains how a Hall Effect Magnetometer works.

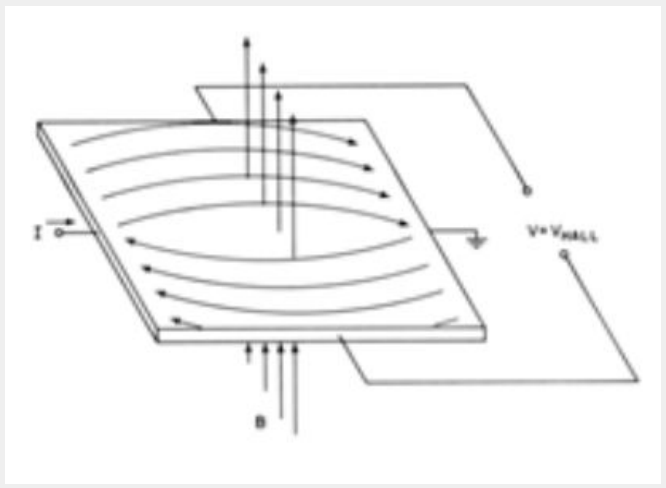
Working of a Hall Effect Magnetometer

The Hall Effect principle states that when a current carrying conductor is placed in a magnetic field, a voltage will be generated perpendicular to the direction of the magnetic field and the flow of current.

When a constant current is passed through a thin sheet of semiconducting material, the electrons would flow straight from one to the other side of the plate.



Now, if some magnetic field is brought near the plate, it would disturb the straight flow of the electrons and the electrons would get unevenly distributed on both sides. This would cause the formation of voltage across the plate.



That means if a Voltmeter is now connected across these two sides, it will get some voltage which depends on the magnetic field strength and its direction.

This principle is used to measure the Earth's magnetic field which gives the heading of the drone.

Magnetometer in Primus X

Primus X uses a 3 axis Magnetometer. The magnetometer aligns itself with the north of Earth's magnetic field and finds drone heading with respect to the Earth's North pole. But during this it faces 2 major issues.

- Magnetic field due to other components such as motors, batteries, etc.
- Earth's magnetic declination.

The magnetic field used for determining the heading is the Earth's magnetic field. In addition to the Earth's magnetic field, there are additional magnetic fields which cause interference. Interference can be caused by ferromagnetic materials or equipments in the magnetometers vicinity. Due to this interference, the sensor data becomes unreliable.

This is where the magnetometer calibration helps. For the magnetometer, magnetic fields due to motors and battery of the drone will always be the same even if the orientation of drone changes. Sensor readings can be corrected by simply removing the offset, which is done by magnetic calibration.

Approaching the problem

The aim of this project is to control the heading of PlutoX using the heading of the phone. This objective can be achieved by achieving two sub objectives.

- Getting the heading of the phone from the app.
- Setting the heading for PlutoX based on the phone heading.

While using the phone to fly PlutoX, the data regarding the orientation of the phone is stored in the User block. It is using this data that a pilot able to fly using tilt mode. This data also contains the data of the phone's Magnetometer. This Magnetometer data will be used as the heading for PlutoX.

However, there will be an error if the phone's raw data is directly used for PlutoX. This is because the orientation of the phone while flying, and the orientation of PlutoX will be some degrees (close to 90) apart. For example, if the heading of the phone is 90° while starting to fly, the heading of PlutoX could be 0° or 360° at that time. The error would be the difference between the heading of the phone and the heading of the drone, which comes down to 90°.

In order to remove this error, subtract 90° from the phone's heading. So, if 90° is subtracted from 90° as in the above example, it gives 0° or 360° which is the heading of PlutoX.

However, this could cause another issue in certain cases where the angle would become negative. For example, if the phone's heading is changed by 30° i.e. changed from 90° to 60°, then 60° - 90° will be -30°, which is negative. In such cases, add 360° and use it as the heading for PlutoX. So, -30° + 360° will be 330° hence PlutoX will change its heading by 30° only.

In a nutshell, the logic for this project will include getting the data of phone's heading from the app, correcting the data, specifying the condition if the sum goes beyond 360°, and setting this data as the heading for PlutoX.

Creating a new project

- Create a new PlutoX Project.
- Name it as 'Phone Clone'

Starting with the coding.

Headers

Data regarding the phone's heading is accessible from User Block. Hence, include "User.h" header. In order to set the heading for PlutoX, access to Control data will be needed. For this, include "Control.h" header. Use the LEDs and Pluto Monitor for indication purposes. Hence, include "Utils.h" header. Use the header "Estimate.h" to get the value of heading of the drone. Also declare a variable 'Error' to store the error value and 'PlutoXheading' to store the data of drone's heading.

```
#include "PlutoPilot.h"
#include "Control.h"
#include "User.h"
#include "Utils.h"
#include "Estimate.h"

int16_t Error=0, PlutoXHeading=0;
```

void plutoInit ( )

No spare hardware needs to be initialized.

void onLoopStart ( )

As in previous projects, disable the default LED behaviour to program them based on the requirements of the project. Calculate the difference between the heading of the phone using API "App.getAppHeading()" and the heading of the drone using the API "Angle.get(AG_YAW)". Store the value in "Error" and print it.

```
void onLoopStart()
{
/*do your one time tasks here*/
        LED.flightStatus(DEACTIVATE);  /*disable the default LED behavior*/

        Error = App.getAppHeading() - Angle.get(AG_YAW);
        Monitor.println("Error is: ", Error);
}
```

void plutoLoop ( )

In the main user code, print the value of the heading of phone, which will be accessed using the API “App.getAppHeading()”. Define the value of the heading for the drone by subtracting the “Error” from the heading of the phone. The heading of the drone is stored in “PlutoXheading”.

Apply the correction if the value of “PlutoXheading becomes negative. Print the value to which PlutoX should now turn to, i.e. print “PlutoXheading”. Set the desired angle of yaw for the drone to the value in “PlutoXheading”. This is done by using the API “DesiredAngle.set”. Print the value of angle of yaw at which PlutoX goes to, to confirm the execution of the project through data. Turn the red and green LEDs ON.

```
void plutoLoop()
{
/*Add your repeated code here*/
        Monitor.println("PhoneHeading: ", App.getAppHeading());

        PlutoXHeading = App.getAppHeading() - Error;
        if (PlutoXHeading < 0)
        {
                PlutoXHeading+=360;
        }
        Monitor.println("PlutoX should turn to ", PlutoXHeading);

        DesiredAngle.set(AG_YAW, PlutoXHeading);
        Monitor.println("PlutoX is at: ", Angle.get(AG_YAW));

        LED.set(RED, ON);
        LED.set(GREEN, ON);
}
```

void onLoopFinish

Restore the default LED behaviour.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);      /*Enable the default LED behavior*/
```

```
}
```

The entire code

```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include "Control.h"
#include "User.h"
#include "Utils.h"
#include "Estimate.h"

int16_t Error=0, PlutoXHeading=0;


/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/

}


/*The function is called once before plutoLoop when you activate Developer Mode*/
void onLoopStart()
{
/*do your one time tasks here*/
	LED.flightStatus(DEACTIVATE);  /*disable the default LED behavior*/

	Error = App.getAppHeading() - Angle.get(AG_YAW);
	Monitor.println("Error is: ", Error);
}


/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/
	Monitor.println("PhoneHeading: ", App.getAppHeading());

	PlutoXHeading = App.getAppHeading() - Error;
	if (PlutoXHeading < 0)
```

```
        {
                PlutoXHeading+=360;
        }
        Monitor.println("PlutoX should turn to ", PlutoXHeading);

        DesiredAngle.set(AG_YAW, PlutoXHeading);
        Monitor.println("PlutoX is at: ", Angle.get(AG_YAW));

        LED.set(RED, ON);
        LED.set(GREEN, ON);
}


/*The function is called once after plutoLoop when you deactivate Developer Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);      /*Enable the default LED behavior*/
}
```

Running the project

- Build the project by using build command.
- Flash the code on PlutoX.
- After connecting PlutoX with the phone, open the Pluto Controller app.
- In the Developer Settings, turn the ‘Send Heading’ switch ON.
- Turn the Developer Mode ON.
- While flying, change the heading of the phone and correspondingly observe the heading of PlutoX change.

Project Takeaway

The development of a new feature in this project allowed controlling the heading of PlutoX using the phone’s heading. This was done by using the data of phone’s heading and setting it as the input for the yaw angle of PlutoX. This should broaden the areas of thinking about new ideas to be implemented on PlutoX’s platform.

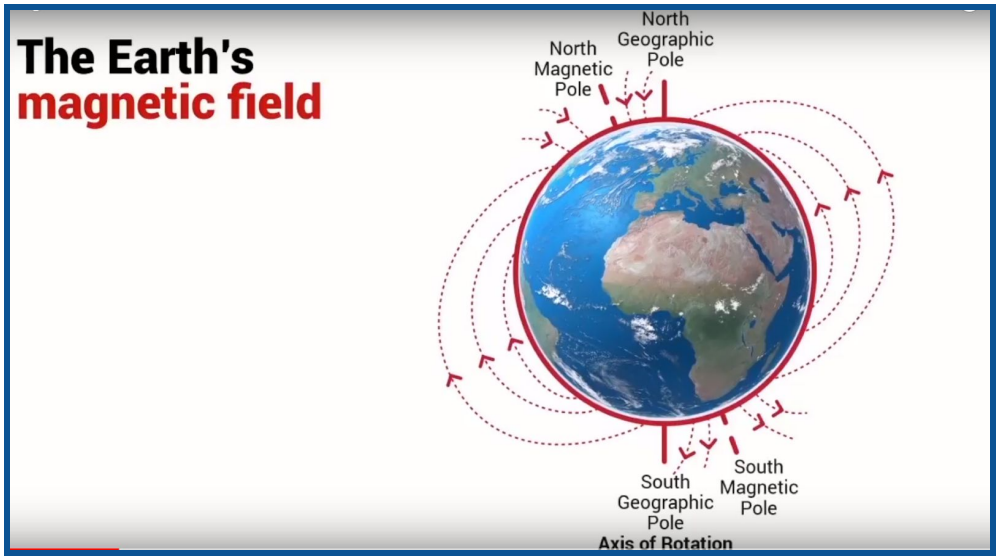
While developing the feature, it was encountered that using the values directly as input would have caused an error. The error was removed by using simple logical

solutions. In future as well, one could have such a logical approach towards solving such problems while programming.

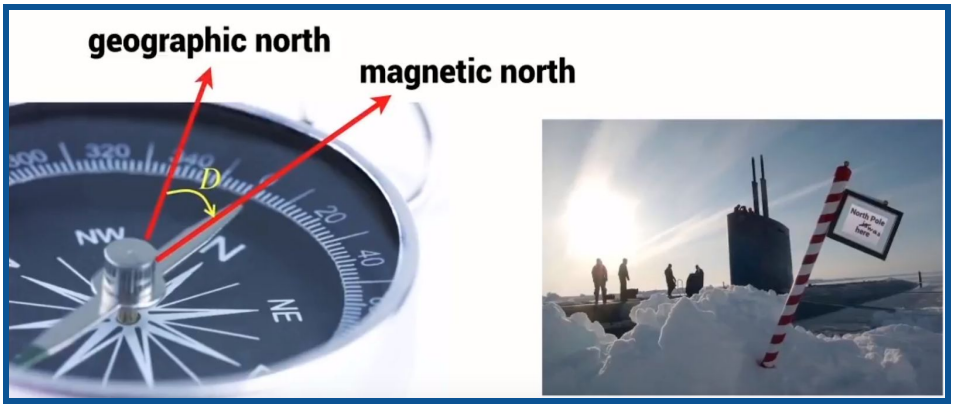
This project helped in understanding how to use external data also for programming purposes. Hence, the knowledge regarding the working of other devices could be utilized in developing more features on PlutoX.

Extra knowledge

When lost in the woods, the best chance of finding a way back might be a tiny magnet. A magnet is what makes a compass point north. The small magnetic pin in a compass is suspended so that it can spin freely inside its casing. The compass aligns itself with the Earth's magnetic field.



The Earth has some very uncanny behaviour in relation to its magnetic properties. The Earth rotates about an axis that passes through the North and the South Poles, known as the Geographic Poles. But our Earth itself is a gigantic magnet and the North and South Magnetic Poles do not align with the North and South Geographic Poles. The angle between the magnetic North and the Geographic North is the Magnetic declination angle.



The Magnetic Poles are continuously shifting from their positions. The Magnetic North Pole is slowly drifting across the Canadian Arctic, moving around 55 km or 34 miles each year in today's times in comparison to mid 1990's where it shifted by 15 km or 9.3 miles each year. This shifting eventually results into flipping of the Poles. It occurs about every 200,000 to 300,000 years and is a very slow gradual process and not a sudden one.

ACTIVITY

You must have heard about Aurora Borealis phenomenon. Did you know that it is caused because of the Earth's Magnetic Field? Find out more about it!

Aurora Borelais[4]

[4] https://www.techtimes.com/articles/240283/20190326/heres-why-the-aurora-borealis-did-not-appear-this-weekend.htm

PROJECT 5: OPEN SESAME (Pluto1.2 & PlutoX)

Objectives

- Working of a Barometer
- Using environmental changes in developing a feature

Problem statement

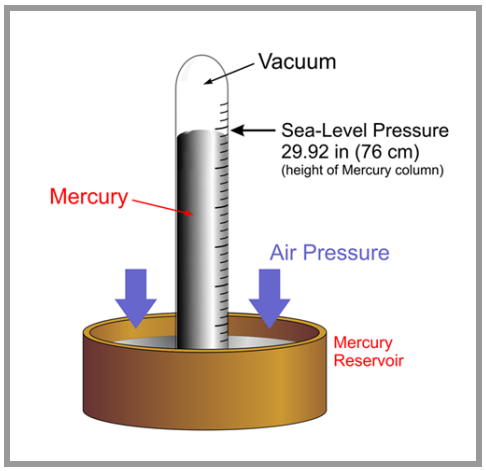
By making changes in the atmospheric pressure in the drone's surrounding observe the reading of Barometer and make the drone take off to a certain altitude.

Explanation

After understanding the working of Accelerometer, Gyro Sensor and Magnetometer and their purpose in drones, the next step is to understand the working and purpose of the next sensor, Barometer.

Basic Barometer

Barometer is an instrument which measures atmospheric pressure using mercury, water or air. Forecasters use changes in air pressure measured with barometers, to predict short-term changes in the weather.



To understand working of a basic barometer, refer the following video:

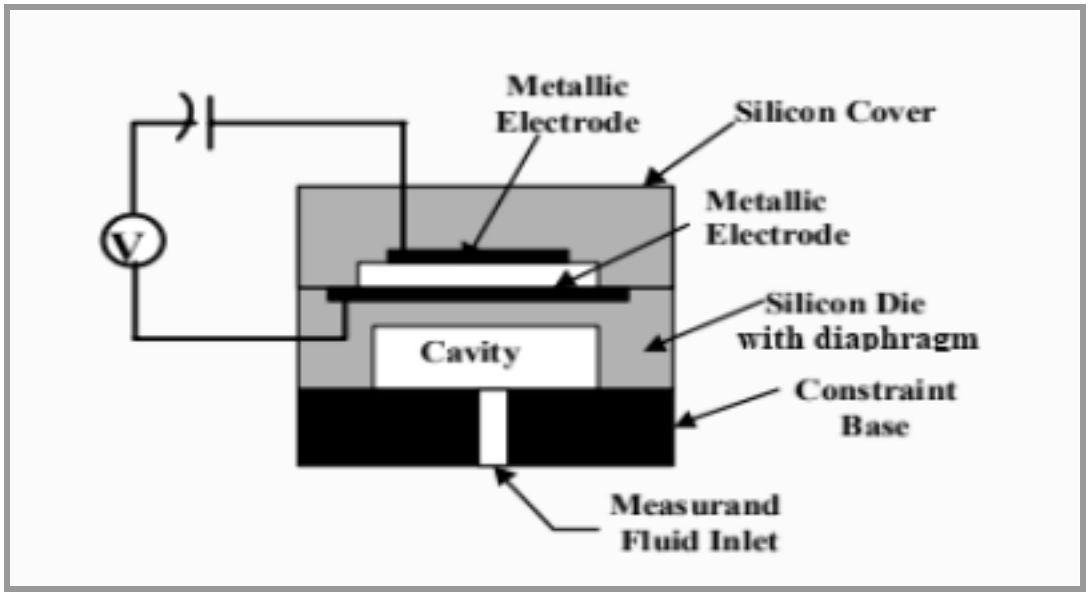
The history of the barometer (and how it works) - Asaf Bar-Yosef

Barometer in Drones

MEMS Barometer is used in drones to get the value of altitude through the atmospheric pressure. The relationship between atmospheric pressure and altitude is that the atmospheric pressure decreases at higher altitudes and increases at lower altitudes. Thus if an initial base value of altitude is set corresponding to the pressure measured, the system can calculate the increase and decrease of altitude corresponding to the decrease and increase in pressure. The data from Barometer helps in drone navigation to achieve desired altitude. Barometer is preferred over GPS because of its higher accuracy.

Working of MEMS Barometer

These pressure sensors work on the principle of mechanical bending of silicon diaphragm by gas pressure.



Mems barometer(pressure sensor)

The pressure in the cavity is taken as reference pressure. If the pressure outside increases more than reference pressure, the diaphragm bends inwards and when the pressure outside decreases as compared to the reference pressure, the diaphragm bends outwards. One way to convert the bending of diaphragm to a corresponding electronic output signals is the capacitive way. The bending of diaphragm in turn changes the distance between two metallic plates which affects the capacitance. We know that,

$$C = \varepsilon A/d$$

Thus any change in distance (d) between the metallic plates changes the capacitance. Using these values the system calculates the pressure changes happening in the surroundings.

Approaching the problem

The project aims at commanding PlutoX to turn ON and take off to a certain level when there is a considerable change in the atmospheric pressure around it. In order to achieve this, the first step is to create a considerable change in the atmospheric pressure. This can be done by closing or opening a door swiftly. Due to the sudden rush of air, the atmospheric pressure will change momentarily. This change in pressure when picked up by the system, should command PlutoX to turn ON and take off to a particular level.

Before beginning to develop the main user code, first develop a program to observe the changes in pressure when the door is opened or closed swiftly. Make a note of the difference in pressure readings which will then be utilized in the main user code.

Creating a new project

- Create a new PlutoX Project
- Name it as “Open Sesame”

Starting with the coding (To get pressure readings)

Headers

To check the pressure readings initially will require access to the Sensor Block, hence, include “Sensor.h”. To print the values on Pluto Monitor and plot a graph for the

same, include “Utils.h”. Also, declare a variable “baropressure” which will save the value of the atmospheric pressure obtained from the sensor.

