

reWRITE

EE149/249A Project Report, Fall 2015
Reia Cho, CJ Geering, Nathaniel Mailoa, Rachel Zhang



I. INTRODUCTION

INSERT TEXT HERE

II. BILL OF MATERIALS

| Component | Price |
|--|-------|
| LightBlue Bean | \$30 |
| BNO055 Absolute Orientation Sensor | \$35 |
| Li-Ion 3.7V 150mAh Battery and Charger | \$13 |
| Force Sensor | \$7 |
| Button and LED | \$2 |
| 3D Printing | \$2 |
| Total | \$89 |

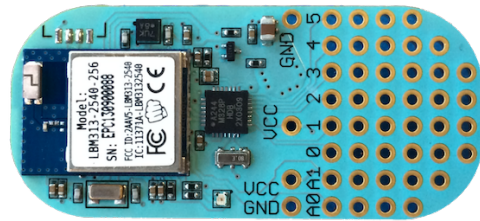


Fig. 2. LightBlue Bean

III. SYSTEM

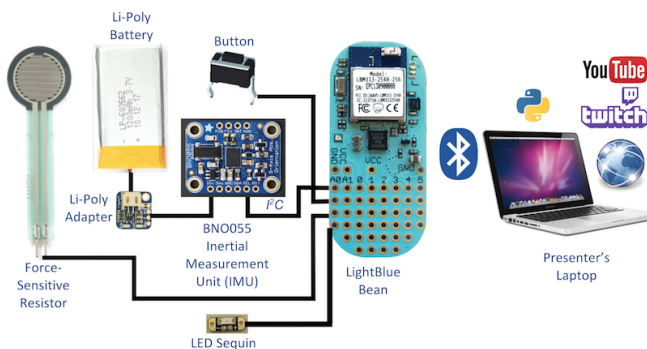


Fig. 1. reWRITE system

INSERT TEXT HERE

A. LightBlue Bean

For the main controller and communication, we initially chose to use the RedBearLab BLE Nano, but ran into issues with the low level BLE protocol. Because the BLE Nano is mainly used to communicate with a smartphone or other BLE modules, we had to develop an XCode project on OSX to use it. Before we developed the project, we searched for a high-level software packet on the PC side, but found none. The BLE nano also had limited pinouts on the breakout board and did not have an onboard LED. The LightBlue

Bean turned out to be a more reasonable platform. The Bean has an 8MHz ATmega328p microcontroller, a LightBlue LBM313 Bluetooth Low Energy (BLE) module, an RGB LED, a temperature sensor, and a 3-axis accelerometer. Unfortunately, it does not have a gyroscope, so we still need a separate IMU module. The Bean runs on a 3V CR2032 coin cell. It also has a pin that can be used to provide power, which we used in our design. The BLE connects to PCs

wirelessly and is easy to program because LightBlue provides libraries that simulate serial communication through a virtual serial port.

Because the Bean and Arduino Nano have similar functionalities and microcontrollers, we used the Nano for early prototyping. We were able to port the Arduino code to the Bean without any compatibility issues.

B. BNO055 Absolute Orientation Sensor Breakout Board

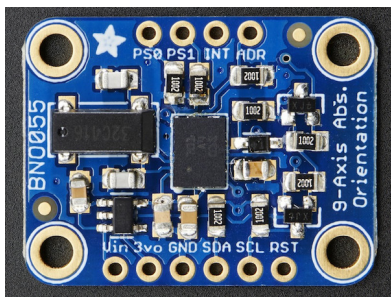


Fig. 3. BNO055 IMU breakout board

This BNO055 is a 9-Degree of Freedom Inertial Measurement Unit (IMU) that provides accelerometer, gyroscope, and magnetometer data. Running onboard the 32-bit M0+ microcontroller is the Bosch Sensortec sensor fusion algorithm that combines sensor data for some calibration and filtering. This gives us linear accelerations because the sensor fusion accounts for gravity. The processed data is available at a data rate of 100Hz. We also used this device to send Euler angles, which is calibrated against the magnetometer to provide absolute orientation with respect to North. The IMU communicates to the Bean through I2C in the Beans analog pins. When the device is initially powered, it must be calibrated before the data is useful. As each sensor is calibrated separately by the onboard microcontroller, a register in the device stores the calibration status for each sensor out of 3. The gyroscope is calibrated by keeping the IMU stationary. The magnetometer is calibrated by drawing a figure eight. The hardest sensor to calibrate is the accelerometer. To do so, we had to keep the IMU stationary for a few seconds in various orientations. Overall, the three sensors take about

