IMU Communication and Data Collection

A sub-focus of the Autonomous Bicycle Project

Ruina Robotics and Biomechanics Lab Spring 2014

ABSTRACT

The goal of the autonomous bicycle project is to design and build a bicycle that can self-steer and stabilize. The Inertial Measurement Unit (IMU) is used to measure the roll angle and roll angular velocity of the bicycle for control purposes. In order to collect data quickly and accurately, a number of steps is needed. Serial communication must be established through programming languages running on the Beagle Bone Black. In addition, parsing algorithms and data packet formats are important to understand. This report summarizes the results and findings of working with the IMU, and aims to allow anyone to easily pickup where the project has left off.

Yu Meng Kate Zhou
Cornell University, 2016
Mechanical Engineering

Table of Contents

Introduction

The overall goal of the autonomous bicycle project has three stages. The aim to is build a bicycle that is able to:

1. Self-stabilize
2. Steer-by-wire
3. Track stand

For all three of these purposes, an Inertial Measurement Unit (IMU) is needed.

The IMU is a sensor that outputs the attitude of a craft using accelerometers and gyroscopes. The particular IMU used for this project is the Microstrain Inertia-Link. The purpose of using such a sensor on the bicycle is to measure the bicycle's lean angle and angular velocity. These data are needed in the dynamics control to keep the bicycle upright.

The goal of this part of the project is to have the IMU deliver necessary data quickly and accurately; all procedures of communicating to and getting information from the IMU should be done on the beagle bone black. This section of the report outlines the work that has been done towards this goal, and it includes a summary of what worked and what did not.

Communications Protocol

The starting place for working with the IMU is the communications protocol that the product comes with. This can also be found under Microstrain's support web page. Under documentation, choose the product, Inertia-Link, and refer to the Data Communications Protocol or the Software Development Kit. The protocol explains how to send information to the IMU and what do the information replied means.

Some general communication information should be highlighted. The IMU uses serial communication, which means that it sends information one bit at a time. These bits can be grouped into bytes; each byte is 8 bits representing a number between 0 and 255. Another representation of bytes is in hexadecimal, which is what the Communications Protocol uses. Similarly, any commands sent to the IMU are also in the same format. Not that the IMU does not use ASCII symbols, which is a way to represent letters and numbers; instead, it communicates directly in binary using floating point numbers. The baud rate of a serial device is the number of symbols sent per second, and in this case, Microstrain Inertia-Link's baud rate is 115.2 Kbps (kilo bits per second).

From the protocol, a data packet is a group of bytes. These bytes are placed in a particular order, so that users know what each byte means. A data packet always starts with a header, saying what kind of data this contains; an example is "Raw Accelerometer and Angular Rate Sensor Outputs". In the middle of the packet is the data itself; every four bytes represents one data value in the format of IEEE-754 floating-point number. Finally, at the end of a packet is a 16-bit checksum; this is an error detection method that ensures the combination of bytes sent previously is indeed a packet.

Communication can occur in 2 modes: Continuous and Poll. In continuous mode, the IMU remembers the command and continuously outputs data packets. In poll mode, the IMU would respond with one data packet for each command received.

Matlab Testing

Testing is done using Matlab 1) to ensure that the IMU is working, and 2) to applied what is learned from the documentations.

Setup

The IMU outputs information through RS232 Serial Communication. The hardware setup of the Matlab testing includes connecting the RS232 serial port to a serial-to-USB converter. The particular converter used is the Viewcon USB2.0 To RS232 Converter and its driver and user manual can be found on a pocket-CD that comes with the product.

Next, the name of the serial device can be found through the terminal application. In Terminal, to see a list of devices connect, enter the device manager by typing "ls /dev/tty.*". The converter should show up as

"/dev/tty.usbserial-xxxxxxxx", where "xxxxxxxx" is the serial number of the device.

Program Structure

Matlab code was provided by students who previously worked in the lab. Here is a breakdown of the code/psedoCode and what each part means and does.

1. Create an serial object
2. Initialize data collection matrices
3. Open serial port
4. Set IMU to fast speed and Continuous mode:
    a. Continuous mode
        i. Command byte 0xC4 (196)
        ii. Requires two additional bytes to confirm intent
    b. Fast mode
        i. fast mode is not a command instead we are writing the value 162 to EEPROM (Electrically Erasable Programmable Read-only Memory)
        ii. The command used is Write Word to EEPROM, command byte 0xE4 (228)
5. For Loop – read data 1 byte at a time
    a. If the byte read is the header
        i. Read the rest of the bytes that would up a packet (ie. if the packet has n bytes, read n-1 bytes)
    b. If checksum is correct
        i. Store this packet

Two functions are used in program:

1) "initiateIMU()" which is used to create a serial object and set its baud rate; it requires the exact name of the device as appeared in Terminal device manager as described above, and

2) “setIMU(serialobject, command) which is used to set the IMU to continuous or fast mode based on the communications protocol

The main Matlab functions used for serial communication are fread – to read a byte, fwrite – to send a byte to the IMU, fopen – to open the serial port, and fclose – to close the serial port.