```
#include "PlutoPilot.h"
#include "Sensor.h"
#include "Utils.h"


int32_t baropressure;
```

void plutoInit ( )

No spare hardware to be initialized.

void onLoopStart ( )

No tasks to be included here.

void plutoLoop ( )

In order to get the pressure value from Barometer, use “Barometer.get( )” API. This API can give the values of pressure and temperature, whichever is mentioned in the brackets. This value will be fed in “baroPressure” and it will be printed using the “Monitor.println( )” API and plotted on the red graph using “Graph.red( )” API

```
void plutoLoop()
{
/*Add your repeated code here*/

        baropressure=Barometer.get(PRESSURE);

        Monitor.println("Pressure", baropressure);
        Graph.red(baropressure);
}
```

void onLoopFinish

No tasks to be included here.

The entire code

```
/*Do not remove the include below*/
```

```
#include "PlutoPilot.h"
#include "Sensor.h"
#include "Utils.h"


int32_t baropressure;


/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/

}


/*The function is called once before plutoLoop when you activate Developer Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

}


/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/

        baropressure=Barometer.get(PRESSURE);

        Monitor.println("Pressure", baropressure);
        Graph.red(baropressure);
}


/*The function is called once after plutoLoop when you deactivate Developer Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/
```

```
}
```

Running the project (To get pressure readings)

- Build the code
- Flash the code on PlutoX
- Turn the Developer mode ON and connect Cygnus to PlutoX as well.
- Place PlutoX near the door and then open/close the door swiftly.
- Check the values of atmospheric pressure on Pluto Monitor and Pluto Graphs.
- Note the difference in pressure when the door is opened/closed.

Starting with the coding (Main User Code)

Headers

The data from Barometer will be accessible through “Sensor.h” header. To print the value of the pressure and also plot the graph, include “Utils.h” header. Commanding PlutoX to arm and take off will require access to User Block hence, include “User.h” header. Also, to set an altitude for PlutoX to reach after taking off, include “Control.h” header.

While calculating the difference between the pressure values, use an Absolute function, which will act as a modulus, so that the difference is always a positive number.

Also declare two variables “initPressure” to save the data of initial pressure and “currPressure” to save the data of the current pressure.

```
#include "PlutoPilot.h"
#include "Control.h"
#include "Sensor.h"
#include "User.h"
#include "Utils.h"
#include "Math.h"

int16_t initPressure;
int16_t currPressure;
```

void plutoInit ( )

No spare hardware to be initialized.

void onLoopStart ( )

As soon as the Developer Mode is turned ON, set the value of atmospheric pressure at that moment in the variable “initPressure” using the same API as before. Also, as done before, deactivate the default LED behaviour.

```
void onLoopStart()
{
/*do your one time tasks here*/

        initPressure=Barometer.get(PRESSURE);       /*Get Initial Temperature
when you turn the developer mode*/
        LED.flightStatus(DEACTIVATE); /*Disable the default led behavior*/
}
```

void plutoLoop ( )

In order to calculate the difference, first get the value of the current pressure. Save this value in the variable “currPressure” using the same API. Then, to arm the drone when the difference is a considerable one, include a condition using **if** statement. The condition will be that if the absolute of the difference between the initial and the current pressure is more than 8, then the next set of tasks should be executed. These next tasks are to turn the red LED ON, arm the drone using “Command.arm()” API and take PlutoX to an altitude (z axis) of 100 cms using the API “DesiredPosition.set()”.

If the condition in **if** statement is satisfied, PlutoX will take off to an altitude of 100cms. However, if the condition is not satisfied, then the program will continuously check for the condition. For this reason, update the initial pressure value in “initPressure” as it had saved the value initially only when the developer mode was turned ON. “initPressure” will be updated by adding the initial pressure and current pressure, both individually multiplied by their correction factors.

Simultaneously, print the values of initial and current pressures and plot a graph for current pressure.

```
void plutoLoop()
```

```
{
/*Add your repeated code here*/

        currPressure=Barometer.get(PRESSURE);      /*Get      pressure      values
continuously*/

        if(fabs(currPressure-initPressure)>8)   /*calculate    if    the    difference
between the current pressure and on start or initial pressure is greater than 8*/
        {
                LED.set(RED, ON);   /*turn on led to indicate if condition is true*/
                Command.arm();     /*arm the drone*/
                DesiredPosition.set(Z,100);          /*set the drone altitude to 100cms*/
        }

        initPressure=((initPressure*0.9)+(currPressure*0.1));

        Monitor.println("oldPressure", initPressure);
        Monitor.println("newPressure", currPressure);
        Graph.red(currPressure);
}
```

void onLoopFinish

Restore the default behaviour of the LEDs.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/
        LED.flightStatus(ACTIVATE);       /*Enable the default led behavior*/
}
```

The entire code

```
/* Do not remove the include below*/
#include "PlutoPilot.h"
#include "Control.h"
#include "Sensor.h"
#include "User.h"
#include "Utils.h"
#include "Math.h"
```

```
int16_t initPressure;
int16_t currPressure;

/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/* Add your hardware initialization code here*/

}


/*The function is called once before plutoLoop when you activate Developer Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

	initPressure=Barometer.get(PRESSURE);	/*Get Initial Temperature when you turn the developer mode*/
	LED.flightStatus(DEACTIVATE); /*Disable the default led behavior*/
}


/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/

	currPressure=Barometer.get(PRESSURE);	/*Get pressure values continuously*/

	if(fabs(currPressure-initPressure)>8)	/*calculate if the difference between the current pressure and on start or initial pressure is greater than 8*/
	{
		LED.set(RED, ON);	/*turn on led to indicate if condition is true*/
		Command.arm();	/*arm the drone*/
		DesiredPosition.set(Z,100);	/*set the drone altitude to 100cms*/
	}

	initPressure=((initPressure*0.9)+(currPressure*0.1));

	Monitor.println("oldPressure", initPressure);
```

```
        Monitor.println("newPressure", currPressure);
        Graph.red(currPressure);
}


/*The function is called once after plutoLoop when you deactivate Developer Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/
        LED.flightStatus(ACTIVATE);      /*Enable the default led behavior*/
}
```

Running the project

- Build the project
- Flash the code on PlutoX
- Get in a closed environment to conduct the experiment
- Connect PlutoX with Pluto Controller app and turn the Developer Mode ON.
- Connect Cygnus with PlutoX as well.
- Observe the value of current pressure on Pluto Monitor
- Open/close the door swiftly and observe the change in pressure on the monitor and graph.

NOTE: If the difference in pressure is not more than 8, then PlutoX will not take off. In such a case, change the value in the user code from 8 to a smaller value.

Project Takeaway

In this project, the first part was learning about the working of Barometer sensors and then programming PlutoX to arm when the atmospheric pressure changed by a considerable value. It is an example that environmental conditions can also be involved for the development of an idea. Different changes in environmental factors are sensed by particular sensors and they can be used to develop different features. To give an idea, a fire fighting drone could be developed using a chemical sensor or a temperature sensor.

Activities

- Set the required value of pressure difference to a very low value to observe the drone taking off even with a good strong blow near the Barometer.
- This project idea can be used for surveillance, in case somebody enters the room secretly, the drone will detect the pressure change and send a warning message or maybe it can start recording videos or take images of the room.

Did you know?

- We all know that the Earth is surrounded by a layer of air called the atmosphere. The gravitational force of Earth continuously pulls the atmosphere towards itself. Due to this, the atmosphere exerts a pressure on the surface of the Earth and the people on it. Did you know that at a given point of time an atmospheric pressure of 101 kPa acts on our body. Then why don't we get crushed by this force?

  This is because the air pressure present inside the body is equal to atmospheric pressure present outside the body. Thus they cancel each other out and we feel no force on our body. Our ears, nose, lungs, stomach contain air which balances the pressure inside and outside our body.

- Do you hear a pop sound when you enter a tunnel or when you are in an aeroplane? Why do you hear it?

  When the air pressure outside the body is equal to the air pressure inside the body we feel nothing strange. But when the pressure outside becomes lower than the inside pressure, the air gushes out of our body and when the air pressure inside becomes lesser than the outside air pressure, air rushes in. This movement of air causes the sudden popping sound.

PROJECT 6: TEMPERATURE REACTIVE DRONE (Pluto1.2 & PlutoX)

Objectives

- Relationship between temperature and pressure
- Using the environmental changes in developing ideas

Problem statement

To control the drone by changing the surrounding temperature.

Explanation

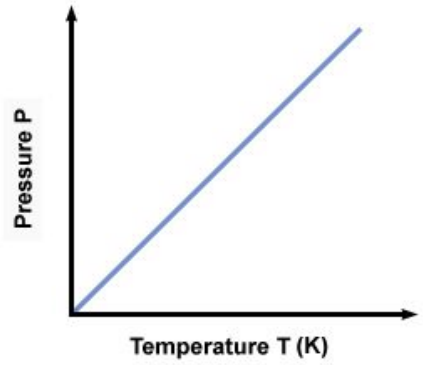
There is always a warning mentioned on the back of an aerosol can: Contents under pressure; do not heat. Why would heating this aerosol, be dangerous?

Aerosol cans contain liquid and gas. The gaseous particles are in continuous motion and keep on colliding with the walls of the container. When the container is heated, these gaseous and liquid particles absorb the heat energy, and get excited. This increases the speed of the gaseous particles and because of this they collide with the container walls with higher frequency and intensity. This increases the pressure inside the container. Also, the liquid particles absorb the energy and transform into gaseous state, which also contributes to the increase in the pressure inside the container. When the pressure increases to a very large extent, the container might not be able to withstand it and hence might explode. Hence, the warning.

Gay-Lussac's law

The above relationship between temperature and pressure is known as Gay-Lussac's law. It states that if the volume of a container is held constant as the temperature of

a gas increases, the pressure inside the container will also increase. As with the other gas laws, this one can be represented in the form of an equation: $P_1/T_1 = P_2/T_2$



$$\frac{P_1}{T_1} = \frac{P_2}{T_2}$$

As seen in the previous experiment, the atmospheric pressure can be measured using the Barometer Sensor. Barometers alone can do a fairly decent job of tracking low accuracy altitudes for short periods of time, but they are only really accurate within a foot (30cms).

In order to get very accurate data (upto 10 cms), one way is to combine a barometer with a temperature sensor. PlutoX contains a barometer module with an inbuilt temperature sensor.

The barometer module continuously takes values of pressure as well as temperature. The barometer performs its calculations to provide nearly accurate data for altitude. There are some predefined values of temperature which are mapped with pressure. Every particular set of pressure values are compared with a temperature value and its offset is calculated. This offset is subtracted to give precise altitude values.

Approaching the problem

This project is based on detecting the changes in temperature of the surrounding of drone. Program the drone to perform a particular task if the temperature changes. Since the aim through this project is to understand the working of the sensor, the task in this project would be to simply print the temperature values along with a message on the screen.

Once the Developer Mode is turned ON, note down the initial value of the temperature at that moment. After that, continuously monitor the value of

temperature and based on its relation with the initial value, print a message indicating whether it is hot, cold or same temperature.

Before beginning with the main user code, first develop a code to read the temperature values from the temperature sensor.

Creating a new project

- Create a new PlutoX project.
- Name it “TemperatureReactiveDrone”.

Starting with the coding. (To get temperature readings)

Headers

To get the readings from a sensor, it is necessary to include “Sensor.h” header. To print the values using Pluto Monitor and Pluto Graph, include “Utils.h” header. Declare a variable “temp” in which the temperature readings will be stored.

```
#include "PlutoPilot.h"
#include "Sensor.h"
#include "Utils.h"

int16_t temp;
```

void plutoInit ( )

No spare hardware to be initialised.

void onLoopStart ( )

No tasks to be performed here.

void plutoLoop ( )

The first task will be to get the values of temperature from the sensor and store it in “temp” variable. The values will be accessed by using “Barometer.get(TEMPERATURE)” API. Divide this value by 100 to get the value in degree celsius. Print these values using “Monitor.println()” API and plot the values on red graph using “graph.red()” API.

```
void plutoLoop()
{
/*Add your repeated code here*/

        temp = Barometer.get(TEMPERATURE)/100;    /*Get current Temperature*/
        Graph.red(temp, 1);
        Monitor.println("Temperature", temp);
}
```

void onLoopFinish

No tasks to be performed here.

The entire code

```
/* Do not remove the include below*/
#include "PlutoPilot.h"
#include "Sensor.h"
#include "Utils.h"

int16_t temp;


/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/* Add your hardware initialization code here*/

}

/*The function is called once before plutoLoop when you activate Developer Mode*/
void onLoopStart()
{
/* do your one time tasks here*/

}

/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/
```

```
        temp = Barometer.get(TEMPERATURE)/100;    /*Get current Temperature*/
        Graph.red(temp, 1);
        Monitor.println("Temperature", temp);
}

/*The function is called once after plutoLoop when you deactivate Developer Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/

}
```

Running the project (To get temperature readings)

- Build the project
- Flash the code on PlutoX
- Connect PlutoX with the app and with Cygnus
- Observe the values of temperature on the screen

Starting with the coding. (Main user code)

Headers

Again, data will be required from sensors. Hence, include “Sensor.h” header. To control the LEDs and to print and plot the values of sensor, include “Utils.h” header. Initialize two variables, “initemp” and “currtemp” to store values of the initial temperature and the current temperature.

```
#include "PlutoPilot.h"
#include "Utils.h"
#include "Sensor.h"

int16_t initemp; /*initialize a variable to save initial temperature value*/
int16_t currtemp; /*initialize a variable to save current temperature values*/
```

void plutoInit ( )

No spare hardware to be initialized.

void onLoopStart ( )

Deactivate the default LED behavior and turn the blue LED ON here. Also, the initial temperature reading will be taken in this part of the code since it will be needed only once. It will be done by using the "Barometer.get( )" API and stored in "initemp" variable. This value will also be printed once.

```
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable the default led behaviour*/
        LED.set(BLUE, ON);
        initemp= Barometer.get(TEMPERATURE)/100; /*Get initial temperature
when you turn ON the developer mode */
        Monitor.print("initemp", initemp);
}
```

void plutoLoop ( )

Here, get the value of the current temperature and store it in "currtemp" variable and then print and plot them on the screen. These values will be compared to the "initemp" value and based on their relation, the program will display a specific message. This code can be developed using **if else** loop.