30 to 60 seconds to calibrate, depending on how familiar the user is with the calibration technique. Since it is a bit tedious to do, we tried to save the calibration on the Bean and load it to the IMU on reset, but that does not work well because the calibration is not constant over time and environment. To help with calibration, we use the Beans RGB LED to signify the calibration status using color-coding. Even after we receive a full calibration, it is still not perfect, as will be discussed in the position reconstruction section.

C. Li-Ion Battery

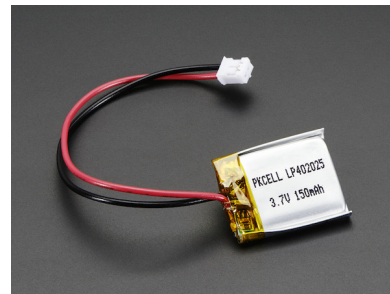


Fig. 4. 3.7V 150mAh Li-Ion battery

To power all of our devices, we use a 3.7V 150mAh Li-Ion battery with a breakout board. We chose the relatively small capacity to fit the form factor of the reWRITE. The LightBlue Bean, however, needs a 3.0 to 3.6 V source since it does not have an onboard regulator. Luckily the IMU breakout board has a 5V to 3.3V regulator, so we power the Bean up through the IMU board.

D. Force Sensor



Fig. 5. Force sensor

To detect when someone is writing with the marker, we attached a force sensor to the back of

the marker casing such that the marker pushes against it when it is in use. The force sensor is, effectively, a variable resistor – about 8 KOhms when not pushed and about 2 KOhms when pushed. We used a resistive divider circuit and a `digitalRead()` function on the LightBlue Bean to determine when the marker was being used. To prevent the marker from constantly pushing the force sensor when the user is not writing, we added a spring between the force sensor and the marker.

E. Button and LED Sequin

Our last two peripherals were a button and an LED Sequin. The LED Sequin is a tiny module containing a surface-mount LED and resistor and was used for user feedback to show whether or not the force sensor was being pushed. The reWRITE also has a small tactile button used a resistive divider circuit similar to that of the force sensor. This lets the user signify that the marker was ready for position calibration (to establish a point of origin). It was also used to clear the plotting canvas in the reconstruction.

IV. CASING

For our project, we needed a custom marker casing to attach a force sensor and mount our sensors, peripherals, and microcontroller. Because of cost, ease of use, and endless design possibilities, we chose to 3D print a casing. To make the 3D model, we used Autodesk's Fusion 360 3D CAD/CAM software and the LulzBot TAZ 5 3D printer. For filament, we initially tried ABS plastic, but we switched to PLA plastic to alleviate warping issues. Our first version was a threaded design with three components: the end cap, the body, and the front cap. To optimize printing time, we printed the body on its side. Each component had support material to retain structure. However, printing threads proved to be difficult to do well. Thus our second version was a snap lock design with the same three main components. This design was more simple but to improve printing quality, we printed the body in two parts and added more support material. With the finished print, we used hot glue to fix the force sensor with the spring on the end cap, and

secured the marker in the casing with the front cap. Unfortunately, the design does not allow the marker to be capped while it is in the casing.