Parsing

The parsing code was provided by students previously in the lab as well. What each byte means in a data packet is outlined in the communications protocol. The rest of the work that needs to be done is changing the data format.

One important thing to note is that a binary number can be reported in two ways: little endian- or big-endian. In little-endian binary format, the leftmost byte represents the smallest power of two; whereas in big-endian binary format, the rightmost byte represents the smallest power of two. The IMU uses little-endian format, while Matlab uses big-endian format, thus a conversion is required. Swapbyte is the function used to reverse the order of bytes in an array in Matlab.

To calculate time:

1. Take out the 4 bytes that represent the time value from the packet
2. Reverse the order of the 4 bytes
3. Change type of the numbers to uint32 (32 bit unsigned integer)
4. Typecast again to single
5. Divide value by 19660800 (formula from communication protocol)

To calculate checksum:

1. Take out the 2 bytes the represent the checksum from the packet
2. Reverse the order of the bytes
3. Change type to uint8 (8 bit unsigned integer)
4. Typecast to uint16

To calculate a data value (i.e. angle):

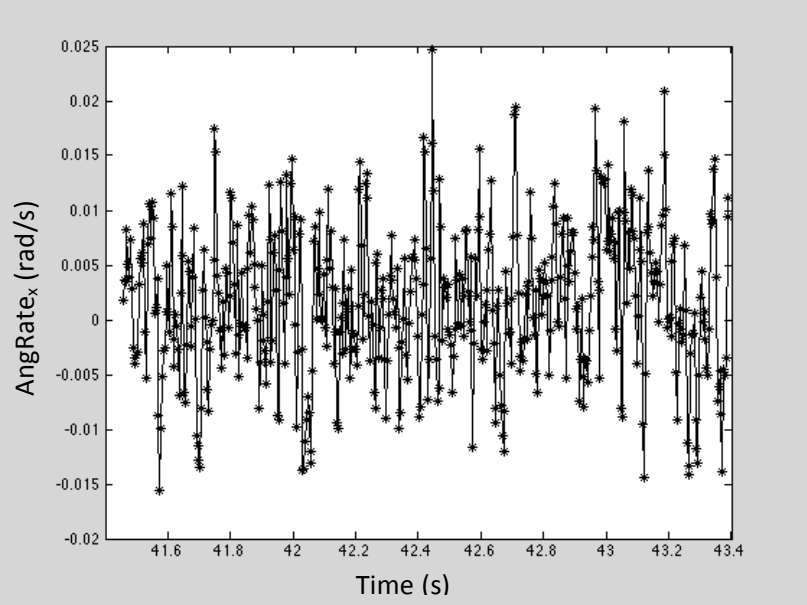
1. Take out the 4 bytes the represent the data value from the packet
2. Reverse the order of the 4 bytes
3. Change type of the numbers to uint32 (32 bit unsigned integer)
4. Typecast again to single

Results & Graphs

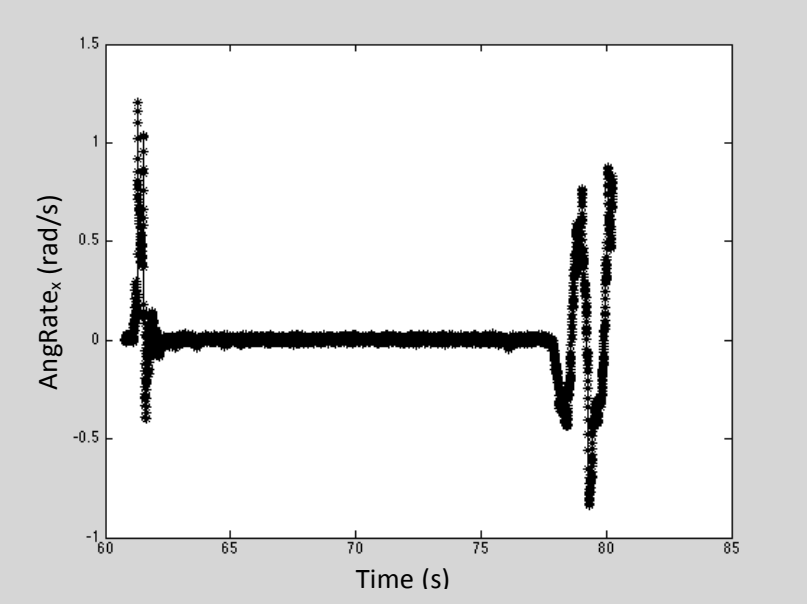
The graphing portion of the code is not in real time; it plots all the data points and their corresponding time values after the serial port closes. Nonetheless, these plots can be used as a sanity-check to make sure that the values produced make sense.