If the value of current temperature is more than the initial temperature value, then print "HOT" and turn the red LED ON. Else, if the value of current temperature will be less than the initial temperature value, then print "COOL" and turn the green LED ON. Else, the temperature values will be equal and hence we will print "EQUAL" and turn the blue LED ON.

```
void plutoLoop()
{
/*Add your repeated code here*/

        currtemp= Barometer.get(TEMPERATURE)/100;       /*Get
current Temperature*/

        Graph.red(initemp, 1);
        Graph.green(currtemp, 1);
```

```
        Monitor.println("Currenttemp", currtemp);


        /*Check the relation between initial and current temperature and turn on
LED for indicating different conditions*/
        if (currtemp>initemp)
        {
                LED.set(RED, ON);
                LED.set(BLUE, OFF);
                Monitor.println("HOT");
        }


        else if (currtemp<initemp)
        {
                LED.set(GREEN, ON);
                LED.set(BLUE, OFF);
                Monitor.println("COOL ");
        }


        else
        {
                Monitor.println("EQUAL");
                LED.set(BLUE, ON);
                LED.set(RED, OFF);
                LED.set(GREEN, OFF);
        }
}
```

NOTE: it is important to turn the other LEDs OFF because if they are turned ON in some earlier part of the code, they will remain in ON state until turned OFF.

void onLoopFinish

Restore the default LED behavior.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE); /*Enable the default led behavior*/
}
```

The entire code

```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include "Utils.h"
#include "Sensor.h"

int16_t initemp;
int16_t currtemp;

/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/

}


/*The function is called once before plutoLoop when you activate Developer Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable the default led behaviour*/
        LED.set(BLUE, ON);
        initemp= Barometer.get(TEMPERATURE)/100; /*Get initial temperature when you turn ON the developer mode */
        Monitor.print("initemp", initemp);
}

/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/
```

```
        currtemp= Barometer.get(TEMPERATURE)/100;       /*Get                current
Temperature*/

        Graph.red(initemp, 1);
        Graph.green(currtemp, 1);
        Monitor.println("Currenttemp", currtemp);


        /*Check the relation between initial and current temperature and turn on
LED for indicating different conditions*/
        if (currtemp>initemp)
        {
                LED.set(RED, ON);
                LED.set(BLUE, OFF);
                Monitor.println("HOT");
        }


        else if (currtemp<initemp)
        {
                LED.set(GREEN, ON);
                LED.set(BLUE, OFF);
                Monitor.println("COOL ");
        }


        else
        {
                Monitor.println("EQUAL");
                LED.set(BLUE, ON);
                LED.set(RED, OFF);
                LED.set(GREEN, OFF);
        }
}

/*The function is called once after plutoLoop when you deactivate Developer
Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/
```

```
        LED.flightStatus(ACTIVATE); /*Enable the default led behavior*/
}
```

Running the project

- Build the project
- Flash the code on PlutoX
- Connect PlutoX with the app and also with Cygnus
- Turn the Developer Mode ON
- Observe the initial and current temperature readings on Pluto Monitor and Pluto Graph.

NOTE: Try to change the temperature by taking PlutoX in different temperature areas like near AC and in room temperature.

Project Takeaway

This project helped in understanding the workings of a Barometer Sensor in obtaining the value temperature. The primary objective of this project, just like the project 'Open Sesame' was to understand how environmental changes can be utilized for developing an idea. This project printed certain messages on screen based on the changes in temperatures.

Using the structure of this project, one could develop a drone which would detect temperatures at a volcano and help a team of volcanologists rather than the volcanologists being exposed to the environment.

Fun Facts

- 56.7 °C (134 °F) is the hottest temperature ever recorded on Earth. It was recorded on 10 July, 1913 at Greenland Ranch, Death Valley, California, USA.
- −89.2 °C (−128.6 °F) is the coldest temperature ever recorded on Earth. It was recorded at Vostok Station in Antarctica on July 21, 1983.

Brain teasers

- Can humans actually feel the temperature of an object?

  It is always assumed that the skin can sense the temperature of the object it touches. But in reality, the thermoreceptive nerves inside the skin can only

detect the temperatures of the skin and not of the object it touches. When the skin touches an object, heat transfer takes place. The heat could be transferred from the skin to the object if the object is at a lower temperature and could be transferred from the object to the skin if the object is at a higher temperature. Based on this heat transfer, both in terms of the amount and rate, the thermoreceptive nerves will sense the temperature change and will send a signal to the brain, which would conclude the temperature of the object touching the skin.

Thus, it is the rate of heat transfer that decides whether we perceive the object as hot or cold.

Try and sense the temperature of a metal slab and a book placed in the same environment. It would feel that the metal slab is much cooler than the book but in reality both of them are at the same temperature!

- How do we feel the heat of the sun even when it is 14,96,00,000 km away from earth?

  The surface of the Sun is at a temperature of 5,778 K. The Sun is extremely hot and it radiates its energy in all directions. This radiated energy is in the form of photons which travel at the speed of light. Since the space is largely empty, there is very less obstruction to the radiation directed towards the Earth. Out of the 3.86×10^{26} Watts that the Sun produces every second, only 1.74×10^{17} watts reaches the Earth. Knowing the speed of light and the distance of the Sun from the Earth, it can be easily calculated that it takes about 8 minutes for light to reach the Earth. So the light which we are receiving from the Sun right now was radiated 8 minutes ago!

PROJECT 7: TURTLE TURN (PlutoX)

Objectives

- Thrust
- Aerofoil design

Problem statement

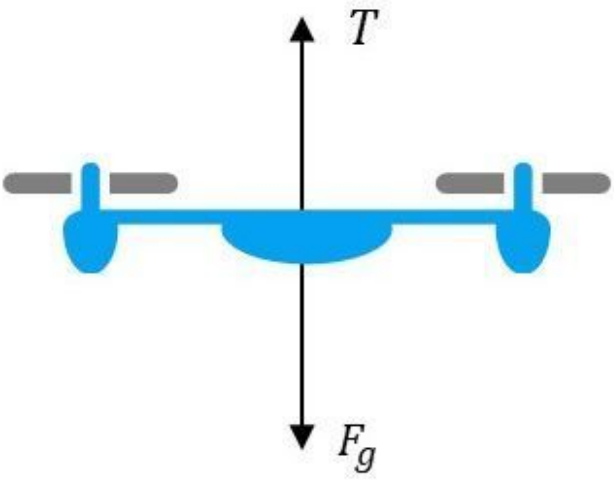
Turn the drone upright from its flipped orientation.

Explanation

Turtles have a hard time getting right back up once they get flipped upside down. Flat shelled turtles right themselves by using a technique reminiscent of classic breakdancing moves. They extend their neck and pivot on their head while pushing off with their leg on whichever side is tilting downward. Drones face the same problem as these turtles.

When a person begins flying a drone, it is bound to crash numerous times. Even while tinkering, the drone will crash a number of times. Many times, the drone crashes at an angle or even completely flipped on its top. In such cases, it becomes a task to go to the crash site and flip it back with hands. This project aims at removing this inconvenience by enabling the drone to flip back on its own so that the user can then fly it directly.

It is necessary to understand the physics behind the working of the propulsion system in order to develop this feature. The propulsion system is a combination of motors and propellers. A quad copter has four propellers, two of them being clockwise and the other two being anti clockwise propellers. This combination is essential to eliminate the issue of reactive torque. When these propellers spin in their defined direction, the air is thrown downwards and the drone is pushed upwards just as Newton's third law of motion suggests. The force which lifts the drone upwards is called Thrust.



Thrust in a drone[5]

One of the primary factors on which the thrust generated depends upon is the aerofoil design of the propellers. The below diagram shows a cross section of an aerofoil.

Cross section of an aerofoil[6]

Thrust generation due to aerofoil design takes place by a combination of conservation of mass, conservation of angular momentum and conservation of energy. In order to

[5] https://www.droneomega.com/drone-motor-essentials/

[6] http://www.ppl-flight-training.com/chord-line.html

understand how they all work together, check out this video which explains about the generation of lift by aerofoil.

Lift and Wings - Sixty Symbols

Imagine a drone flipped upside down after crashing. If the drone is armed in this orientation, the propellers would still spin in their own predefined direction. Because of this, the air would now be pushed upwards with respect to the orientation of the drone and this would not provide the required thrust. In order to achieve the required objective of flipping the drone, the propellers will have to spin in their respective opposite direction so that the air will be pushed downwards and the thrust provided in the upward direction.

The direction of rotation of the propellers can be changed by changing the direction of rotation of the motors on which they are mounted. Thus, for this project, the reversible motor drivers M1 - M4 will be used instead of the unidirectional motor drivers M5 - M8.

It must be noted that if the propellers are rotated in the opposite direction to what they are actually designed for, it causes a decrease in the thrust generated and hence a decrease in efficiency. This is because while rotating in the opposite direction, the trailing edge and the leading edge exchange their roles. While the trailing edge leads, it is very thin as compared to the other edge.

Aerofoil in reverse direction[7]

When the airflow hits the trailing edge first, it does not completely move along the surface of the propeller and is turned slightly upwards. Hence the overall angular momentum is not directed downwards properly as it would have if the propeller would have spun in its original direction. This causes a decrease in thrust and hence the loss in efficiency. However, the amount of thrust generated will be enough for PlutoX to flip back to its original orientation.

Approaching the problem

For this project, motor drivers M1 - M4 will be used which have to be initialized. Since these motor drivers are bidirectional, the direction in which they should rotate have to be defined within the program. The propellers have to spin in the opposite direction to what they are designed to spin, thus, the clockwise propellers will have to spin in anti clockwise direction and vice versa. This helps to define the direction of rotation for motors, for example, M1 will rotate in anti-clockwise direction since that particular propeller has to be rotated in anti-clockwise direction.

This feature should be executed when the drone is disoriented or flipped upside down. To check this condition of the drone, first task will be to get the data of current angle of roll of the drone. Next, as a precaution, the program should also check if the drone is not armed. When the drone is not armed, check the current angle of roll of the drone. The roll angle has a range of -1800 to 1800, the units being decidegrees. If the drone is tilted, it could be between either 0 to 1800 decidegrees or 0 to -1800 decidegrees. Thus, to eliminate the factor of positive and negative angles, use absolute function which will act as modulus. So, if the absolute of the angle of roll is more than 80°, then give full power to two motors (M1, M2 or M3, M4). This combination of motors is used because roll angle is selected to initiate the feature. If the absolute of the angle of roll is not more than 80° then the power given to all the motors is minimum.

Creating a new project

- Create a new PlutoX project
- Name it “TurtleTurn”

---

7 https://aviation.stackexchange.com/questions/33650/how-would-an-airfoil-react-if-it-was-flown-backwards

Starting with the coding.

Headers

This project will require the angle of roll of the drone, hence include the header “Estimate.h”. In order to control the power to the motors, include the header “Control.h”. Include the header “Motor.h” to access the reversible motor drivers. To get the status arming of drone, include “User.h” header. Include the header “Utils.h” for debug purposes.

Also declare a variable “angle” to store the roll angle values and define the absolute function.

```
#include "PlutoPilot.h"
#include "Control.h"
#include "Estimate.h"
#include "Motor.h"
#include "Utils.h"
#include "User.h"

int16_t angle;

#define ABS(x) ((x) > 0 ? (x) : -(x))
```

void plutoInit ( )

The reversible motor drivers need to be initialized. Use the API “Motor.initReversibleMotors()” to so the same.

```
void plutoInit()
{
/*Add your hardware initialization code here*/

        Motor.initReversibleMotors();
}
```

void onLoopStart ( )

Deactivate the default LED behavior. Set the direction of rotation of the motors in this loop as the system will not require the motors to rotate in the reverse direction

after the drone flips back to the correct orientation. Set the direction of rotation by using “Motor.setDirection()” API.

```
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable default LED Behaviour*/

        /*Reverse the motor direction*/
        Motor.setDirection(M1, ANTICLOCK_WISE);
        Motor.setDirection(M2, CLOCK_WISE);
        Motor.setDirection(M3, ANTICLOCK_WISE);
        Motor.setDirection(M4, CLOCK_WISE);
}
```

void plutoLoop ( )

The core logic will begin by getting the angle of roll of the drone. Access the values using “Angle.get()” API and store the values in the variable “angles” and print them.

Next, use **if** condition to check the arming of the drone. The API “FlightStatus.check(FS_ARMED)” is used to check whether PlutoX is armed. Since the condition should work if PlutoX is not armed, use the negation of the value, i.e. “!FlightStates.check(FS_ARMED)”. The “!” mark will return the negation of the status value. Thus, if PlutoX is armed, it will return a negative value and the condition will not be satisfied; if PlutoX is not armed, it will return a positive value and the condition will be satisfied, and the next tasks inside the loop will be executed.

The next task is to check the angle of roll using **if else** condition. If the absolute value of the angle of roll or “angle” variable has a value greater than 800 decidegrees, then the motors M1 and M2 should be given full power. The power given to the motors has a range of 1000 to 2000 where, the maximum power corresponds to 2000. The power to the motor is set using the API “Motor.set()”. In case the absolute of the roll angle is not greater than 800 decidegrees, then give the least power to the motors, i.e. 1000.

```
void plutoLoop()
{
/*Add your repeated code here*/
```

```
        angle=Angle.get(AG_ROLL);       /*read current angle value*/
        Monitor.println("Angle: ", angle);

        if(!FlightStatus.check(FS_ARMED))       /*check if drone is armed*/
        {
                if (ABS(angle)>800)  /*checks if the inverted*/
                {
                        /*set the motor input to max*/
                        Motor.set(M1, 2000);
                        Motor.set(M2, 2000);
                }
                else
                {
                        Motor.set(M1, 1000);
                        Motor.set(M2, 1000);
                        Motor.set(M3, 1000);
                        Motor.set(M4, 1000);
                }
        }
}
```

void onLoopFinish

Restore the default LED behavior. Set the power to all the motors to minimum. This is to stop the spinning of propellers after the drone is brought back to its original orientation. Also, set the motor directions back to the default so that the user can fly the drone after arming it.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);      /*Enable the default LED behavior*/

        /*set motor value to default*/
        Motor.set(M1, 1000);
        Motor.set(M2, 1000);
        Motor.set(M3, 1000);
        Motor.set(M4, 1000);
```

```
        /*set motor directions to default*/
        Motor.setDirection(M1, CLOCK_WISE);
        Motor.setDirection(M2, ANTICLOCK_WISE);
        Motor.setDirection(M3, CLOCK_WISE);
        Motor.setDirection(M4, ANTICLOCK_WISE);
}
```

The entire code

```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include "Control.h"
#include "Estimate.h"
#include "Motor.h"
#include "Utils.h"
#include "User.h"

int16_t angle;

#define ABS(x) ((x) > 0 ? (x) : -(x))

/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/

        Motor.initReversibleMotors();
}

/*The function is called once before plutoLoop when you activate Developer Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable default LED Behaviour*/

        /*Reverse the motor direction*/
        Motor.setDirection(M1, ANTICLOCK_WISE);
        Motor.setDirection(M2, CLOCK_WISE);
        Motor.setDirection(M3, ANTICLOCK_WISE);
        Motor.setDirection(M4, CLOCK_WISE);
```

```
}

/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/

        angle=Angle.get(AG_ROLL);       /*read current angle value*/
        Monitor.println("Angle: ", angle);

        if(!FlightStatus.check(FS_ARMED))       /*check if drone is armed*/
        {
                if (ABS(angle)>800)  /*checks if the inverted*/
                {
                        /*set the motor input to max*/
                        Motor.set(M1, 2000);
                        Motor.set(M2, 2000);
                }
                else
                {
                        Motor.set(M1, 1000);
                        Motor.set(M2, 1000);
                        Motor.set(M3, 1000);
                        Motor.set(M4, 1000);
                }
        }
}

/*The function is called once after plutoLoop when you deactivate Developer Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);     /*Enable the default LED behavior*/

        /*set motor value to default*/
        Motor.set(M1, 1000);
        Motor.set(M2, 1000);
        Motor.set(M3, 1000);
        Motor.set(M4, 1000);
```

```
        /*set motor directions to default*/
        Motor.setDirection(M1, CLOCK_WISE);
        Motor.setDirection(M2, ANTICLOCK_WISE);
        Motor.setDirection(M3, CLOCK_WISE);
        Motor.setDirection(M4, ANTICLOCK_WISE);
}
```

NOTE: While performing this experiment, remove the prop guards.

Running the project

- Build the project
- Flash the code on PlutoX
- Ensure that the motors are connected to ports M1 - M4
- Connect PlutoX to the PlutoController app
- Remove the prop guards and keep the drone upside down
- Turn the Developer Mode ON
- The drone will perform the turtle turn and will stand upright ready to be armed and flown.

Project Takeaway

This project is an example of solving problems or inconveniences faced in drone operations using the platform that PlutoX provides. Using the available reversible motor drivers, the propellers were made to rotate in the reverse direction which generated thrust to flip the drone back to the default orientation. Moreover, after the problem is solved, the drone is ready to be flown again after arming it.

Looking at the normal flight operations of a drone and visualising the unlimited ideas that can be executed on drone, this project aims at providing the basic knowledge of understanding the inconveniences and solving them. With more such ideas programmed into a drone, it can provide the best user experience along with an increased scope of practical applications.

PROJECT 8: X RANGER (PlutoX)

Objectives

- Ranging Sensor VL53L0X (ToF)
- Integrating X-Ranging along with VL53L0X on Primus X
- Reading values from X-Ranging

Problem statement

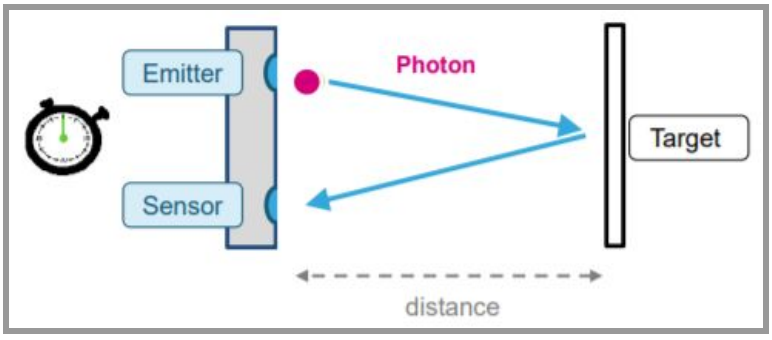
To install X-Ranging board and Ranging Sensor (VL53L0X) on Primus X and read data from the sensor.

Explanation

The previous projects were aimed at understanding the workings of the inbuilt sensors of PlutoX and trying to execute ideas using them. A lot more can be done using PlutoX as there are a lot of sensors that can be integrated with PlutoX. This project focuses on integrating the Ranging Sensor VL53L0X.

VL53L0X

VL53L0X is a Time-of-Flight (ToF) laser-ranging module using I2C communication. It can measure absolute distances upto 2m. The sensor emits photons, which are partially reflected from the target. These reflected photons are received by the sensor receiver which then determines the amount of time taken by the photons to return.