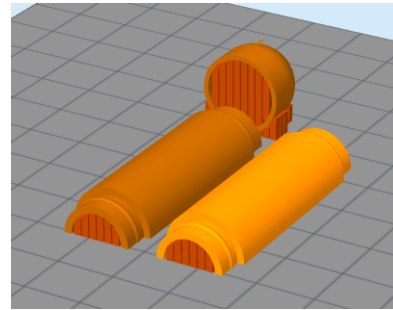


Fig. 6. Casing model

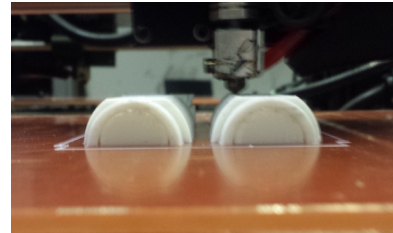


Fig. 7. 3D printing the casing

V. POSITION RECONSTRUCTION

To digitize writing with IMU data, we had to develop position reconstruction using various methods of post-processing. The reconstruction and plotting code is written in Python with NumPy and Matplotlib libraries, and is provided in the appendix.

A. IMU Sensor Data

The Bosch IMU provides accelerometer, gyroscope, and magnetometer data. The on-chip Sensortec sensor fusion subtracts the gravity vector to provide linear acceleration. It also uses proprietary algorithms to fuse all nine degrees of freedom to provide more reliable data. The sensor fusion with the gyroscope and the magnetometer provide absolute orientation to North as well as gyroscope stability. From this, we are able to use the Euler angles in position reconstruction. Yaw (rotation about the z-axis) and pitch (rotation about the y-axis) have ranges of 360 degrees, whereas roll (rotation about the x-axis) has a range of 180 degrees.

B. Filtering and Thresholding

Once the LightBlue Bean has transmitted the IMU data, we must filter it because the on-chip sensor fusion is not enough for position reconstruction. First we noted that the IMU calibration was not always perfect. Figure [?] the raw data obtained from the IMU on a trial run when it is stationary. As seen in the green line in the plot, the data drifts and gets periodically recalibrated around once every 10 seconds. To solve this issue, we implemented a high pass filter by subtracting the reading away from some mean value computed through a low-pass infinite impulse response (IIR) filter with a very low cutoff frequency. The high-pass filter eliminates this offset as seen in figure [?]. High pass plots Next, since the data is noisy, we implemented an IIR low pass filter, which significantly reduces the noise as shown in figure [?]. Low pass plots Lastly, we implemented acceleration thresholding: if the measured acceleration is under a certain threshold, we assume it is due to noise and ignore it. Unfortunately, this thresholding makes it difficult to recognize slow movement. Therefore, we require the user to write with fast movements.

C. Transformation to Fixed Reference Frame

Since the acceleration data provided is in the frame of reference of the IMU, we needed to perform a change of basis to real world coordinates. The fixed frame of reference is chosen to be the first position recorded. To perform this basis transformation, we used the Directional Cosine Matrix from [reference here]. QG is the coordinates from the fixed reference frame while QP is the coordinates from the IMU reference frame.

D. Velocity Adjustment

As we collected acceleration data and tried to reconstruct the position, we discovered that we needed to adjust velocity. When acceleration is near zero for some period of time, it is highly likely that the marker is stationary – humans typically cannot produce perfectly zero acceleration in a reasonable amount of time. The