Continuous Mode:

Graph 1: Laying the IMU flat without any movement, we see slight noise produced. The claimed accuracy that came with the product is +- 1 deg=0.017 radians, which is around how much the following data points deviate from 0.

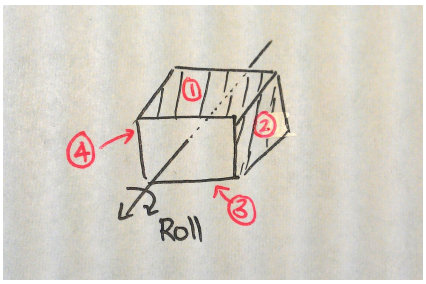


Graph 2: This plot is created from first placing the IMU flat and then turning it. This is done to compare the data plotted for no motion vs. yes motion.



Poll Mode:

Testing was also done in poll mode. In poll mode, the roll angle of the IMU is plotted because we can have a better idea of what is the accurate roll angle as opposed to what angular rate we are turning the IMU at.



Above is a sketch of the IMU modeled as a rectangular prism. Each of the four sides parallel to the roll angle axis, is shaded and labeled in in red. The roll angles of the IMU when placed on each of the four sides are plotted. It is expected that adjacent side roll angles should be off by $\frac{\pi}{2}$, while opposite sides should be off by π.

The results match the expectation and show that the program structure in poll mode reads data correctly. Not all data and calculations are needed to demonstrate this point, but some examples are shown below. One of the possible reasons of why the angles do not match up exactly to what is expected is the unevenness of the IMU casing. More testing and trials are needed to find other reasons contributing the larger error in Sample Calculation 2.

Sample Calculation 1:

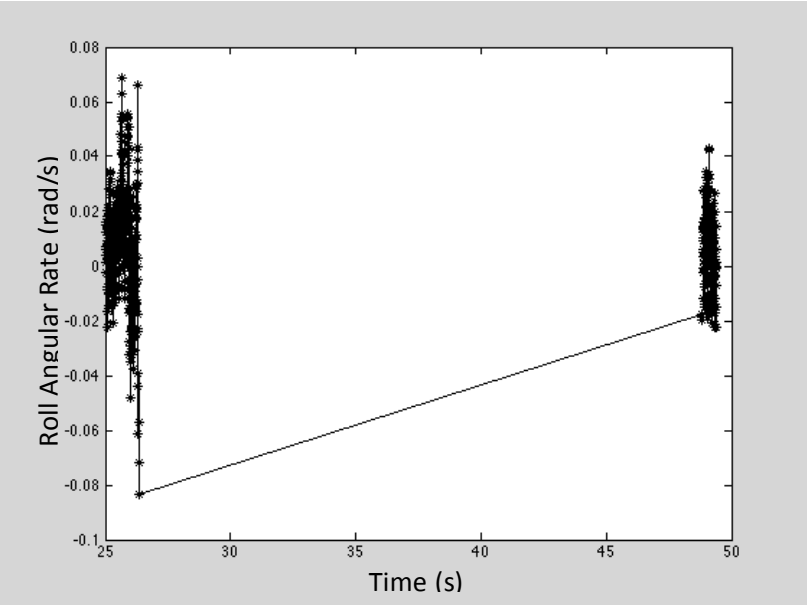
$$average\ angle\ read\ on\ side\ 1 = 0.25\ rad$$
$$average\ angle\ read\ on\ side\ 3 = -2.91\ rad$$
$$0.25 - (-2.91) = 3.16\ rad \sim \pi\ rad$$
$$3.16 - \pi = 0.02\ rad$$

Sample Calculation 2:

$$average\ angle\ read\ on\ side\ 2 = 1.48\ rad$$
$$average\ angle\ read\ on\ side\ 3 = -2.91\ rad$$
$$2.91 - 1.48 = 1.43 \sim \frac{\pi}{2}$$
$$\frac{\pi}{2} - 1.43 = .14\ rad$$

Debugging

Although most graphs generated are reasonable, some did not. There are graphs with regions with missing data points because of lost packets.



Using the debug mode of Matlab it is found that some data points are not plotted because of invalid checksums. The checksum is a 16 bit (2 bytes) number is equal to the sum of all preceding bytes between 0 and 65535 with rollover. The previous Matlab code did not account for the rollover of the checksum, that is if the sum of all previous bytes is 65537, it is equivalent to 1.

Taking account of the checksum rollover resolved some of the lost packet problems, but there are still occasional gaps in data. One suspicion of the problem is desynchronized data packets, and it is very likely to happen when first starting in continuous mode. Below is an example that helps to explain this problem.