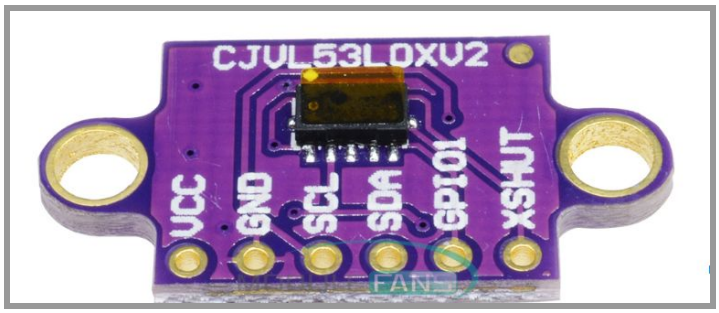
Working principle of ToF Sensors[8]

[8] https://www.onelectrontech.com/stmicroelectronics-vl53l1x-tof-proximity-sensor-detects-distance-up-to-4-meters/

This Time-of-Flight is converted into the distance of the target from the sensor by using the formula,

Distance = (Photon Travel Time/2) x Speed of photon

Since the speed of photons is essentially the speed of light, the distance of up to 2m can be travelled in just 13.33 ns. The photon travel time is not affected by the target reflectance, providing higher accuracy irrespective of the target material.



VL53L0X Sensor

VL53L0X Pin Description

Pin Number	Signal Name	Signal Type	Description
1	VIN	Supply	Supply, to be connected to main supply
2	GND	Ground	To be connected to main ground
3	SCL	Digital input	I2C serial clock input
4	SDA	Digital input/output	I2C serial data
5	GPIO1	Digital output	Interrupt output. Open drain output.
6	XSHUT	Digital input	Xshutdown pin, Active LOW

X-Ranging specifications

Sensor used	VL53LOX
Pins of VL53LOX	➢ VCC ➢ GND ➢ SCL ➢ SDA ➢ GPIO ➢ RESET
SLOTS FOR SENSOR	4 (FRONT, BACK, LEFT, RIGHT) + 1 CONNECTOR
LEDS AVAILABLE	4 (FRONT, BACK, LEFT, RIGHT)
RESET PINS USED IN X-RANGING (OF UNIBUS)	FRONT - Pin 10 BACK - Pin 9 LEFT - Pin 15 RIGHT - Pin 13 DOWN - Pin 8
MODE OF COMMUNICATION	I2C

Approaching the problem

This project will require mounting the X-Ranging board onto Primus X and integrating Ranging Sensor VL53L0X with it. This inclusion of hardware will have to be initialized in the program. After initialization, the ranging value can be obtained from the sensor using an API, which will be stored in a variable declared along with the headers. The ranging values will be printed by printing the values of the variable. To visually confirm the working of the sensor, use LEDs.

Creating a new project

- Create a new PlutoX Project
- Name it as “XRanger”

Starting with the coding.

Headers

For accessing X-Ranging, include the header “XRanging.h” and also include “Utils.h” header for using LEDs and print APIs. Also declare a variable “Data” to store the ranging data obtained from sensors. Set the value initially to zero to avoid garbage values.

```
#include "PlutoPilot.h"
#include "XRanging.h"
#include "Utils.h"

int16_t Range=0;
```

void plutoInit ( )

Initialize the X-Ranging over here by using the API “XRanging.init();”. Use LEFT, RIGHT, FRONT or BACK depending on which part of the X-Ranging board the sensor is mounted on.

```
void plutoInit()
{
/*Add your hardware initialization code here*/

        XRanging.init(LEFT); /*Initialize Left Ranging sensor*/
}
```

void onLoopStart ( )

Deactivate the default LED behavior.

```
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE); /*Disable the default led behavior*/
}
```

void plutoLoop ( )

Store the ranging values in the variable “Data” using the API “XRanging.getRange();” print these values and program blue and red LEDs to turn ON.

```
void plutoLoop()
{
/*Add your repeated code here*/

        Range=XRanging.getRange(LEFT); /*Get Data from Left Ranging sensor*/
        Monitor.println("LRange", Range);
        LED.set(BLUE,ON);
        LED.set(RED, ON);
}
```

void onLoopFinish

Restore the default LED behavior.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);      /*Enable the default led behavior*/
}
```

The entire code

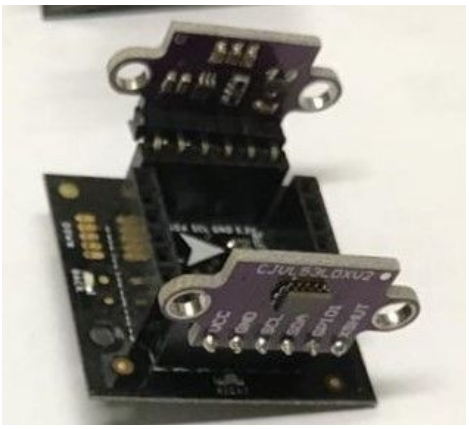
```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include "XRanging.h"
#include "Utils.h"

int16_t Range=0;


/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/
```

```
        XRanging.init(LEFT); /*Initialize Left Ranging sensor*/
}


/*The function is called once before plutoLoop when you activate Developer Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE); /*Disable the default led behavior*/
}


/* The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/

        Range=XRanging.getRange(LEFT); /*Get Data from Left Ranging sensor*/
        Monitor.println("LRange", Range);
        LED.set(BLUE,ON);
        LED.set(RED, ON);
}


/*The function is called once after plutoLoop when you deactivate Developer Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);      /*Enable the default led behavior*/
}
```

Running the project

- Build the project
- Flash the code on PlutoX
- Switch PlutoX OFF to connect the X-Ranging and VL53L0X sensor module with Primus X. follow these steps:
    - Insert the M2 nylon screw from under the board.

- Place the M2 nylon Hexspacer above the nylon screw.
- Place the X-Ranging above the Primus X board on the Unibus such that the hole on X-Ranging aligns with the Hexspacer.
- Tighten the nut on the Hexspacer so that the X-Ranging properly fits onto the Primus X board.
- Place the VL53L0X sensor in correct configuration in any of the slots provided on X-Ranging board. The Vcc pin of VL53L0X should connect with 3.3V in X-Ranging, such that the sensor always points outside.



Mounting VL53L0X on X-Ranging

- After mounting X-Ranging and VL53L0X on Primus X, switch ON PlutoX.

NOTE: In case the hardware is integrated while PlutoX is already ON, the attached hardware will not be initialized. To initialize it, restart PlutoX. The hardware initializes only when PlutoX switches ON.

- Connect PlutoX with the Pluto Controller app.
- Turn the Developer Mode ON.
- Connect Cygnus with PlutoX
- Observe the ranging values on Pluto Monitor

NOTE: Perform this experiment inside closed doors with minimum exposure to sunlight. This is because the ranging sensors transmit and receive photons and sunlight also contains photons which interfere with the actual signal and cause distortions in the received signal. When the sunlight interferes with the signal, an error value (-100) can be observed in Cygnus.

Project Takeaway

In this project, ranging sensor VL53L0X was integrated with PrimusX using X-Ranging board. This sensor gives the ranging data i.e. the distance between the sensor and the target. ToF sensors can be used for a variety of purposes, where knowing the distance of a target from the system is vital. For example, collision avoidance drones use ToF sensors to continuously detect the distance of the drone from its surrounding objects.

This project is the base for the next projects, which utilize this knowledge for developing interesting applications on PlutoX.

Activity

- Once comfortable working with one ranging sensor, attach 2 sensors and get ranges from both of them. Calculate the dimensions of the room by adding the values from both these sensors.

- Try to tinker around and place the sensor at the bottom or top side of the drone and code it to always maintain a specified distance from the floor or ceiling.

PROJECT 9: AIR PONG (PlutoX)

Objectives

- Application of VL53L0X sensor
- Collision avoidance system

Problem statement

Using two VL53L0X sensors create a feedback control system enabling the drone to avoid collision, using which the drone can be used for playing with.

Explanation

Drones have always been mesmerising the users by its flying capabilities. The future consists of many more fun uses of drones than simply flying. Using a variety of sensors, innovative ideas can be implemented on drones for recreational purposes. This project aims at creating one such application, in which the user will be able to play the classic Ping Pong, where PlutoX would be replacing the ball.

This project is basically about building a collision avoidance system for the drone. A system collides with an obstruction when the distance between the system and the obstruction becomes zero. The main idea behind collision avoidance system is to not let the distance be reduced to zero or to be on the safer side, not let the distance be reduced below a threshold value.

The distance between a system (PlutoX) and its surrounding can be read by using VL53L0X sensors as done in the Project 8. Using two of these sensors on either sides of PlutoX will provide a feedback loop, continuously giving a feedback in the form of ranging values. By using this feedback, a collision avoidance system can be developed with respect to those two sides. If two people hold Ping Pong rackets on both sides of PlutoX containing VL53L0X sensors, the rackets could act as the obstruction and the drone will try to avoid it, thus becoming a fun game.

Approaching the problem

This project will use two VL53L0X sensor modules which will be mounted on the left and the right berg strip on X-Ranging. The sensors will constantly detect the distances

of target on their respective sides. When the user will start bringing the racket close to the right or left side of the flying drone, the distance detected by the sensor on that side will start reducing.

The project requires the drone to avoid colliding with the racket. In order to achieve this, the only option with the drone is to move away from the racket if the distance becomes too small. Thus, the logic would be to check the ranging values from the sensors. If the ranging value on the left side becomes less than the threshold value, then the drone should roll right; if the ranging value on the right side becomes less than the threshold value, then the drone should roll left; and if the values on both sides are more than the threshold values, then the drone should roll as per the input given by the user. The ranging value received from VL53L0X has units of mm. The threshold value selected for this project is 500mm (can be changed by the tinkerer). So, if the sensor reading goes below 500mm, then the drone will roll in the opposite direction.

It was mentioned in the earlier project that sunlight contains photons which cause distortions in the sensor readings. Hence, when an error occurs due to sunlight, the sensor reading becomes -100. Now, this value is also less than the threshold value of 500mm. This means that by the logic applied so far, the drone will roll in the opposite direction even due to an error caused by sunlight. To avoid this, the conditional statement should be that **if** the ranging value goes below 500 mm **and** the value is positive i.e. above zero, then the drone should roll in the opposite direction.

The next task is to give roll values. The roll control has a range of 1000 to 2000 with 1500 being the value at rest i.e. when the drone is not inclined on either side. The value for left roll increases from 1500 to 1000 and the value for right roll increases from 1500 to 2000. In this project, when the drone has to roll in the opposite direction, it does not need to roll at a very high rate. Hence, set the value of 1400 to roll left and 1600 to roll right.

In order to execute the loop at a faster rate, change the user loop frequency to 4ms

Creating a new project

- Create a new PlutoX Project
- Name it as “AirPong”

Starting with the coding.

Headers

To access the X-Ranging, include the header “XRanging.h”. Include the header “User.h” to set RC commands for roll and to change the user loop frequency. Use “Utils.h” header to use LEDs and print APIs.

```
#include "PlutoPilot.h"
#include "Xranging.h"
#include "User.h"
#include "Utils.h"
```

void plutoInit ( )

The VM53L0X sensors will be mounted on the sides labelled LFT and RGT i.e. left and the right sides. To initiate them, use API “XRanging.init(LEFT)” and “XRanging.init(RIGHT)”. Also set the user loop frequency to 4 ms by using the API “setUserLoopFrequency()”.

```
void plutoInit()
{
/*Add your hardware initialization code here*/

        XRanging.init(LEFT);     /*Initialize Left Ranging sensor*/
        XRanging.init(RIGHT);    /*Initialize Right Ranging sensor*/

        setUserLoopFrequency(4); /*Change loop frequency for faster loop
execution*/
}
```

void onLoopStart ( )

Deactivate the default LED behavior.

```
void onLoopStart()
{
/*do your one time tasks here*/
```

```
	LED.flightStatus(DEACTIVATE); /*Disable the default led behavior*/
}
```

void plutoLoop ( )

The main user loop will contain **if else** conditions. The first **if** condition will check for obstacle on the left side of the drone. Ranging values will be obtained by the API "XRanging.getRange()". The ranging value should be less than 500mm and more than 0 mm i.e. should be positive. If the condition is satisfied, then set the value of roll at 1600, which will make the drone roll right side. Use the API "RcCommand.set()". Also turn the red LED ON and blue LED OFF.

The second condition, **else if**, will check for the obstacle on the right side. The structure will remain similar to the first condition. Set the RC_ROLL value to 1400 to roll the drone towards the left. Set the red LED OFF and the blue LED ON.

The third condition, **else**, is for when the ranging values on both sides are more than 500mm. In that case, the RC_ROLL value will be set by the user. Since the input given by the user is stored as rcData, set the rcCommand values based on the rcData values by using the API "RcData.get(RC_ROLL)". Also turn both red and blue LEDs OFF.

```
void plutoLoop()
{
/*Add your repeated code here*/

	/*if the sensor detects an obstacle on the left side i.e. range less than 500, roll right*/
	if(XRanging.getRange(LEFT)<500 && XRanging.getRange(LEFT)>0)
	{
		RcCommand.set(RC_ROLL, 1600);
		LED.set(RED,ON);
		LED.set(BLUE,OFF);
	}

	/*if the sensor detects an obstacle on the right side i.e. range less than 500, roll left*/
	else if(XRanging.getRange(RIGHT)<500 && XRanging.getRange(RIGHT)>0)
	{
		RcCommand.set(RC_ROLL, 1400);
		LED.set(RED,OFF);
```

```
                LED.set(BLUE,ON);
        }

        /*when no obstacle is detected, let control be with the user*/
        else
        {
                RcCommand.set(RC_ROLL,RcData.get(RC_ROLL));
                LED.set(RED,OFF);
                LED.set(BLUE,OFF);
        }
}
```

void onLoopFinish

Restore the default LED behavior.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE); /*Enable the default led behavior*/
}
```

The entire code

```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include "Xranging.h"
#include "User.h"
#include "Utils.h"


/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/

        XRanging.init(LEFT);     /*Initialize Left Ranging sensor*/
        XRanging.init(RIGHT);    /*Initialize Right Ranging sensor*/
```

```
	setUserLoopFrequency(4); /*Change loop frequency for faster loop execution*/
}


/*The function is called once before plutoLoop when you activate Developer Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

	LED.flightStatus(DEACTIVATE); /*Disable the default led behavior*/
}


/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/

	/*if the sensor detects an obstacle on the left side i.e. range less than 500, roll right*/
	if(XRanging.getRange(LEFT)<500 && XRanging.getRange(LEFT)>0)
	{
		RcCommand.set(RC_ROLL, 1600);
		LED.set(RED,ON);
		LED.set(BLUE,OFF);
	}

	/*if the sensor detects an obstacle on the right side i.e. range less than 500, roll left*/
	else if(XRanging.getRange(RIGHT)<500 && XRanging.getRange(RIGHT)>0)
	{
		RcCommand.set(RC_ROLL, 1400);
		LED.set(RED,OFF);
		LED.set(BLUE,ON);
	}

	/*when no obstacle is detected, let control be with the user*/
	else
	{
		RcCommand.set(RC_ROLL,RcData.get(RC_ROLL));
```

```
                LED.set(RED,OFF);
                LED.set(BLUE,OFF);
        }
}


/*The function is called once after plutoLoop when you deactivate Developer Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE); /*Enable the default led behavior*/
}
```

Running the project

- Build the project
- Flash the code on PlutoX
- Switch OFF PlutoX and attach X-Ranging board. Remember to tighten the X-Ranging board with the screw provided with the kit.
- Insert the VM53L0X modules correctly on the berg strip labelled LFT and RGT
- Switch ON PlutoX. This will initialize the attached hardware.
- Connect PlutoX with the Pluto Controller app
- Turn the Developer Mode ON
- Conduct the experiment in a place with minimum exposure to sunlight.

Project Takeaway

The project is based on the application of knowledge of VM53L0X sensor module in developing a collision avoidance system. This system can be updated to avoid collision on all four sides as well. Such systems can be used for different applications, for example, to make drones safer for flying in public places, can be used indoors in industries or warehouses without the fear of any loss due to collision and it can also be used innovatively for recreational purposes.

Among the total drone user base, a majority use drones for recreational purposes. But these fun applications are more or less limited to simply flying. With innovative ideas, drones can be included in regular games or sports and possibly develop new games

around drones, like the drone racing. This project aims at creating a different way of thinking for the application of ideas.

Activity

Stair climbing drone

Place ranging sensors at the bottom of the drone and program it to fly at a constant height. When the drone flies over stairs it should detect the increase or decrease in height and try to maintain the constant height, making it look like climbing the stairs.

PROJECT 10: WALLS ARE LAVA (PlutoX)

Objectives

- Basics of PD control system
- Implementation of PD control system

Problem statement

Program the drone to fly autonomously staying at the centre and never touching any wall on the sides.

Explanation

This project is based on the popular game Floor is Lava. Imagine that while flying, the walls are lava, and the objective of the pilot is to not touch the walls. This project is, however, not based on the flying skills of the pilot but rather based on the programming skills of the tinkerer.

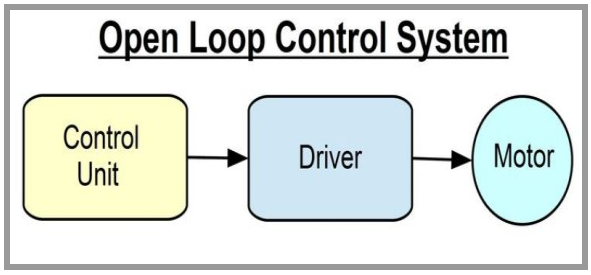
There are various ways by which the required objective can be achieved. One of the ways is an extension of the Project 9, Air Pong. By using ranging sensor, the obstacles on either side of the drone can be avoided.