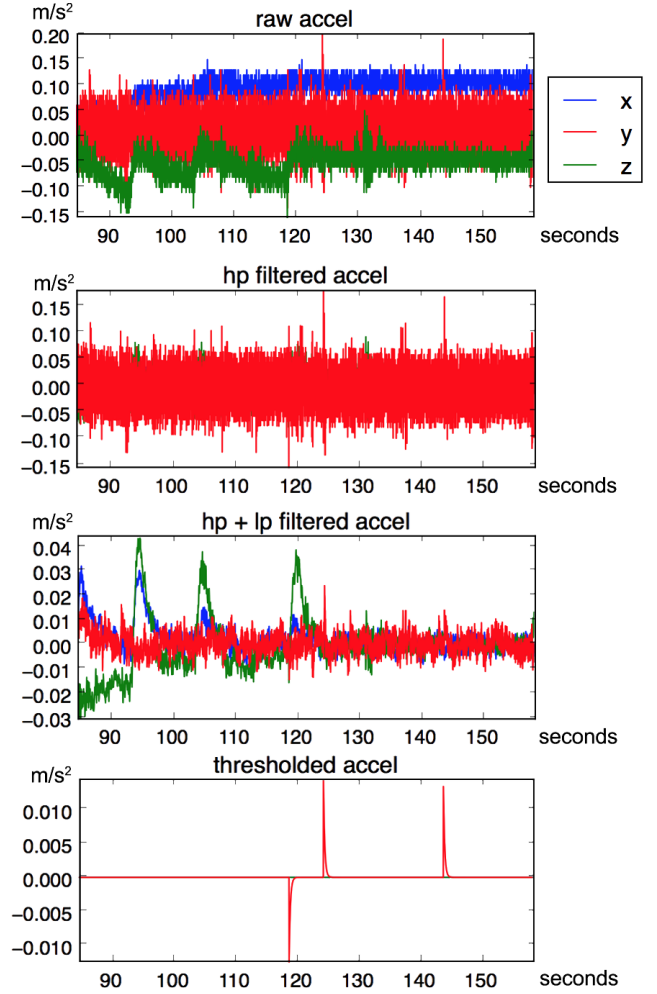


Fig. 8. (a) Raw data from IMU; (b) Processed with high pass filter; (c) Processed with low pass filter; (d) Thresholded

IMU data is sampled as is imperfect, which causes the velocity to be nonzero as shown in figure [?], which shows velocity reconstruction for a step in the y-axis direction. To counter this issue, we assumed that if absolute acceleration is within some small bound for a set time, the velocity should be zero. Then we shift the velocity towards zero by comparing its current value to its value the first and last time the acceleration left the acceleration bounds near zero. Assuming i is the index when the acceleration leaves the bounds and j is the index when the acceleration enters the bounds, the velocity warping formula we used is shown below:

$$v_{\text{adjusted}}[n] = v_{\text{unadjusted}}[n] - \left(\frac{n-i}{j-i} \right)^2 v_{\text{unadjusted}}[j]$$

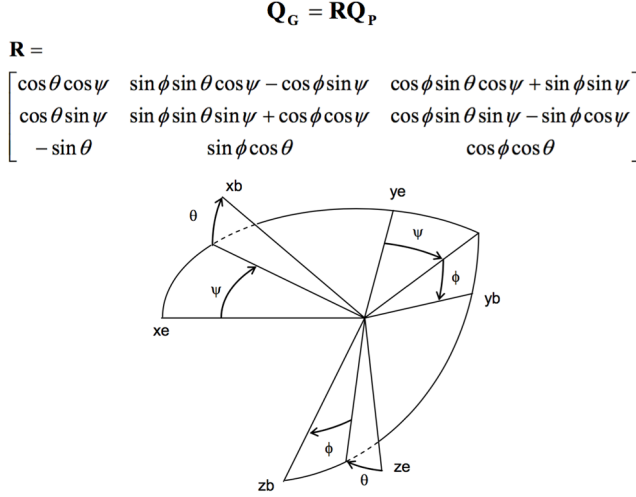


Fig. 9. Angles corresponding to the DCM [Premerlani and Bizard]

This algorithm is executed on the 3 axis of movements independent from each other and ensures that the velocity is back to zero when the acceleration is near zero. The resulting velocity is shown in figure [?].

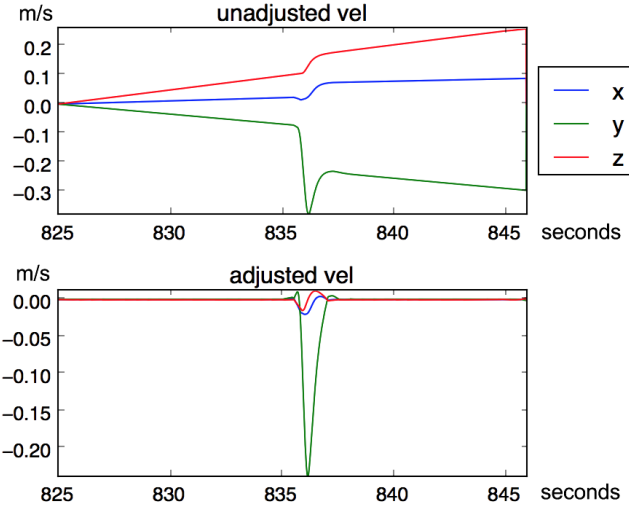


Fig. 10. (a) Velocity integrated from acceleration data; (b) After velocity adjustment