To make the problem clear, let's use some simplifications. Assume that the packet we are interested in has 3 bytes. Let  be a byte, and  be one data packet. Different packets are represented by different colors; for instance  are two different data packets sent one after another. For simplicity, the three bytes within a packet will not be differentiated, we will assume the first in the trio is the header, and the last is the checksum.

With the current algorithm, data is process accurately if the information received looks like:

In this case, a header  is first detected and the following 2 bytes,  are read. The program put the three bytes together, and check to see that  is indeed a valid packet. Next, the code will check for a header again and will successfully read the next two packets.

However, the problem occurs when the bytes received look like

In this case, the program will take the first three bytes,  , and find that it is not a valid packet. The program will discard of everything just read and check the following three bytes

 , and once again the packet is invalid. This pattern continues and results in a long period of no data (demonstrated by the graph above).

It is true that the program checks for the accurate header in the beginning to synchronize the parsing; however it can easily be desynchronized if any byte that is not a header happens to have the same value as the header.

In poll mode, this problem was not seen because the IMU only sends one packet at a time. This problem is later resolved by using a queue-like data structure in Python described below.

Python – Pyserial

On the actual bicycle, the IMU will communicate through Python, a programming language running on Beagle Bone Black (BBB). Pyserial is the serial interface library for Python, and it is used in this project.

Set up

In the testing phase, the IMU is still connected to the computer through the serial-to-usb adapter. On the bicycle, the IMU's serial port will be connected a BBB cape which converts serial information from RS 232 to UART. In both cases, the python code will run in Terminal. During testing, the code is developed using Komodo Edit text editor.

The Pyserial library can be downloaded from https://pypi.python.org/pypi/pyserial. To install, type the following commands into the Terminal application

1. Navigate the directory to where pyserial was downloaded using "cd"
   Ie. If you want to go to the desktop, type "cd desktop"
2. Type "sudo python setup.py install".

Two ways of testing whether serial communication is successfully established through Python include connecting the serial port to an oscilloscope and to another computer.

Pyserial Functions

The main Pyserial functions used include:

- Read(n) : read n bytes
- Write(data): send data

The IMU itself does not use ASCII communication, but these Pyserial commands do interpret the message sent and read as ASCII symbol. For instance, if you would like to send "2" to the IMU, using write(2) will send the number that the ASCII symbol 2 stands for. Instead, the function chr(2) is used to find the ASCII symbol for 2. For similar reasons, the function ord() is used to change the ASCII symbols received from the IMU to integer values.

It is important to note that this exchange between ASCII and integer is strictly used before Pyserial sends the data and after Pyserial receives the data; nothing to do with the communication methods of the IMU.

Program Structure

To solve the problem with unsynchronized described above, a new data structure is implemented. This data structure acts as a buffer, but it is not the same as the built-in hardware buffer of the computer or the IMU. This data structure will be referred to as the "buffer" or "circular buffer" in the following explanation.

Below is a general structure of how the program works:

Let n be the length of the desired data packet in number of bytes

1. Write command to IMU asking for packet
2. Read all available
3. Place all in buffer
4. If the buffer contains more than n bytes
    a. look at the first n bytes without taking them out of the buffer
    b. make a packet object out of the first n bytes
    c. verify that the packet is valid
        i. if valid -> take these bytes out of the buffer and read the data values
        ii. if invalid -> take out the first byte, and repeat from step 1.

Note: this program is currently used for poll mode, but the following data structure explain should make the program applicable to both poll and continuous mode.

The circular buffer is a first-in-first-out data structure; it is implemented from a Python list. It is named circular because it is like a fixed-size list that is pinned end-to-end; thus the old data can be overwritten when it overflows. It contains the following functions:

- append (x): add object x to the "end" of the buffer (no real end as it is a circle, but next to the previous object added, or it buffer overflows overwrite the oldest object)
- poll() : take out the oldest object in the buffer
- peek(num): look at what the oldest num objects are without taking them out.

Here is an example to illustrate what these functions do. Let rbuffer be a circular buffer object of length 3

Command	What is in the buffer?			Output/return	
rbuffer.append(1)	1				
rbuffer.append(2)	1	2			
rbuffer.peek(1)	1	2		2	
rbuffer.poll()	1			2	
rbuffer.append(3)	1	3			
rbuffer.append(4)	1	3	4		
rbuffer.append(5)	5	3	4		
rbuffer.peek(2)	5	3	4	3	4
rbuffer.poll()	5		4	3	

Note: the empty boxes do have values of zero, but because of the pointer locations they essentially do not exist.

Another class used in this program is the Packet class. It simplifies the main loop of the code by avoiding confusion with calculating checksum and parsing.