The basic idea of walls are lava is to maintain the position of the drone at the center between walls. To achieve this, a control system has to be implemented. A control system manages, commands, directs, or regulates the behavior of other devices or systems using physical components and control functions.

Classes of control system

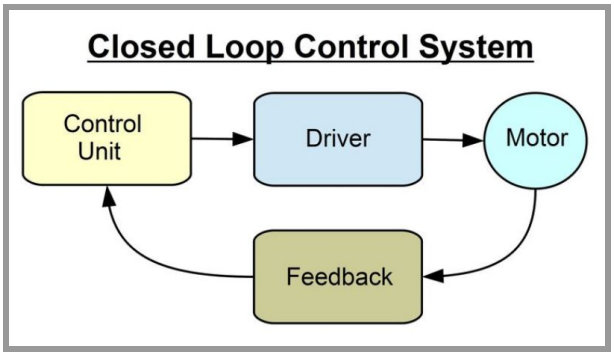
- Open Loop Control System

A control system in which the control action (input) is totally independent of output of the system is called open loop control system. Basically, it does not contain a feedback system.



- Closed Loop Control System

Control system in which the output has an effect on the input quantity in such a manner that the input quantity will adjust itself based on the output generated is called closed loop control system. It contains a feedback system.



Types of Control Systems

- On-off Control System

This type of control system was used in Project 9, Air Pong. On-off control system has only two states, either fully ON or fully OFF. The on-off control system ensured that the drone received zero roll input if the condition was not satisfied and received the set roll value when the condition was satisfied. There was no in between roll value.

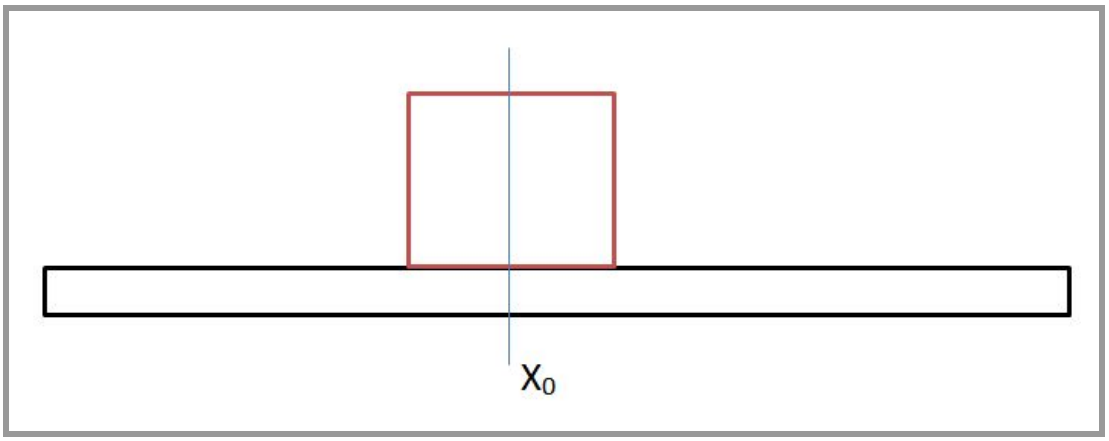
With the use of on-off control system, only two control states are possible which can be used for limited control applications, where these two control states are enough for achieving the control objective. However, the oscillating nature of this control system limits its usage and hence is replaced by PD control systems.

- PD Control System

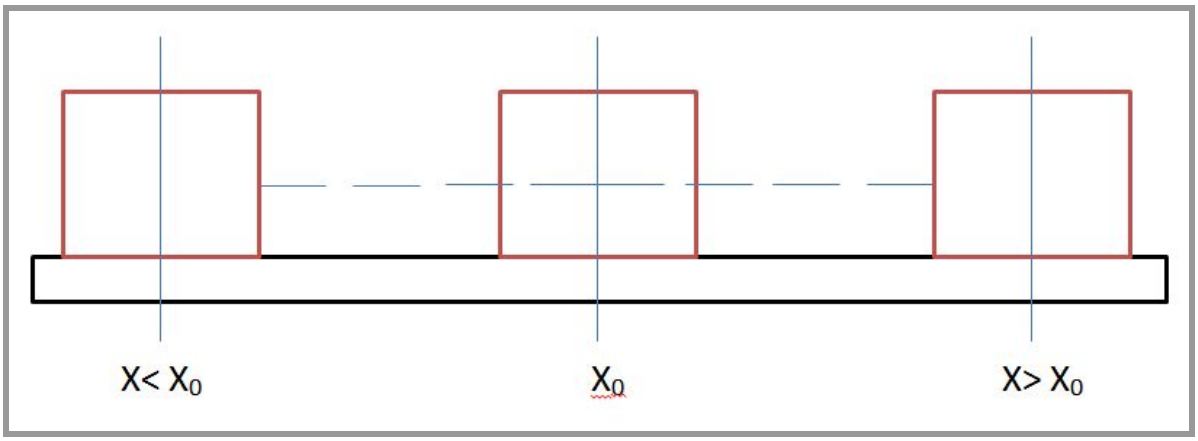
The Proportional Derivative Control System maintains the output such that there is zero error between the process variable and desired output. This control system can be understood by means of a physical system as explained ahead.

P Controller

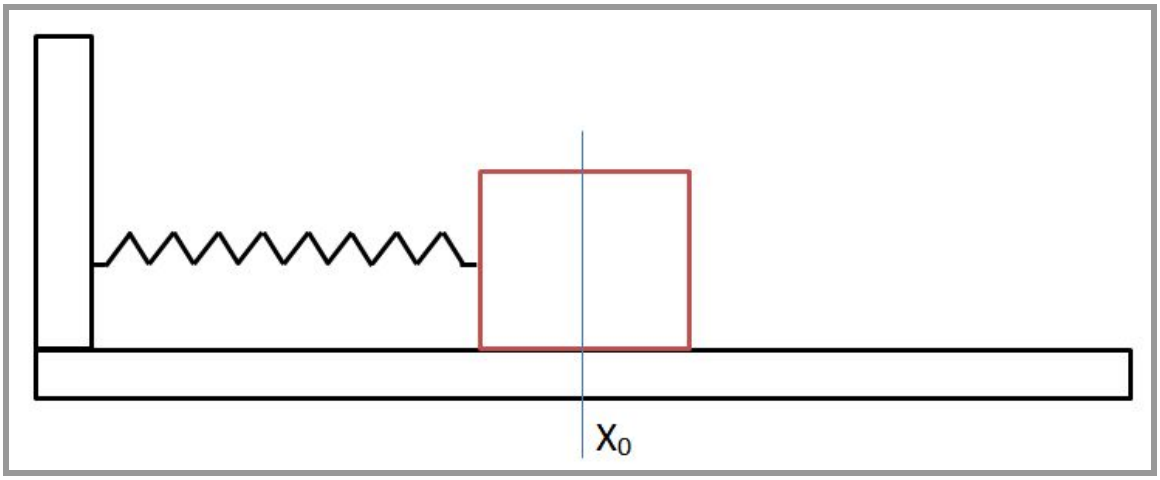
Imagine a block of mass 'M' placed on an ideal frictionless surface. The position of the block is X_0 .



Under an external force, the block will move to a new position X such that $X > X_0$ or $X < X_0$. Design a system such that the block returns to its original position, X_0 , called as the set point.



This can be done by adding a spring such that at spring equilibrium, the mass will also remain at its equilibrium position X_0 . Such a system is called as spring-mass system.



When the external force applied on the mass is removed, the force of the spring will act on the mass to bring it back to its equilibrium position. This force generated by the spring is given by,

$$F = kx$$

Where,

k is the spring constant

x is the distance by which the spring is stretched from its equilibrium position.

In this case, $x = X - X_0$, which is called as the error.

$$\therefore\ F = k\,(X - X_0)$$

This shows that the force generated is directly proportional to the difference between the process variable (X) and the set point (X_0). So, the larger the magnitude of error, larger will be the spring force.

Thus, Proportional Control can be defined as a type of linear feedback control system in which a correction is applied to the controlled variable which is proportional to the difference between the process variable and the setpoint.

The equation of proportional control can be written as,

$$y(t) = kp * e(t)$$

Where,

$y(t)$ is the rate of response

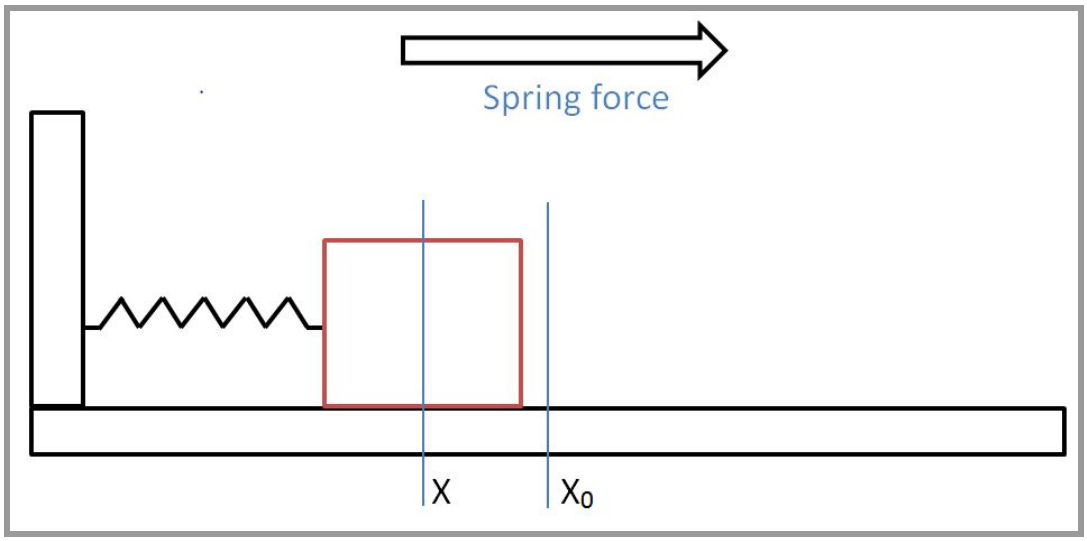
kp is the proportional gain factor

$e(t)$ is the error i.e. the difference between the process variable and setpoint.

This means that the rate of response, $y(t)$, is directly proportional to the proportional gain factor, kp. So, the speed of response can be increased by increasing the value of kp. But an increase in the value of kp will also increase the overshoot, which would degrade the stability of the system.

Overshoot

On the removal of the external force on the mass, the spring will generate a force to bring the mass back to the equilibrium position.



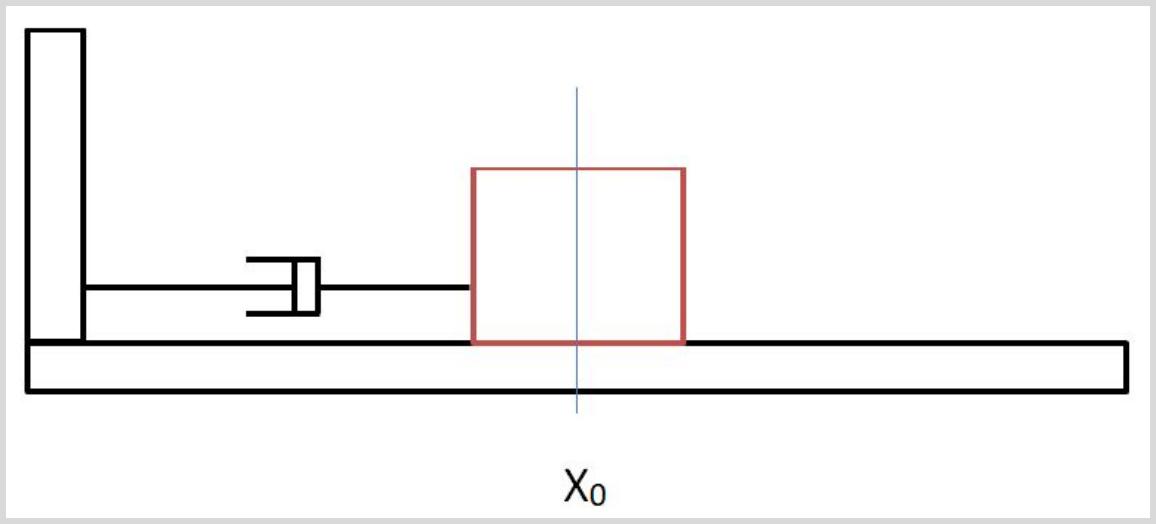
While moving from X to X_0, the force of the spring will decrease as it reaches its equilibrium position. However, as the mass moves from X to X_0, its velocity increases and hence it gains momentum. At the equilibrium position X_0, the spring force is the least but the momentum of the mass is maximum. As a result, due to this increased velocity, the mass overshoots its equilibrium position and moves beyond the equilibrium position.

On the other side of X_0, the velocity eventually becomes zero and the mass stops at a point on this side which becomes its new position. The same process repeats again while it returns from this new position to its equilibrium by spring force. This causes the mass to oscillate about its equilibrium position. Since the system is assumed to be ideal, the mass will continue to oscillate. In case of a real system, the mass will oscillate until it stops at the equilibrium position as the velocity will eventually become zero because of friction between the mass and the surface.

In an ideal system, to bring the velocity to zero, derivative control is used.

D Controller

In case of a physical system, D controller is added by adding a damper. This system is called as mass-damper system.



Damper will reduce the velocity of the mass thereby reducing the oscillations, making it a much more stable system. The force exerted by the damper is given by,

$$F = b\dot{x}$$

Where,

b is the damping coefficient

$\dot{x}$ is dx/dt (rate of change of error with respect to time).

Thus, derivative control does not depend on the magnitude of the error but depends on the rate of change of error. The faster the rate of change of error, more will be the damping force.

The equation for the derivative control can be written as,

$$y(t) = kd * (de(t)/dt)$$

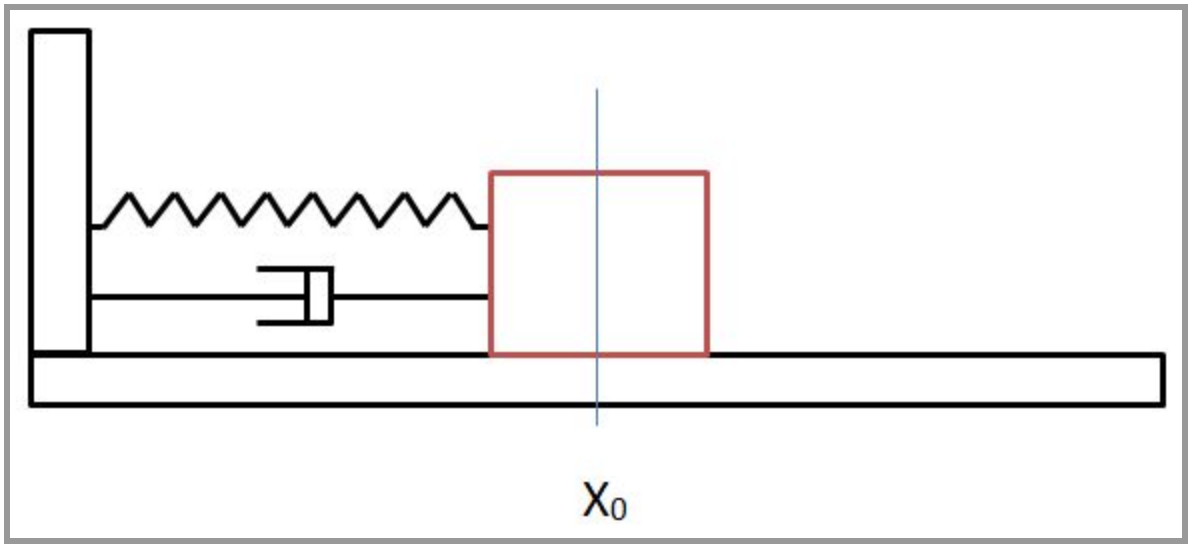
Where,

$y(t)$ is the rate of response

$e(t) = X - X_0$

kd is the derivative gain factor.

The entire system together is called as spring-mass-damper system.



The overall force acting on the mass is given by,

$$F = kx - b\dot{x}$$

i.e., $$y(t) = kp * e(t) - kd * de(t)/dt$$

It can be seen that the damper is used to reduce the velocity. By using different values of kd, there can be three types of systems.

- Under Damped System
- Over Damped System
- Critically Damped System

Under Damped System

This system is a result of using low values of kd. In this system, the velocity of the mass at X_0 is greater than zero. This causes the mass to overshoot the equilibrium position, resulting in oscillations. Thus, the mass reaches the setpoint faster but overshoots, lowering the stability of the system.

Over Damped System

This system is a result of using higher values of kp. In this system, the velocity of the mass becomes zero even before it reaches the equilibrium position. Thus, the system either takes a long time to reach the setpoint or does not reach the setpoint at all.

Critically Damped System

This system is a result of using an accurate value of kp. In this system, the mass reaches the equilibrium position perfectly without any overshoot. It reaches faster than the over damped system.