E. Tip Position Reconstruction

Because the IMU was mounted on the end cap of the marker casing, we used the Euler angles and the distance between the IMU and marker tip to compute the position of the tip. Since the IMU is fixed in the end cap, we could assume that the marker would be along the z-axis of the

IMU. With that and the fact that we are drawing on a 2D-plane, the yaw reading from the IMU did not matter. We calculated that if the reconstructed position of the IMU was (x, y) , the tip position was $(x - \sin(\text{pitch}), y - \sin(\text{roll}))$.

F. Plotting

Once the position was reconstructed, it was plotted. However, to reduce latency, since NumPy uses vector computations, we perform the post-processing every 20 data points. If the force sensor is turned off in one of the 20 data points, the plot is updated to include the last stroke. Although we reconstruct position for all transmitted data, we would only plot position when the force sensor was pressed. We also would clear the figure showing the position if the user pressed the button.

VI. QUANTITATIVE ANALYSIS AND SCHEDULING

A challenge we encountered was scheduling. omg i cant write good intro sentences to save my life. The Bean provides the Arduino serial library through a virtual serial port, but it has a limited data rate. Our target data rate was approximately 67 Hz (1 sample every 15 ms). We calculated the worst case execution time (WCET) for each block of code. The first version of our code is shown in [fig ? first CFG] along with the corresponding WCET. Through this quantitative analysis, we discovered that sending data over `Serial.println()` was infeasible to schedule. It took too much time to interpret numbers as actual characters. To make scheduling feasible, we switched from `Serial.println()` to `Serial.write()` to send BLE messages with a 64 byte maximum payload. This sizing allowed us to send five custom data packets per payload. When the IMU was not fully calibrated, we only sent the calibration data to the PC. Once everything was calibrated, we only sent the data. It took about 24 ms to send each payload. However, for every six data packets, one is dropped. We found this to be a reasonable error because the sampling rate was fast enough for our purposes. FIFO buffer concept Additionally, the Beans RGB LED is controlled by the BLE module instead of

the microcontroller, so it takes about 30 ms to set the LED color. But because we do not have a highly customizable scheduler with preemption, we drop two data packets every time we set the

LED. This meant that we lost force sensor feedback for that time. To solve this issue, we switched to the LED sequin, which only uses a digitalWrite.

A. Custom Data Packets

Each of our custom data packets contains 5 readings of 12 bits. The purpose of each bit can be seen in figure [?]. The valid bit is used to signify that the IMU is fully calibrated. Three bits are assigned to buttons and the force sensor depending on their on or off status. Each axis of acceleration has 12 bits encoded which adds up to 36 bits and allows for an acceleration range of -20.48 to 20.47 m/s^2 . Sixteen bits are assigned to each axis of the Gyroscope to allow for a range of -180 to 180 degrees with 2 decimal places. The last eight bits are to help with detecting packet loss.

VII. MODE OF OPERATION

reWRITE can be modeled okay this is not how the sentence should be phrased but I cant think of a different way to say it right now by the hierarchical state machine in figure [?]. When reWRITE is turned on, it enters the Calibration state. In this state, the IMU calibrates the accelerometer, gyroscope, and magnetometer, printing each level of calibration. As soon as all three sensors are calibrated, reWRITE preemptively transitions and enters the Button state. Here, it waits for the user to press the button before starting position reconstruction. Once the button is pressed, reWRITE enters the Data state. The nested state machine in Data is also a hierarchical state machine. In Data, the system constantly receives new data and plots the reconstructed position. It performs calculations and plots by utilizing synchronous composition of state machines in the Active state. One of the synchronous state machines executes the velocity calibration [reference to that section of report?] when the guard evaluates to true replace with what the guard actually is?. Simultaneously, the

other state machine plots the position when the marker presses the force sensor.