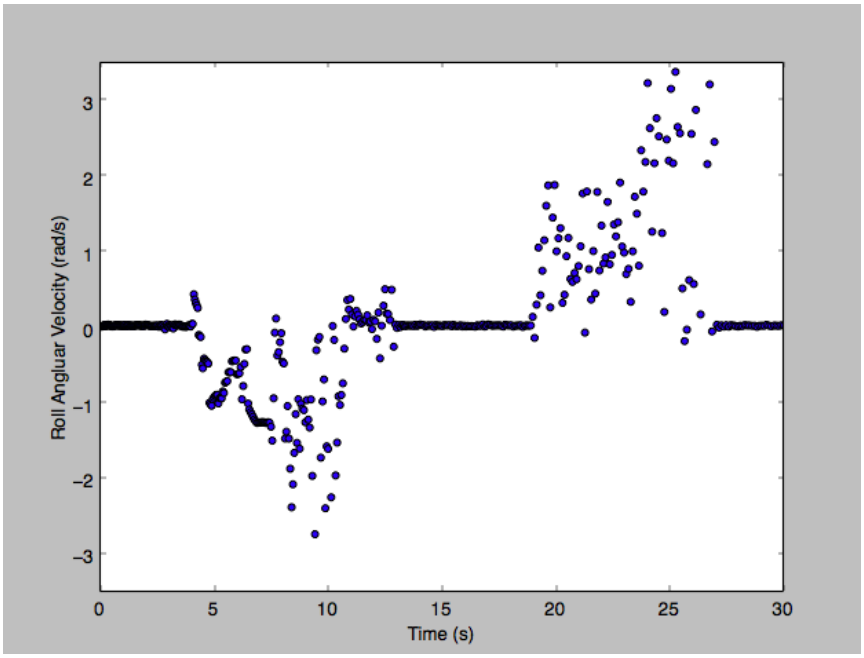
The Packet class has fields bytes (array of all bytes), checksum(int representing the checksum) and datatype (int representing the header number). It also has the isValid() function, which returns true of false based on whether the checksum is correct. Finally, there are functions that return the data values, such as roll angle, roll angular rate etc.

The data type conversion to floating point number is done through the library numpy. The order of the bytes is first reversed, then the bytes are converted to 8 bit integer values and finally to floating point 32 bit numbers. This procedure is similar to what is done in Matlab.

Results and Testing

Many tests are done to ensure that the data outputted is accurate. Testing is done to determin, the pitch, roll and yaw axes. One specific testing is outlined here:

The following graph is generated from first turning the IMU in one direction along its roll axis, putting the IMU down, and then turning it in the opposite direction.



The y-axis is the roll angular rate in radians/sec, and the x-axis is time in seconds. The result matches expectation. The plot starts off at 0 rad/s as the sensor is being placed flat on the table. In the next section (4-12s), almost all points are negative as the sensor is being turned counter clockwise. There are some positive values from the unsteadiness of the hand when picking up and placing down the sensor. The flat line at 0 occurs again from 12-18s as the IMU is being placed flat. From 18-26s, the IMU is being turned clockwise, and positive roll angular rate values are produced. Finally, from 26s to the end, the roll angular rate is zero, as the IMU is placed flat.

Real time plotting

In order to further ensure whether the values outputted are accurate, real-time plotting is implemented. This is done through the library Matplotlib.pyplot. The scatter() function is used to scatter plot the data points, with no lines connecting the adjacent points. The function draw() is used to redraw the current figure without creating a new window.

The link for the video showing the real-time angle plotting is https://www.youtube.com/watch?v=872K1Qaw8Oc&feature=youtu.be. In this video, the IMU is turned along its roll angle and it can be seen that the angle decreases to $-\pi$ and then loops around starting from π going down again. One complete revolution around its roll axis is reflected by the range of points from $-\pi$ to π.

When using real-time plotting, there is a delay observed. One contribution to this delay is the plotting functions. Using the clock function, part of the time library in Python, the time needed to read one data point and plot it is 0.06 sec; while the time it takes to only read a data point ranges from 0.002 to 0.007 seconds. There may be other reasons for the delay; mores testing is needed.

Next Steps

The main accomplishment of this semester is being able to read accurate data in real time; however, timing is still a concern. As shown in the lag from the real-time plotting video, more testing is needed to eliminate the time delay. In addition, more testing using continuous mode in Python can be done to see whether the time delay will be reduced.

Appendix A: Final Presentation

Attached at the end of the report.