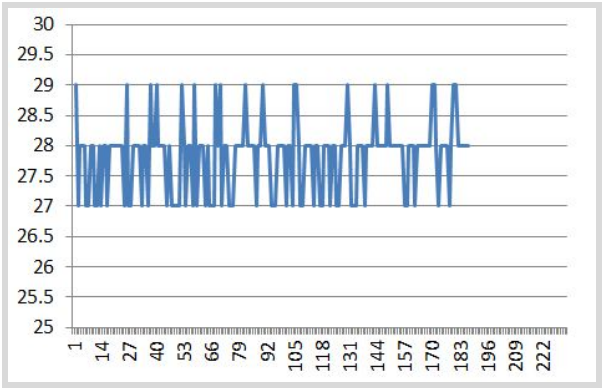
In a nutshell, PD control system can be used to bring a system to a desired setpoint. The difference between the process variable and the setpoint is called as error. Proportional control tries to bring the system to the setpoint by applying a correction proportional to the error. In doing this, the system oscillates. To damp the oscillations, derivative control is used which acts on the rate of change of error.

Low Pass Filter

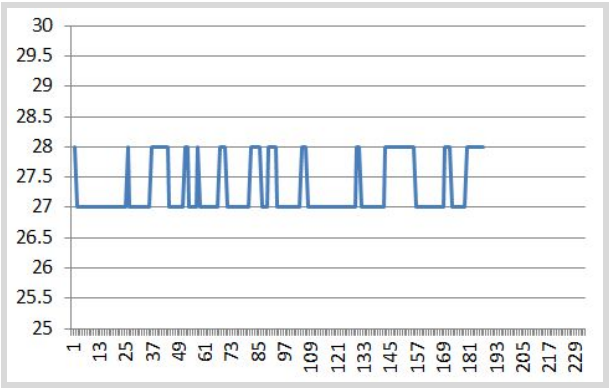
While working with sensors, it is important to understand that the sensor data contains a lot of noise. If this data is directly used in PD control, the error gets amplified causing stability issues. To avoid this problem, a Low Pass Filter is used.

Low Pass Filter is a filter that passes signals with the frequency lower than the cut-off frequency and removes the signals with higher frequency. In this project, low pass filter is applied to the sensor data and the error data so that the noise is filtered out and only the required signal passes to the next system.

The following graphs show the unfiltered and filtered data when the PlutoX is kept at the same spot. The filtered data contains much less noise, giving better results.



Unfiltered



Filtered

Use the function: $f(t) = \alpha e(t) + (1 - \alpha)e(t-1)$

Where α is a constant whose value is taken as 0.8 for this project.

Approaching the problem

The basic logic in this project is to find the error by comparing the distances on the left and the right side of the drone. PD controller will be used to get better control over the error correction. The output of the PD controller will be used for correction of current roll value of the drone.

For calculating the error, the first requirement is to get the ranging values from the ranging sensors. Suppose first the ranging value is obtained from the left ranging sensor.

This data will contain noise and hence it has to be passed from a low pass filter. There might also be an error in the data due to sunlight. In such a case, the previous filtered ranging data is used instead of the error data. Therefore, the data should be passed through the LPF only if it is greater than zero. If the data is negative, it indicates disturbance from sunlight and so in such a case, the previous filtered ranging data can be used in place of this negative value. The same steps are followed for right ranging sensor.

The filtered ranging data from both sensors is stored in two respective variables. Next, error is calculated by finding the difference between both variables. This error is also passed through LPF to get less noisy filtered value of error.

Coming to the PD controller, the equation for getting the value of PD is,

$$y(t) = kp * e(t) - kd * de(t)/dt$$

$y(t)$ is the output of the controller. In order to obtain the output, the values of the terms on the RHS have to be fed into the equation. The values of kp and kd can be set by the user in the app itself (Explained in 'Running the Project' section). These values are then accessed by using API. $e(t)$ is the filtered error that was calculated earlier.

$de(t)/dt$ is the rate of change of error. The numerator is obtained by calculating the difference between the current filtered error and the previous filtered error. Set the value of dt equal to 0.1.

After feeding these values in the equation for PD controller, the output PD will be received. This output value is the correction value for the error. As discussed in

Project 9, Air Pong, the roll value ranges from 1000 to 2000 with 1500 as the value at rest. The correction value of PD is added to 1500 i.e. the value at rest. If the correction value is negative, the roll value will go from 1500 towards 1000, i.e. the drone will roll left. If the correction value is positive, the roll value will go from 1500 towards 2000, i.e. the drone will roll right.

However, a problem may arise if the correction value goes outside the range of -500 to 500 since it will in turn make the roll value go outside the range of 1000 to 2000. In order to avoid this, define a function “constrain” to limit the values of PD between -500 to 500. This function will return values based on the value of PD. If the value of PD is above 500, it will return the maximum possible value which is 500. If the value of PD is less than -500, it will return the minimum possible value which is -500. If the value of PD is within the allowed range, it will return the value of PD.

After adding the value of PD to 1500, set it as the rcCommand value for roll. In the end, update the previous values of left ranging, right ranging and the error by the respective new values obtained in the current execution of the loop.

Creating a new project

- Create a new PlutoX Project
- Name it as “WallsAreLava”

Starting with the coding.

Headers

Include header “XRanging.h” to get access to the X-Ranging. The PID can be accessed by including the header “Control.h”. The value of roll can be set by including the header “User.h”. Also include the header “Utils.h” for debugging purposes.

Define the value of dt equal to 0.1; Declare the required variables under float. Declare the variable “RollValue” under int16_t as the value of roll has to be an integer. Declare “UserPID” under PID.

Also, define the function “constrain” under int. The function is coded at the end, with the logic as discussed earlier.

```
#include "PlutoPilot.h"
```

```
#include "Control.h"
#include "XRanging.h"
#include "Utils.h"
#include "User.h"

#define dt 0.1

float XLeft=0;                          /*left ranging value*/
float XRight=0;                         /*right ranging value*/
float left;                             /*filtered left ranging value*/
float right;                    /*filtered right ranging value*/
float LeftOld=0;                /*previous filtered value of left ranging*/
float RightOld=0;               /*previous filtered value of right ranging*/
float Error;                    /*unfiltered error*/
float FilError;                 /*filtered error*/
float PreError=0;               /*previous filtered error*/
float Derivative;               /*difference between current and previous error*/
int16_t RollValue=0; /*roll value for rcCommand*/
float KP;                               /*proportional gain value*/
float KD;                               /*proportional derivative value*/
int PD;                                 /*output of PD controller*/


PID UserPID;

int constrain(int amt, int low, int high); /*declaring the function*/
```

```
int constrain(int amt, int low, int high)
{
        if (amt < low) /*if PD value is less than -500*/
                return low;
        else if (amt > high)  /*if PD value is more than 500*/
                return high;
        else
                return amt;
}
```

void plutoInit ( )

Initialize the left and right ranging sensors.

```
void plutoInit()
{
/*Add your hardware initialization code here*/

        /*Initialize Ranging sensor*/
        XRanging.init(LEFT);
        XRanging.init(RIGHT);
}
```

void onLoopStart ( )

Deactivate the default LED behavior.

```
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable the default led behavior*/
}
```

void plutoLoop ( )

Get the ranging value from the left ranging sensor using the API "XRanging.getRange(LEFT)" and store this ranging value in "XLeft". Note that the value is divided by 10 to convert its unit to cm from mm. The low pass filter is applied **if** the value of XLeft is greater than zero. In the LPF, the filtered value is stored in "left", "XLeft" is the current ranging value and "LeftOld" is the previous filtered ranging value. **Else**, use the previous filtered ranging value "LeftOld" as the new filtered ranging value "left". Note that "LeftOld" was set to zero while declaring otherwise it could contain an initial garbage value, which would have been used as it is during the execution of the loop for the first time, bringing errors.

Print the value of "left". Follow the same structure for right ranging sensor.

Next task is to access the KP and KD values from the user. Using the API "PIDProfile.get(PID_USER, &UserPID)" will get the PID values set by the user and save the values in "UserPID". The values of P and D can then be stored into "KP" and "KD" by using the APIs "UserPID.p" and "UserPID.d" respectively. Divide each of them by 20 to appropriately scale down the value so that they can be used in the program.

After this, write the code for the PD controller. Get the error by calculating the difference between “right” and “left” and store it in “Error”. Filter this error. Use “Error” as the current value, “PreError” as the previous filtered value and store the new filtered error value in “FilError”. Calculate the difference between “FilError” and “PreError” and store it in “Derivative”, which will be used in derivative part.

Since all the terms required for calculating PD are now known, code the main PD equation. The PD value received could be out of the range of -500 to 500. Use the constrain function here to restrict the value of PD within the required range.

Set the value of roll i.e. “RollValue” as the addition of PD and 1500. This roll value is set as RcCommand using the API “RcCommand.set()” and print the RcCommand.

In the end, update the previous filtered values “LeftOld”, “RightOld” and “PreError” by their respective new filtered values “left”, “right” and “FilError”.

```
void plutoLoop()
{
/*Add your repeated code here*/

        XLeft=(XRanging.getRange(LEFT))/10;   /*Get sensor data in cm*/
        if(XLeft>0)     /*If not affected by sunlight*/
        {
                left= (0.8*XLeft)+(1-0.8)*LeftOld;    /*LPF*/
        }
        else
        {
                left=LeftOld;
        }
        Monitor.print("Left distance", left);

        XRight=(XRanging.getRange(RIGHT))/10;
        if(XRight>0)   /*If not affected by sunlight*/
        {
                right=(0.8*XRight)+(1-0.8)*RightOld;        /*LPF*/
        }
        else
        {
                right=RightOld;
```

```
        }
        Monitor.println("Right distance", right);

        PIDProfile.get(PID_USER, &UserPID);      /*Get the PID data from App*/

        KP=UserPID.p/20;    /*Get P value*/
        KD=UserPID.d/20;    /*Get D value*/

        /*PID controller*/
        Error = right - left;
        FilError=(0.8*Error)+(1-.8)*PreError;
        Derivative = (FilError-PreError);

        PD=(KP*FilError) + (KD*(Derivative/dt));
        PD=constrain(PD, -500, 500);       /*Constrain to match MAX RC Command value*/

        RollValue=PD+1500;
        RcCommand.set(RC_ROLL, RollValue);
        Monitor.println("RC", RcCommand.get(RC_ROLL));

        /*Update the values*/
        LeftOld=left;
        RightOld=right;
        PreError=FilError;
}
```

void onLoopFinish

Activate the default LED behavior.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE); /*Enable able the default led behavior*/
}
```

The entire code

```
/*Do not remove the include below*/
```

```
#include "PlutoPilot.h"
#include "Control.h"
#include "XRanging.h"
#include "Utils.h"
#include "User.h"

#define dt 0.1

float XLeft=0;                  /*left ranging value*/
float XRight=0;                 /*right ranging value*/
float left;                     /*filtered left ranging value*/
float right;                /*filtered right ranging value*/
float LeftOld=0;            /*previous filtered value of left ranging*/
float RightOld=0;           /*previous filtered value of right ranging*/
float Error;                /*unfiltered error*/
float FilError;             /*filtered error*/
float PreError=0;           /*previous filtered error*/
float Derivative;           /*difference between current and previous error*/
int16_t RollValue=0; /*roll value for rcCommand*/
float KP;                       /*proportional gain value*/
float KD;                       /*proportional derivative value*/
int PD;                         /*output of PD controller*/

PID UserPID;

int constrain(int amt, int low, int high); /*declaring the function*/

/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/

        /*Initialize Ranging sensor*/
        XRanging.init(LEFT);
        XRanging.init(RIGHT);
}


/*The function is called once before plutoLoop when you activate Developer
Mode*/
void onLoopStart()
{
```

```
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable the default led behavior*/
}


/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/

        XLeft=(XRanging.getRange(LEFT))/10;    /*Get sensor data in cm*/
        if(XLeft>0)     /*If not affected by sunlight*/
        {
                left= (0.8*XLeft)+(1-0.8)*LeftOld;    /*LPF*/
        }
        else
        {
                left=LeftOld;
        }
        Monitor.print("Left distance", left);

        XRight=(XRanging.getRange(RIGHT))/10;
        if(XRight>0)   /*If not affected by sunlight*/
        {
                right=(0.8*XRight)+(1-0.8)*RightOld;        /*LPF*/
        }
        else
        {
                right=RightOld;
        }
        Monitor.println("Right distance", right);

        PIDProfile.get(PID_USER, &UserPID);      /*Get the PID data from App*/

        KP=UserPID.p/20;    /*Get P value*/
        KD=UserPID.d/20;    /*Get D value*/

        /*PID controller*/
        Error = right - left;
        FilError=(0.8*Error)+(1-.8)*PreError;
        Derivative = (FilError-PreError);
```

```
        PD=(KP*FilError) + (KD*(Derivative/dt));
        PD=constrain(PD, -500, 500);      /*Constrain to match MAX RC Command value*/

        RollValue=PD+1500;
        RcCommand.set(RC_ROLL, RollValue);
        Monitor.println("RC", RcCommand.get(RC_ROLL));

        /*Update the values*/
        LeftOld=left;
        RightOld=right;
        PreError=FilError;
}


/*The function is called once after plutoLoop when you deactivate Developer Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE); /*Enable able the default led behavior*/
}