VIII. ACKNOWLEDGEMENTS

We would like to acknowledge the following individuals for their support and contribution in this project.

- Trung Tran, *National Instruments*
- Prof. Sanjit Seshia, *UC Berkeley*
- Matthew Weber, *UC Berkeley*
- Eric Kim, *UC Berkeley*
- Casey Rogers, *UC Berkeley 3D Modeling Club*

REFERENCES

- [1] Premerlani, W., Bizard, P.: Direction cosine matrix IMU: theory. <http://gentlenav.googlecode.com/files/DCMDraft2.pdf>

IX. APPENDIX 1: LIGHTBLUE BEAN CODE

```

#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/ImuMaths.h>

#define LED (0)
#define FORCE_SENSOR (1)
#define BUTTON (3)

Adafruit_BNO055 bno = Adafruit_BNO055();

uint8_t sys, gyroCal, accelCal, magCal = 0;
long time = 0;
boolean button = 0;
boolean sync = 0;
int do_loop = 0;
boolean debug = 0;
byte count = 0;
byte cur_count = 0;
char buff[60] = {0};
int reading_no = 0;
int ax = 0;
int ay = 0;
int az = 0;
int ex = 0;
int ey = 0;
int ez = 0;
boolean force = 0;

imu::Vector<3> accel;
imu::Vector<3> euler;

// Timer ISR to trigger reading
ISR(TIMER1_COMPA_vect){
    do_loop = 1;
    count++;
}

void setup(void){
    // Set to red for calibration
    Bean.setLed(255,0,0);

    sync = 0;
    do_loop = 0;
    reading_no = 0;
    count = 0;

    pinMode(LED, OUTPUT);
    pinMode(BUTTON, INPUT);
    pinMode(FORCE_SENSOR, INPUT);

    cli();          // disable global interrupts

```

```

TCCR1A = 0;      // set entire TCCR1A register to 0
TCCR1B = 0;      // same for TCCR1B

// set compare match register to desired timer count:
OCR1A = 117;     // 15ms period

// turn on CTC mode:
TCCR1B |= (1 << WGM12);

// Set CS10 and CS12 bits for 1024 prescaler:
TCCR1B |= (1 << CS10);
TCCR1B |= (1 << CS12);

// enable timer compare interrupt:
TIMSK1 |= (1 << OCIE1A);
sei();           // enable global interrupts

// Start serial communication
Serial.begin(57600);
Serial.println("Orientation_Sensor_Raw_Data_Test"); Serial.println("");

/* Initialise the sensor */
if(!bno.begin())
{
  /* There was a problem detecting the BNO055 ... check your connections */
  Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C ADDR!");
  while(1);
}

delay(1000);

bno.setExtCrystalUse(true);
}

void loop(void){
  if (do_loop){
    // Get data from IMU, force sensor and button
    cur_count = count;
    bno.getCalibration(&sys, &gyroCal, &accelCal, &magCal);
    accel = bno.getVector(Adafruit_BNO055::VECTOR_LINEARACCEL);
    euler = bno.getVector(Adafruit_BNO055::VECTOR_EULER);
    force = digitalRead(FORCE_SENSOR);
    button = digitalRead(BUTTON) | (force << 1);

    // If just got synced
    if (sync == 0 && gyroCal == 3 && accelCal == 3 && magCal == 3){
      buff[0] = char(1 << 7);
      buff[1] = char(sys << 6 | accelCal << 4 | gyroCal << 2 | magCal);
      buff[11] = char(cur_count);

      Serial.write((const unsigned char*)buff, 12);
      Serial.flush();
      Bean.setLed(246, 255, 0);
      reading_no = 0;
    }
  }
}