Appendix B: Code Documentation

See attached code in zipped files

1. Matlab: parseAccData2.m
2. Matlab: Rea_Angular_Velocity.m
3. Matlab: Polling.m
4. Python: ReadIMUPY.py

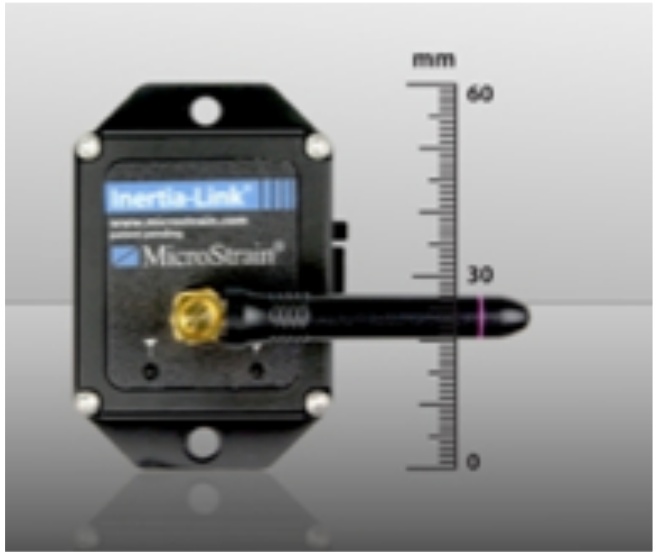
Note:

- 1. is part of previous work done the lab.
- 2. is an edited version of previous work in the lab.
- 3. and 4. are newly written.

Kate

IMU

Early Stages

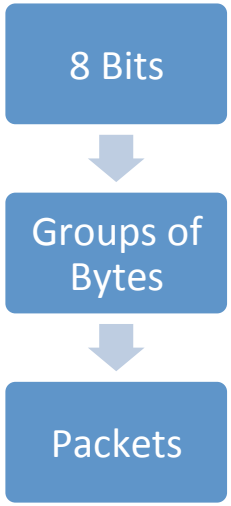


Selection: Microstrain Inertial-Link

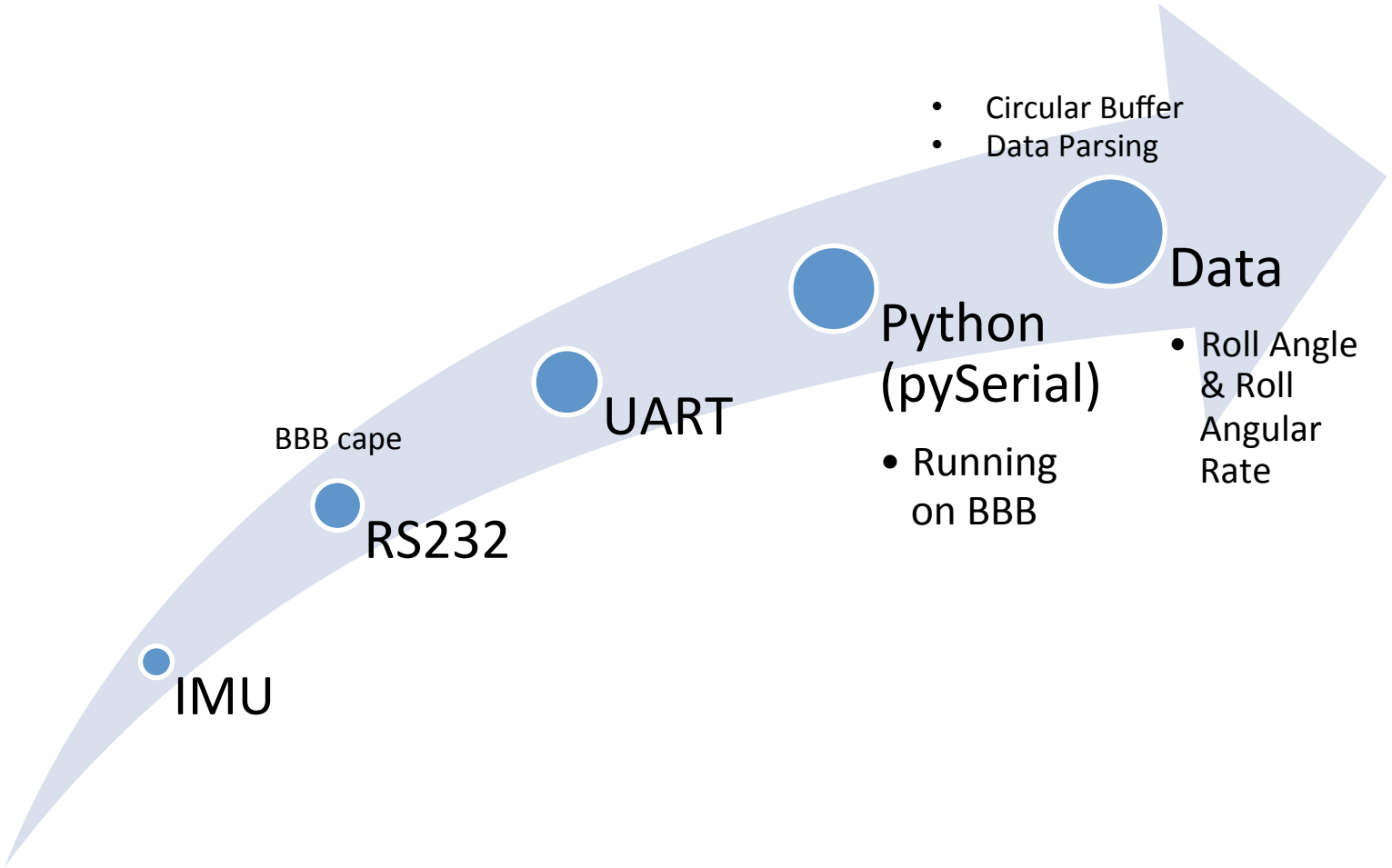
Baud rate: 115200 bps
Interface: RS-232
Claimed accuracy: ±1 degree