int constrain(int amt, int low, int high)
{
        if (amt < low) /*if PD value is less than -500*/
                return low;
        else if (amt > high)  /*if PD value is more than 500*/
                return high;
        else
                return amt;
}
```

Running the project

- Build the project
- Flash the project on PlutoX
- Turn PlutoX OFF and integrate X-Ranging along with two VM53L0X sensors on the left and right side. Follow the instructions as in earlier projects.

- Turn PlutoX ON to initialize the new hardware.
- Conduct the experiment in a closely walled place such as a corridor, to observe the changes conveniently. Place the drone in such a way that the sides face the walls.
- Connect PlutoX with Pluto Controller app
- Turn the Developer Mode ON
- Tune P and D values. Go to Menu > Drone Settings > Set PID values > User. Follow these steps:
    - The initial value of P and D both is set to zero. Hence, KP and KD are also zero and because of this, the value of PD will also be zero.
    - Starting from zero, increase the value of only P by a small number and fly the drone.
    - With an increase in the value of P, it can be observed that the drone oscillates with higher frequency. Bring an obstacle in the path of the ranging sensor to make it roll on the opposite side. This will induce some oscillations.
    - When the drone oscillates at a very high frequency, set the value of P just below that value.
    - Now, increase the D value by small number.
    - An increase in D will result in reduced oscillations.
    - Adjust the D value such that the drone reaches its set point without any oscillations. This can be observed by bringing an object in the path of the ranging sensor.
    - The ideal values of P and D will result in zero oscillations.

Project Takeaway

This project gets tinkering to the next level. Along with the theory concepts of PD Controller, it gives a practical experience of it. Control systems are used in almost every industry, including drone industry. By now, tinkerer must be comfortable with accessing the values from ranging sensors so this project aims at making the tinkerer aware about control systems.

Low Pass Filter was also used in this experiment. It is used when the data is noisy, which results in errors if fed directly into the system. It is worth noting that in the previous experiment, LPF was not used as the noise did not affect the system considerably. In the current project, using PD controls meant an amplification of signals which made the use of LPF necessary.

In order to achieve stability in any application, PD control system can be used.

Activity

Let the drone fly in between the walls without touching them just like the above problem statement, the only difficulty is that the walls are discontinuous, i.e. a part of the wall on any one side is missing. Now program the drone to stop it from drifting away.

PROJECT 11: HYBRID DRONE (PlutoX)

Objectives

- Working and control of DC motors
- Reversible drivers
- Movement/Control of rover

Problem statement

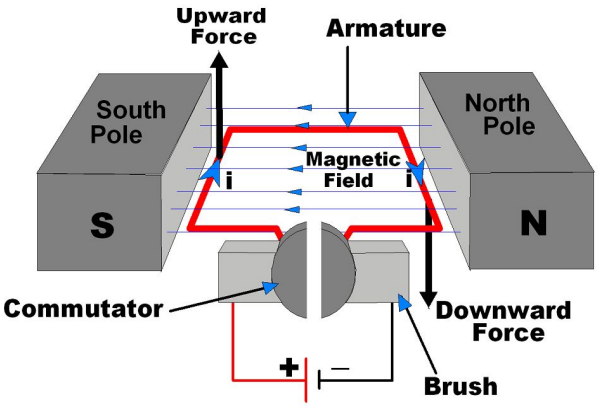
Using the hybrid addon wheels, transform the drone into a hybrid drone, which can run on the ground as a rover when needed.

Explanation

This project uses the addon hybrid wheels. These wheels are attached to DC motors. These motors will be connected to the motor drivers M1 and M4. These are the reversible motor drivers using H bridge. It is important to understand the basic working of DC motors and H bridge drivers, which are explained ahead.

DC Motors

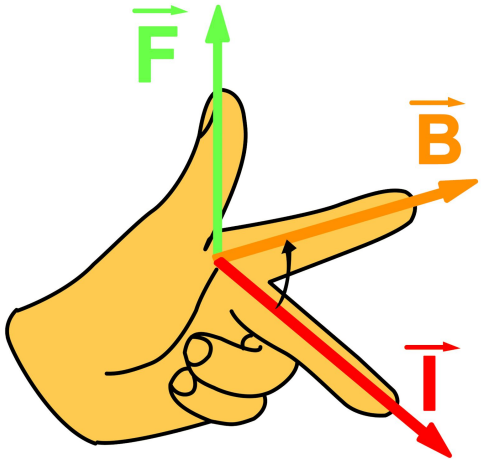
A motor is an electrical machine which converts electrical energy into mechanical energy. The principle of working of a DC motor is that "When a current carrying conductor is placed in a magnetic field, it experiences a mechanical force".



DC Motor Conceptual Diagram[9]

[9] https://www.electronicwings.com/sensors-modules/dc-motor

In the above diagram, a current carrying coil is placed in the magnetic field. This coil hence experiences a force. The direction of this force is given by Fleming's Left Hand Rule.



Fleming's Left Hand Rule[10]

Fleming's Left Hand Rule states that if the thumb, index finger and middle finger are held mutually perpendicular to each other, and if the index finger represents the direction of the magnetic field (B), the middle finger represents the direction of the current (I), then the direction of the thumb represents the direction of the force (F) that acts on the current carrying conductor.

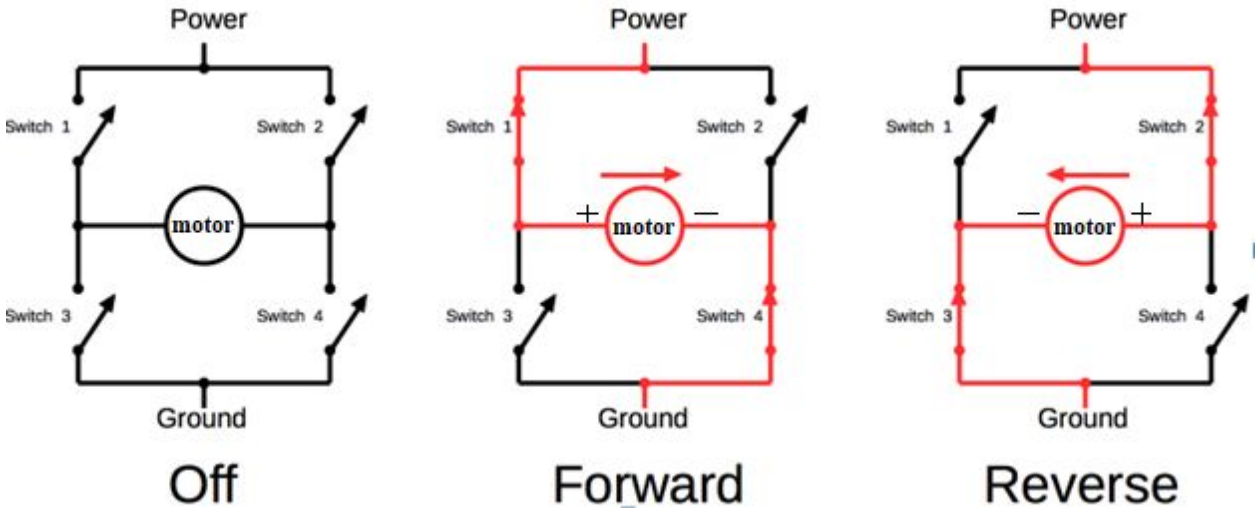
As a result of this force experienced by the coil, it starts rotating in the direction shown in the diagram. Detailed working of DC Motors can be understood by the following video:

DC Motor, How it works?

H Bridge

The motor drivers M1 - M4 are reversible motor drivers that use H Bridge. H Bridge is an electronic circuit that changes the polarity of the voltage given to the motor, thus changing its direction of rotation.

[10] http://electricalacademia.com/wp-content/uploads/2017/01/left-hand-rule.gif



H-bridge is a simple circuit, containing four switching elements, with the load at the center, in an H-like configuration. The top end of the bridge connected to power and bottom end connected to ground. The polarity of voltage applied to the motor can be controlled by controlling the switches. The switching are usually BJT or FET.

- In OFF condition, all the switches are opened. No current flows through the brushed motor.

- In FORWARD condition, switches 1 and 4 are closed. This sets the polarity as shown in the diagram to move the motor in forward direction.

- In REVERSE condition, switches 2 and 3 are closed. This changes the polarity as shown in the diagram which moves the motor in reverse direction.

Movement/Control of Rover

It is crucial to understand how the rover movements can be controlled. The rover will be controlled using the joystick in the app itself. The throttle stick and roll stick will be used for the control.

The throttle stick basically controls the forward and reverse movement of the rover. Roll stick will move the rover in clockwise or counterclockwise direction. The other movements of the drone are a combination of the movement of both the sticks.

For example, turning left while moving forward can be done by moving the rover in forward direction and rotating it counterclockwise. All the possible movements of the rover are mentioned below:

Movements	Throttle	Roll
Forward	Up	-
Reverse	Down	-
Clockwise	-	Right
Counterclockwise	-	Left
Forward + Right turn	Up	Right
Forward + Left turn	Up	Left
Reverse + Right turn	Down	Left
Reverse + Left turn	Down	Right

Approaching the problem

The rover should function only when the drone is on the ground. To find out whether the drone is on the ground or not, check whether the drone is armed or not. If it is not armed, only then should it work as a rover.

The two hybrid wheels will be connected to the reversible motor drivers M1 and M4. The input to these drivers will be given by throttle and roll controls. These inputs will decide the power to be given to the motors and the motor direction as well. Thus, the programming for the project can be divided into two parts:

- Getting the RcData of throttle and roll from the user.
- Calculating the input value for the motors M1 and M4.

The RcData values of throttle and roll can be accessed by using their APIs and stored in particular variables as done in earlier projects. These values together would decide the input values for the motors M1 and M4. The formula for calculation of input value for motors M1 and M4 can be obtained as follows.

The range of RcData for throttle is 1000 - 2000 with 1500 being the value at the rest. When the throttle is increased (stick moving upwards), the value of throttle ranges from 1500 to 2000. When the throttle is decreased (stick moving downwards), the

value of throttle ranges from 1500 to 1000. Throttle stick will control the forward and reverse movement of the rover. Thus, when the RcData value of throttle is between 1500 and 2000, the rover should move forward and when the value is between 1500 and 1000, the rover should move in the reverse direction.

The range of RcData for roll is 1200 - 1800 with 1500 being the value at the rest. When the roll stick moves right, the value of roll ranges from 1500 to 1800. When the roll stick moves left, the value of roll ranges from 1500 to 1200. Roll stick will control clockwise and counterclockwise movement of the rover. Thus, when the RcData value of roll is between 1500 and 1800, the rover should rotate clockwise and when the value is between 1500 and 1200, the rover should rotate counterclockwise.

Subtracting the central value of 1500 from RcData of both throttle and roll will give their respective deviation from the centre point. For throttle stick, upward movement will give positive values and downward movement will give negative values. For roll stick, right movement will give positive values and left movement will give negative values.

The following table represents the relation between the subtracted RcData of throttle and roll with the direction of motors M1 and M4 for primary motions of the rover. For motors, positive represents forward direction and negative represents reverse direction.

Motion of the rover	**Direction of Motor**		**Subtracted RcData Values**	
	M1	**M4**	**Throttle**	**Roll**
Forward	Positive	Positive	Positive	-
Reverse	Negative	Negative	Negative	-
Clockwise	Positive	Negative	-	Positive
Counterclockwise	Negative	Positive	-	Negative

For motor M1: It can be seen that the values of throttle and roll are in agreement with the direction of rotation of motor M1. Hence, these values can be directly added to get the input value of motor M1.

Input value for M1 = Subtracted RcData of throttle + Subtracted RcData of roll

For motor M4: The value of throttle is also in agreement with the direction of rotation of motor M4. However, the value of roll is opposite to that of the direction of rotation of motor M4. Hence, the opposite (i.e. negative) value of RcData of roll should be added to the value of RcData of throttle.

Input value for M4 = Subtracted RcData of throttle - Subtracted RcData of roll

After calculation, positive values of M1 and M4 would mean the direction should be forward and the negative values would mean reverse direction.

The calculated values are constrained between -500 and +500 by using the constrain function as used in Project 10, Walls Are Lava. These calculated values can not be directly used as input for motors M1 and M4 since the input has to be in the range of 1000 - 2000. Hence, the calculated values will be first passed through an absolute function to remove the negativity if any, and then multiplied by 2 and added to 1000, giving a higher value. These values are then set as input for motors M1 and M4.

Creating a new project

- Create a new PlutoX project
- Name it as “HybridDrone”

Starting with the coding

Headers

To get access to the RcData of throttle and roll, include the header “User.h”. Reversible motors M1 and M4 can be used by including the header “Motor.h”. The access to set input value for motors will be obtained by including the header “Control.h”. Include “Utils.h” header for debugging.

Declare the variables “roll_value” and “throttle_value” to store RcData of roll and throttle respectively. Declare variables “m1Value” and “m4Value” which will contain the values to be given as input to motors M1 and M4 respectively.

Also declare the “constrain” function and “generatePWM” function. These functions are defined at the end of the entire program based on the logic described earlier.

```
#include "PlutoPilot.h"
#include  "Control.h"
#include  "User.h"
#include  "Motor.h"
#include  "Utils.h"

#define ABS(x) ((x) > 0 ? (x) : -(x))

int16_t roll_value;
int16_t throttle_value;
int16_t m1Value;
int16_t m4Value;

int constrain(int amt, int low, int high);
int generatePWM(int amt);
```

```
int constrain(int amt, int low, int high)
{
	if (amt>high)
	{
		return high;
	}
	else if (amt<low)
	{
		return low;
	}
	else
	{
		return amt;
	}
}
```

```
int generatePWM(int amt)
{
	amt= ABS(amt);

	amt=1000+(amt*2);

	return amt;
}
```

void plutoInit ( )

Initiate motors M1 and M4 separately instead of initiating all the reversible motors together. Use API “Motor.init( )” to initiate the motors.

```
void plutoInit()
{
/*Add your hardware initialization code here*/

        Motor.init(M1);       /*Initialize motor M1*/
        Motor.init(M4);       /*Initialize Motor M4*/
}
```

void onLoopStart ( )

Set the values of “roll_value” and “throttle_value” to zero. Deactivate the default LED behavior.

```
void onLoopStart()
{
/*do your one time tasks here*/

        roll_value=0;
        throttle_value=0;
        LED.flightStatus(DEACTIVATE); /*Disable the default led behavior*/
}
```

void plutoLoop ( )

Use the API “FlightStatus.check(FS_ARMED)” to check whether the drone is armed or not. **If** it is armed, set the power to the motors M1 and M4 to the minimum values and activate the default LED behavior. **Else**, use the API “RcData.get( )” to get the values of “RC_THROTTLE” and “RC_ROLL” and store them in the variables “throttle_value” and “roll_value” respectively.

Now, calculate the values of “m1Value” and “m4Value” which will be a combination of both “throttle_value” and “roll_value” as discussed earlier. Also, use the constrain function to restrict the value of both “m1Value” and “m4Value” in the range of -500 to +500.

Using the API "Motor.setDirection( )", set the direction of motor M1 based on the condition that **if** the value of "m1Value" is positive then the direction should be forward **else**, set it in reverse direction. After setting the direction, set the input value for motor M1 using the API "Motor.set( )" and the function "generatePWM". Repeat the same set of instructions for motor M4.

```
void plutoLoop()
{
/*Add your repeated code here*/

	if (FlightStatus.check(FS_ARMED))	/*if drone is armed motors M1,M4 should be OFF*/
	{
		LED.flightStatus(ACTIVATE);	/*Enable the default led behavior*/
		Motor.set(M1, 1000);
		Motor.set(M4, 1000);
	}
	else
	{
		/*get data from user*/
		throttle_value= RcData.get(RC_THROTTLE);
		roll_value= RcData.get(RC_ROLL);
	}

	/*Condition for movement of wheels*/
	m1Value= (throttle_value-1500)+(roll_value-1500)/2;
	m1Value= constrain(m1Value, -500, 500);

	m4Value= (throttle_value-1500)-(roll_value-1500)/2;
	m4Value= constrain(m4Value, -500, 500);

	if (m1Value>0)
	{
		Motor.setDirection(M1, FORWARD);
	}
	else
	{
		Motor.setDirection(M1, BACKWARD);
	}
	Motor.set(M1, generatePWM(m1Value));
```

```
        if (m4Value>0)
        {
                Motor.setDirection(M4, FORWARD);
        }
        else
        {
                Motor.setDirection(M4, BACKWARD);
        }
        Motor.set(M4, generatePWM(m4Value));
}
```

void onLoopFinish ( )

Restore the default LED behavior. Set the values of motors M1 and M4 to the minimum value.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE); /*Enable the default led behavior*/
        Motor.set(M1, 1000);
        Motor.set(M4, 1000);
}
```

The entire code

```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include  "Control.h"
#include  "User.h"
#include  "Motor.h"
#include  "Utils.h"

#define ABS(x) ((x) > 0 ? (x) : -(x))

int16_t roll_value;
int16_t throttle_value;
int16_t m1Value;
int16_t m4Value;

int constrain(int amt, int low, int high);
```

```
int generatePWM(int amt);


/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/

	Motor.init(M1);		/*Initialize motor M1*/
	Motor.init(M4);		/*Initialize Motor M4*/
}


/*The function is called once before plutoLoop when you activate Developer
Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

	roll_value=0;
	throttle_value=0;
	LED.flightStatus(DEACTIVATE); /*Disable the default led behavior*/
}


/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/

	if (FlightStatus.check(FS_ARMED))		/*if drone is armed motors M1,M4
should be OFF*/
	{
		LED.flightStatus(ACTIVATE);		/*Enable the default led behavior*/
		Motor.set(M1, 1000);
		Motor.set(M4, 1000);
	}
	else
	{
		/*get data from user*/
		throttle_value= RcData.get(RC_THROTTLE);
		roll_value= RcData.get(RC_ROLL);
```

```
        }

        /*Condition for movement of wheels*/
        m1Value= (throttle_value-1500)+(roll_value-1500)/2;
        m1Value= constrain(m1Value, -500, 500);

        m4Value= (throttle_value-1500)-(roll_value-1500)/2;
        m4Value= constrain(m4Value, -500, 500);

        if (m1Value>0)
        {
                Motor.setDirection(M1, FORWARD);
        }
        else
        {
                Motor.setDirection(M1, BACKWARD);
        }
        Motor.set(M1, generatePWM(m1Value));

        if (m4Value>0)
        {
                Motor.setDirection(M4, FORWARD);
        }
        else
        {
                Motor.setDirection(M4, BACKWARD);
        }
        Motor.set(M4, generatePWM(m4Value));
}


/*The function is called once after plutoLoop when you deactivate Developer
Mode*/
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE); /*Enable the default led behavior*/
        Motor.set(M1, 1000);
        Motor.set(M4, 1000);
}
```

```
int constrain(int amt, int low, int high)
{
	if (amt>high)
	{
		return high;
	}
	else if (amt<low)
	{
		return low;
	}
	else
	{
		return amt;
	}
}


int generatePWM(int amt)
{
	amt= ABS(amt);

	amt=1000+(amt*2);

	return amt;
}
```

Running the project

- Build the code
- Flash the code on PlutoX
- Switch OFF Plutox and mount the hybrid wheels onto PlutoX.
- Connect the left wheel to motor driver M1 and the right wheel to motor driver M4.
- Switch ON PlutoX to initialize motor drivers M1 and M4.
- Turn the Developer Mode ON and ensure that the drone is not ARMed.
- Use the drone as a rover.
- Use throttle and roll for controlling the rover.
- To fly the drone, ARM it and start flying as usual. The rover mode will not function while the drone is flying.

Project Takeaway

This project used addon wheels to transform the drone into a rover, making it a hybrid drone. The major takeaway in this project is the calculation of the input power given to the motors M1 and M4. This enabled controlling the drone on land with just two motors.

The project focuses on making the drone a multi terrestrial vehicle, working both in air and on land. Such features could prove to be very important in applications such as rescue drones, delivery drones, etc. Interesting games can be developed using a multi terrestrial drone. Arenas could include flying tasks and driving tasks, making it more challenging and more fun!

PROJECT 12: AUTO-STABILIZATION (PlutoX)

Objectives

- Enhancing user experience
- P - controller
- Sensitivity control

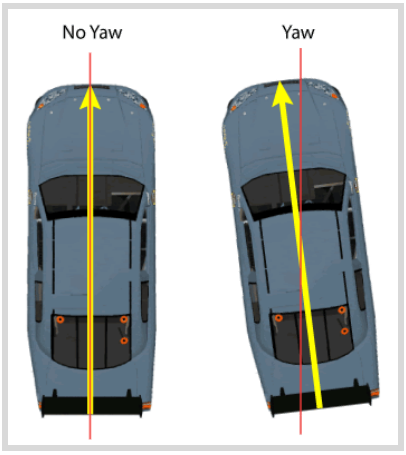
Problem statement

Upgrade the Hybrid Drone such that while in rover mode, it will always travel in a straight line.

Explanation

In the experiment 11, Hybrid Drone, it is evident from the execution of the project that the rover does not always travel in a straight line. It deviates from its path without any roll input which is not what a user would want. This project aims at understanding such issues and solving them, resulting in an enhanced user experience. The drone will be configured such that it will correct its path on its own so that it will travel in a straight line.

When the rover deviates while moving straight, it is the yaw angle of the drone that changes. This yaw angle will be used for the auto stabilization feature. Also, P controller will be used to compensate for the error.



Sensitivity control
The Pluto Controller app contains a sensitivity setting which is located in Control Settings. It contains three levels; min, mid and high. By default, the sensitivity is set to minimum.