Communications Protocol

- Modes: poll vs. continuous
- Command references
- Data Packets

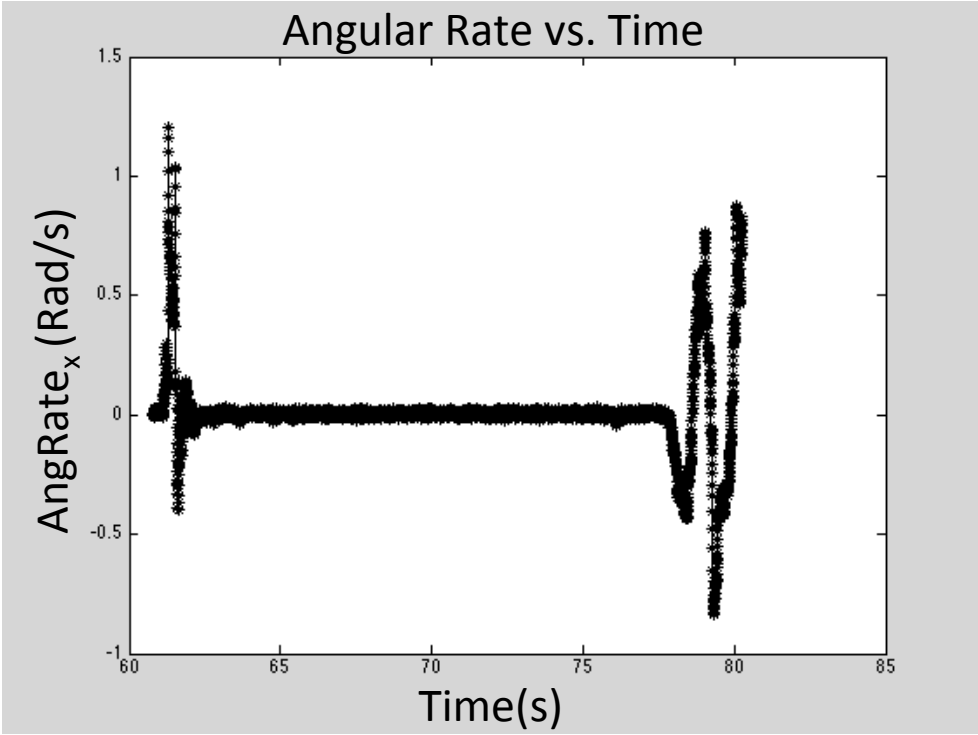


Roadmap


IMU
BBB cape
RS232
UART
Python (pySerial)
• Running on BBB
• Circular Buffer
• Data Parsing
Data
• Roll Angle & Roll Angular Rate

Matlab Testing

- Purpose:
  - understand modes
  - learn commands
  - understand data formats
  - parsing (checksum calculation)
- Interface:
  - RS232 to USB converter
- Modes:
  - Polled and Continuous



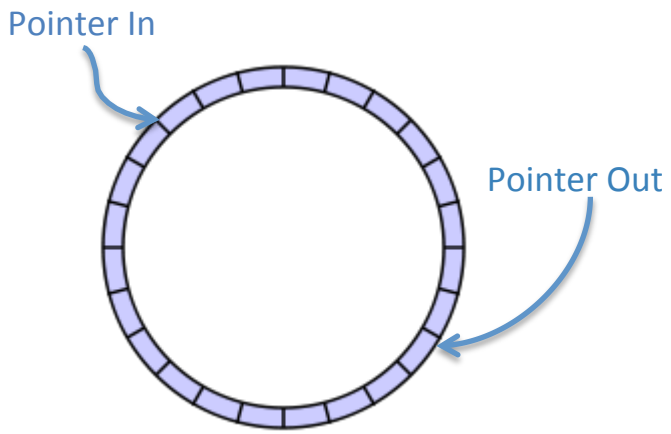
Using Python

- Pyserial:
  - Library that provides support for serial communication

- General Program Structure:
  - Ask for data
  - Put all received in circular buffer
  - If enough bytes in buffer, take out, validate and parse
  - If not, go through loop again

Circular Buffer

- What is it?
  - Fixed Size Data Structure Connected End-to-end
  - Not Computer Or IMU Hardware Buffer

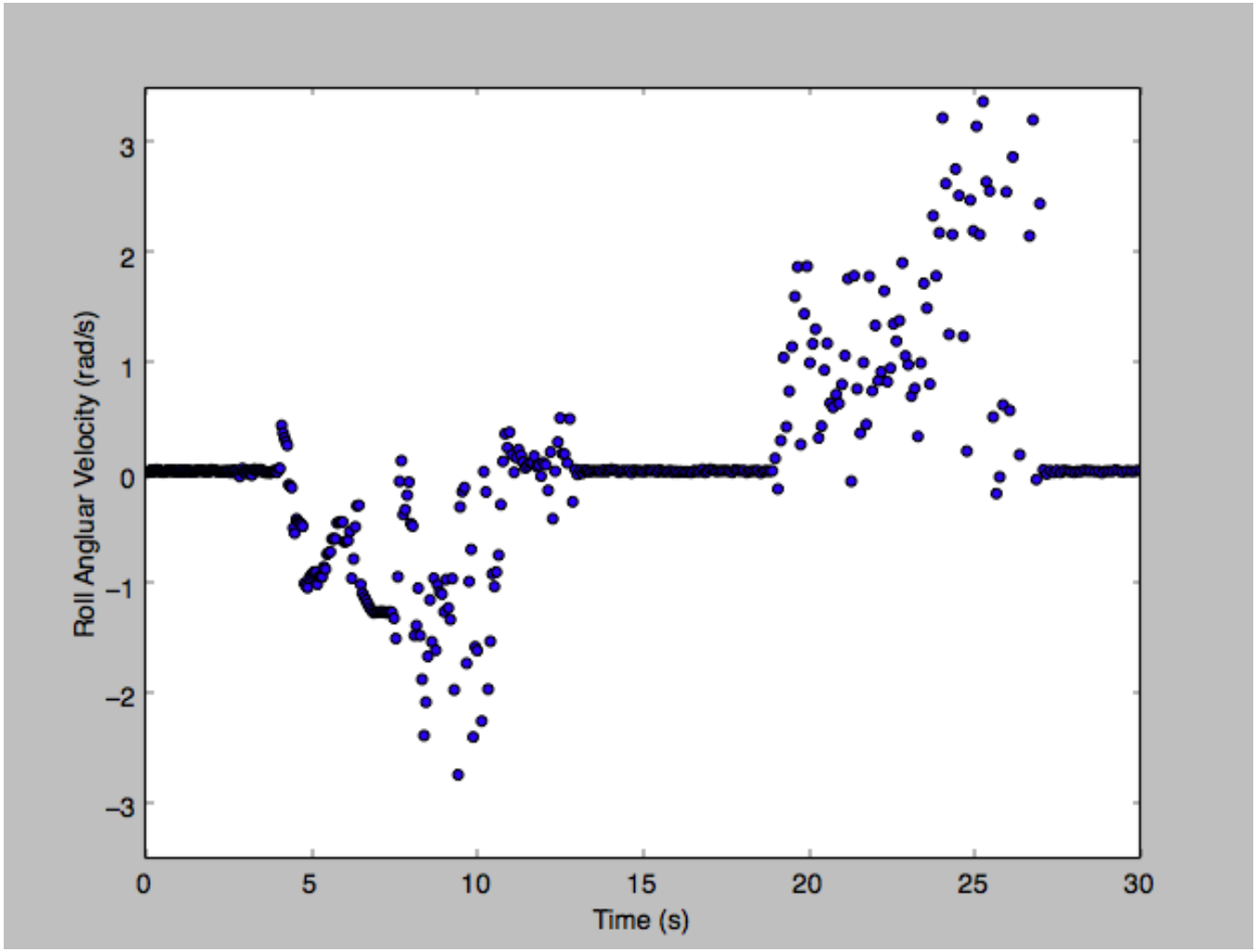


- Why Use It?
  - Overwrites Old Data
  - Compatible With Poll And Continuous Mode
  - Know The Size

```
# FIFO Data structure
class Queue:
    def __init__(self, size_max):
        self.max=size_max
        self.data=[0 for i in range (size_max)]
        self.Pin=0
        self.Pout=0
        self.isempty=1
```

Video: Roll Angle Demo

Roll Angular Velocity



Plotting Lag Problem:

With plotting:

0.06 sec/data pt.

Without plotting:

0.0002 - 0.007 sec/ data pt.