The sensitivity control is for the roll and pitch joystick. Less sensitivity means a small change for a considerable stick movement. High sensitivity means large change even with a small stick movement. This feature works by altering the RcData range for roll and pitch.

As seen earlier, the RcData of roll and pitch range between 1000 - 2000. However, this range changes with respect to the sensitivity setting. For maximum sensitivity, the RcData range is 1000 - 2000. If these values are put together corresponding to the stick movement, it would look like this:

ROLL LEFT……...ROLL REST……..ROLL RIGHT
1000……..1500……...2000

The maximum stick movement will correspond to the extreme values of 1000 and 2000.

For minimum sensitivity, the range becomes 1200 - 1800. This simply means that it will require more stick movement to reach a particular RcData value as compared to the maximum sensitivity setting.

ROLL LEFT……...ROLL REST……..ROLL RIGHT
1200……..1500……...1800

This also means that even with maximum movement of the stick, the extreme mark of 1000 or 2000 will never be touched with minimum sensitivity.

Approaching the problem

The issue to be tackled in this project is deviation of the rover while moving in a straight line. When it deviates, the yaw angle changes. This means that there will be an error with respect to the yaw angle. The basic logic of this project will hence be to calculate the error and compensate for it.

To go on the next level of enhancement of the user experience, provide an option for the user to turn auto-stabilization ON or OFF. This can be done by using a switch. Since a separate switch is not available in the controls, use pitch input as the switch since pitch control will not be used in the rover mode. Using maximum sensitivity, if the value of pitch goes beyond 1930, then turn the auto-stabilization ON. If the value goes below 1070, turn the auto-stabilization OFF. The extreme values of 1000 and 2000 are not selected as the triggers because of the possibility of errors, which would not let the values reach those extremes.

If the auto-stabilization is turned ON, take a reading of the current heading, i.e. the current yaw angle. The next step is to check whether the deviation is because of error of the system or because of the roll input given by the user. If it is because of the error of the system, the deviation of RcData of roll from the rest value (1500) should ideally be zero and practically, it should be close to zero. Thus, if the absolute of the value of RcData of roll after subtracting 1500 from it is less than 30, then the deviation of rover is because of the error.

In such a case, calculate the error in the heading. The error is the difference between the reading taken while auto-stabilization was turned ON and the current yaw reading. The rover should return to its set heading from the shortest direction. This means that if the set heading is nearer by rotating clockwise instead of counterclockwise, then the program should ensure that the rover rotates clockwise to return to its set heading. This is done by setting a condition which will change the error value to that corresponding to the shortest direction.

There can be two ranges of error, depending on the direction of rotation of the rover. The first range is from 0 to +180 from one side and the second is from 0 to -180 which is from the other side. It should be noted that +180 and -180 are the same positions, both reached from either direction of rotation.

If the calculated error goes beyond +180, then subtract 360 from it. This will transform the value of error to the range of 0 to -180. If the calculated error goes beyond -180 on the negative side, then add 360 to it. This will transform the value of error to the range 0 to +180.

The final error is multiplied with the constant of proportional and added to 1500 to obtain the value for roll as in the previous project, Hybrid Drone.

After turning the auto-stabilization ON, if the value of RcData of roll after subtracting 1500 from it is more than 30, then the deviation of rover is because of the input from the user. In such a case, accept the roll input from the user and change the set heading. If the auto-stabilization is turned OFF, simply accept the roll values from the user.

The remaining part of the project remains the same as the previous project, Hybrid Drone. Calculate the input values for the motors M1 and M4 as done earlier, restrict them using the "constrain" function, set the direction of the motors and give the input for the motors generated from the "generatePWM' function.

Creating a new project

- Create a new PlutoX project
- Name it as "Auto-Stabilization"

Starting with the coding

Headers

Include the header "Control.h" to set the input values for motors M1 and M4. "Estimate.h" will give access to the current yaw angle. "Motor.h" will give access to the reversible motors. Including the header "User.h" will give access to the user inputs. Include the header "Utils.h" for debugging. Define the absolute function as done in earlier projects.

Declare the required variables. Set the initial value of auto-stabilization to false. Declare the functions "constrain" and "generatePWM". They are defined at the end of the entire program.

```
#include "PlutoPilot.h"
#include "Control.h"
#include "Estimate.h"
#include "Motor.h"
#include "User.h"
#include "Utils.h"

#define ABS(x) ((x) > 0 ? (x) : -(x))
```

```
bool isAutoStabilized=false;      /*To check if auto-stabilization is ON or OFF*/

int16_t setHeading=0;                   /*To store initial heading value*/
int16_t heading_error=0;          /*To store error value*/
int16_t k=10;                           /*Constant of proportional*/

int16_t M1_Value;                       /*Calculated input value for M1 */
int16_t M4_Value;                       /*Calculated input value for M4*/
int16_t M1_Valuef;                      /*generated PWM for M1*/
int16_t M4_Valuef;                      /*generated PWM for M4*/

int16_t Roll_value;
int16_t Throttle_value;

int constrain(int amt, int low, int high);
int generatePWM(int amt);
```

```
int constrain(int amt, int low, int high)
{
        if (amt < low)
                return low;
        else if (amt > high)
                return high;
        else
                return amt;
}
```

```
int generatePWM(int amt)
{
        amt= ABS(amt);

        amt=1000+(amt*2);

        return amt;
}
```

void plutoInit ( )

Initiate motors M1 and M4 separately instead of initiating all the reversible motors together. Use API “Motor.init( )” to initiate the motors.

```
void plutoInit()
{
/*Add your hardware initialization code here*/

        Motor.init(M1);             /*Initialize motor M1*/
        Motor.init(M4);             /*Initialize motor M4*/
}
```

void onLoopStart ( )

Deactivate the default LED behavior. Set the initial values of “Roll_value” and “Throttle_value” equal to zero.

```
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable the default led behavior*/
        Roll_value=0;
        Throttle_value=0;
}
```

void plutoLoop ( )

The first step remains the same, which is to check whether the drone is ARMed or not. The API “FlightStatus.check(FS_ARMED)” will return TRUE if the drone is ARMed. So, if the drone is not ARMed, it will return FALSE and negation (!) of that will be TRUE. **If** the drone is not ARMed, check whether the auto-stabilization is ON or OFF.

Get the RcData of pitch using the API “RcData.get()”. **If** the value is equal to or above 1930, then set the value of “isAutoStabilized” TRUE and using the API “Angle.get()” store the value of yaw angle at that moment into the variable “setHeading”. Turn the blue and green LEDs ON and red LED OFF. **Else if** the value of RcData of pitch is less than 1070, set the value of “isAutoStabilized” to FALSE and turn the red LED ON and blue and green LEDs OFF.

If the rover is auto-stabilized, then check whether the deviation is due to error or due to the input from the user. Use the API “RcData.get(RC_ROLL)” to get the value of RcData of roll. Subtract 1500 from it. **If** the absolute of this value is less than 30, then the deviation is due to the error. Calculate the error by subtracting the current yaw angle from “setHeading”. The error is stored in “heading_error” variable. **If** the value

of error is more than 180, then subtract 360 from it. **Else if** the value of error is less than -180, then add 360 to it. Multiply “heading_error” by proportional constant “k” and then add the product to 1500. Store this value in “Roll_value”.

Else, the deviation will be because of the user input. The rover should accept the user input. Use the API “Angle.get()” to store the value of roll in “Roll_value”. Set the current yaw angle as the new “setHeading” value.

The next part is for when the auto-stabilization is OFF. In that case, simply use “Angle.set()” API to store the value of roll in “Roll_value”. Thus, in all the above cases, the main task is to get the value of “Roll_value”, which will be used in further calculations.

The further program remains similar to the previous project, Hybrid Drone. Store the RcData value of throttle in the variable “Throttle_value”. Use the values of “Throttle_value” and “Roll_value” to calculate the values of “M1_Value” and “M4_Value”. While doing so, divide the value of “Roll_value” by 2 to reduce the sensitivity by a certain extent. Tinkerers can try the program without dividing by 2 to see the effect. Use the constrain function to restrict the value of both “M1_Value” and “M4_Value” in the range of -500 to +500.

Using the API “Motor.setDirection( )”, set the direction of motor M1 based on the condition that **if** the value of “M1_Value” is positive then the direction should be forward **else**, set it in reverse direction. Repeat the same for the motor M4.

After setting the direction, use the function “generatePWM” to get the values of “M1_Valuef” and “M4_Valuef”, which will be the input for the motors M1 and M4. Set the input value for motors M1 and M4 using the API “Motor.set( )”. Print the values of “M1_Valuef” and “M4_Valuef”.

If the drone is ARMed, then set the values of motors M1 and M4 to the minimum values.

```
void plutoLoop()
{
/*Add your repeated code here*/

        if(!FlightStatus.check(FS_ARMED))       /*If drone is not armed*/
        {
```

```
		if(RcData.get(RC_PITCH)>=1930)   /*Turn on Autostabilization*/
		{
			isAutoStabilized=true;
			setHeading=Angle.get(AG_YAW);/*take initial heading of the
drone*/
			LED.set(RED, OFF);
			LED.set(BLUE, ON);
			LED.set(GREEN,ON);
		}
		else if(RcData.get(RC_PITCH)<1070)       /*Turn Autostablization
OFF*/
		{
			isAutoStabilized=false;
			LED.set(RED, ON);
			LED.set(BLUE, OFF);
			LED.set(GREEN,OFF);
		}

		if(isAutoStabilized) /*If Autostabilized*/
		{
			if(ABS(RcData.get(RC_ROLL)-1500)<30)   /*If deviation is due to
system error*/
			{
				heading_error=setHeading-Angle.get(AG_YAW);
/*Calculate error(original-current heading)*/

				if(heading_error>180)
					heading_error=heading_error-360;

				else if(heading_error<-180)
					heading_error=heading_error+360;

				Roll_value=1500 +k*heading_error;
			}
			else    /*If deviation is due to user input*/
			{
				Roll_value=RcData.get(RC_ROLL);
				setHeading=Angle.get(AG_YAW);
			}
		}
		else    /*If Auto-stabilization is OFF*/
		{
```

```
                Roll_value=RcData.get(RC_ROLL);
        }

        Throttle_value = RcData.get(RC_THROTTLE);

        /*Condition for movement of wheels*/
        M1_Value = (Throttle_value-1500)+(Roll_value-1500)/2;
        M1_Value = constrain(M1_Value, -500, 500);

        M4_Value = (Throttle_value-1500)-(Roll_value-1500)/2;
        M4_Value = constrain(M4_Value, -500, 500);

        if(M1_Value>0)
        {
                Motor.setDirection(M1, FORWARD);
        }
        else
        {
                Motor.setDirection(M1, BACKWARD);
        }

        if(M4_Value>0)
        {
                Motor.setDirection(M4, FORWARD);
        }
        else
        {
                Motor.setDirection(M4, BACKWARD);
        }

        M1_Valuef =generatePWM(M1_Value);
        M4_Valuef =generatePWM(M4_Value);

        Motor.set(M1,M1_Valuef);
        Motor.set(M4,M4_Valuef);

        Monitor.print("M1:", M1_Valuef);
        Monitor.println("M4:",M4_Valuef );
}
else    /*If drone is armed*/
{
        Motor.set(M1, 1000);
```

```
                Motor.set(M4, 1000);
        }
}
```

void onLoopFinish ( )

Restore the default LED behavior. Set the values of motors M1 and M4 to the minimum value.

```
void onLoopFinish()
{
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);      /*Enable the default led behaviour*/
        Motor.set(M1, 1000);
        Motor.set(M4, 1000);
}
```

The entire code

```
/*Do not remove the include below*/
#include "PlutoPilot.h"
#include "Control.h"
#include "Estimate.h"
#include "Motor.h"
#include "User.h"
#include "Utils.h"

#define ABS(x) ((x) > 0 ? (x) : -(x))

bool isAutoStabilized=false;      /*To check if auto-stabilization is ON or OFF*/

int16_t setHeading=0;                    /*To store initial heading value*/
int16_t heading_error=0;          /*To store error value*/
int16_t k=10;                            /*Constant of proportional*/

int16_t M1_Value;                        /*Calculated input value for M1 */
int16_t M4_Value;                        /*Calculated input value for M4*/
int16_t M1_Valuef;                       /*generated PWM for M1*/
int16_t M4_Valuef;                       /*generated PWM for M4*/

int16_t Roll_value;
```

```
int16_t Throttle_value;

int constrain(int amt, int low, int high);
int generatePWM(int amt);

/*The setup function is called once at Pluto's hardware startup*/
void plutoInit()
{
/*Add your hardware initialization code here*/

        Motor.init(M1);             /*Initialize motor M1*/
        Motor.init(M4);             /*Initialize motor M4*/
}


/*The function is called once before plutoLoop when you activate Developer
Mode*/
void onLoopStart()
{
/*do your one time tasks here*/

        LED.flightStatus(DEACTIVATE);  /*Disable the default led behavior*/
        Roll_value=0;
        Throttle_value=0;
}


/*The loop function is called in an endless loop*/
void plutoLoop()
{
/*Add your repeated code here*/

        if(!FlightStatus.check(FS_ARMED))       /*If drone is not armed*/
        {
                if(RcData.get(RC_PITCH)>=1930)   /*Turn on Autostabilization*/
                {
                        isAutoStabilized=true;
                        setHeading=Angle.get(AG_YAW);/*take initial heading of the
drone*/
                        LED.set(RED, OFF);
                        LED.set(BLUE, ON);
                        LED.set(GREEN,ON);
```

```
                }
                else if(RcData.get(RC_PITCH)<1070)       /*Turn Autostablization
OFF*/
                {
                        isAutoStabilized=false;
                        LED.set(RED, ON);
                        LED.set(BLUE, OFF);
                        LED.set(GREEN,OFF);
                }

                if(isAutoStabilized) /*If Autostabilized*/
                {
                        if(ABS(RcData.get(RC_ROLL)-1500)<30)   /*If deviation is due to
system error*/
                        {
                                heading_error=setHeading-Angle.get(AG_YAW);
/*Calculate error(original-current heading)*/

                                if(heading_error>180)
                                        heading_error=heading_error-360;

                                else if(heading_error<-180)
                                        heading_error=heading_error+360;

                                Roll_value=1500 +k*heading_error;
                        }
                        else    /*If deviation is due to user input*/
                        {
                                Roll_value=RcData.get(RC_ROLL);
                                setHeading=Angle.get(AG_YAW);
                        }
                }
                else    /*If Auto-stabilization is OFF*/
                {
                        Roll_value=RcData.get(RC_ROLL);
                }

                Throttle_value = RcData.get(RC_THROTTLE);

                /*Condition for movement of wheels*/
                M1_Value = (Throttle_value-1500)+(Roll_value-1500)/2;
                M1_Value = constrain(M1_Value, -500, 500);
```

```
                M4_Value = (Throttle_value-1500)-(Roll_value-1500)/2;
                M4_Value = constrain(M4_Value, -500, 500);

                if(M1_Value>0)
                {
                        Motor.setDirection(M1, FORWARD);
                }
                else
                {
                        Motor.setDirection(M1, BACKWARD);
                }

                if(M4_Value>0)
                {
                        Motor.setDirection(M4, FORWARD);
                }
                else
                {
                        Motor.setDirection(M4, BACKWARD);
                }

                M1_Valuef =generatePWM(M1_Value);
                M4_Valuef =generatePWM(M4_Value);

                Motor.set(M1,M1_Valuef);
                Motor.set(M4,M4_Valuef);

                Monitor.print("M1:", M1_Valuef);
                Monitor.println("M4:",M4_Valuef );
        }
        else    /*If drone is armed*/
        {
                Motor.set(M1, 1000);
                Motor.set(M4, 1000);
        }
}

/*The function is called once after plutoLoop when you deactivate Developer
Mode*/
void onLoopFinish()
{
```

```
/*do your cleanup tasks here*/

        LED.flightStatus(ACTIVATE);      /*Enable the default led behaviour*/
        Motor.set(M1, 1000);
        Motor.set(M4, 1000);
}

int constrain(int amt, int low, int high)
{
        if (amt < low)
                return low;
        else if (amt > high)
                return high;
        else
                return amt;
}

int generatePWM(int amt)
{
        amt= ABS(amt);

        amt=1000+(amt*2);

        return amt;
}
```

Running the project

- Build the code
- Flash the code on PlutoX
- Switch OFF Plutox and mount the hybrid wheels onto PlutoX.
- Connect the left wheel to motor driver M1 and the right wheel to motor driver M4.
- Switch ON PlutoX to initialize motor drivers M1 and M4.
- Turn the Developer Mode ON and ensure that the drone is not ARMed.
- Give full pitch up input to turn auto-stabilization ON.
- Control the rover as done earlier. Notice the auto-stabilization.
- Give full pitch down to turn auto-stabilization OFF. Notice the change in motion of rover.
- To fly the drone, ARM it and start flying as usual. The rover mode will not function while the drone is flying.

Project Takeaway

User experience is one of the most important factors in any industry. Good user experience assures that the user will not face difficulty in using the product. In case of Hybrid Drone project, even though the idea was good, its execution highlighted the issue of deviation of rover from its straight path. Auto-stabilization was the solution to the problem, which was programmed in this project.

Any future project designed by a tinkerer might cause such minor inconveniences to the user. Continuous tracking of such problems and developing solutions towards them will ensure a good user experience.


DRONA
AVIATION
WWW.DRONAAVIATION.COM