```



```

// Wait for confirmation from PC
while (Serial.read() != '1');

// Wait for button press
Bean.setLed(0,0,255);
while (digitalRead(BUTTON) != 1);
while (digitalRead(BUTTON) != 0);
Bean.setLed(0,255,0);

sync = 1;
}
else {
// If not synced yet
if (sync == 0){
// Set calibration packet in buffer
buff[12*reading_no] = char(1 << 7);
buff[12*reading_no+1] = char(sys << 6 | accelCal << 4 | gyroCal << 2 |
magCal);
buff[12*reading_no+11] = char(cur_count);

// Turn on LED based on calibration state
if (gyroCal == 3 && magCal == 3){
if (accelCal == 0) Bean.setLed(0,0,0);
if (accelCal == 1) Bean.setLed(73,243,243);
if (accelCal == 2) Bean.setLed(242,189,73);
}
}
else {
// Format data for packeting
ax = int(accel.x()*100) & 0xffff;
ay = int(accel.y()*100) & 0xffff;
az = int(accel.z()*100) & 0xffff;

ex = int(euler.x()*100);
ey = int(euler.y()*100);
ez = int(euler.z()*100);

// Set data packet in buffer
buff[12*reading_no] = char(1 << 7 | button << 4 | ax >> 8);
buff[12*reading_no+1] = char(ax & 0xff);
buff[12*reading_no+2] = char(ay >> 4);
buff[12*reading_no+3] = char((ay & 0xf) << 4 | (az >> 8) & 0xf);
buff[12*reading_no+4] = char(az & 0xff);
buff[12*reading_no+5] = char(ex >> 8);
buff[12*reading_no+6] = char(ex & 0xff);
buff[12*reading_no+7] = char(ey >> 8);
buff[12*reading_no+8] = char(ey & 0xff);
buff[12*reading_no+9] = char(ez >> 8);
buff[12*reading_no+10] = char(ez & 0xff);
buff[12*reading_no+11] = char(cur_count);

// User feedback for force sensor on LED Sequin
if(force) digitalWrite(LED, HIGH);
else digitalWrite(LED, LOW);
}
}

```

```

    }

    reading_no++;
    // If buffer filled with 5 readings , send to PC
    if (reading_no > 4) {
        Serial.write((const unsigned char*)buff , 60);
        reading_no = 0;
    }
}
// Reset ISR signal
do_loop = 0;
}
}

```

X. APPENDIX 2: PYTHON CODE

```

import numpy as np
import matplotlib.pyplot as plt
import math
import sys
import serial
import glob
import time
import re
import os

# Data arrays
ax = np.array([])
ay = np.array([])
az = np.array([])
vx = np.array([])
vy = np.array([])
vz = np.array([])
x = np.array([])
y = np.array([])
z = np.array([])
t = np.array([])
tipx = np.array([])
tipy = np.array([])
forceSensor = np.array([])

# Constant parameters
len_pen = 0.1
accel_thres = 0.15
accel_time_lim = 0.1
scale_factor = [10/9.5, 10/7.5]

# Number of readings per processing loop
CHUNKS = 20

def serial_ports():
    """Lists serial ports
    Raises:
        EnvironmentError:
    """

```

On unsupported or unknown platforms

Returns:

A list of available serial ports

"""

```

if sys.platform.startswith('win'):
    ports = ['COM' + str(i + 1) for i in range(256)]
elif sys.platform.startswith('linux') or sys.platform.startswith('cygwin'):
    # this is to exclude your current terminal "/dev/tty"
    ports = glob.glob('/dev/tty[A-Za-z]*')
elif sys.platform.startswith('darwin'):
    ports = glob.glob('/dev/tty.*')
else:
    raise EnvironmentError('Unsupported platform')
result = []
for port in ports:
    try:
        s = serial.Serial(port)
        s.close()
        result.append(port)
    except (OSError, serial.SerialException):
        pass
return result

```

```

def run():
    global ax, ay, az, vx, vy, vz, x, y, z, t, v_anchor, tipx, tipy, forceSensor
    print("reWRITE_Position_Reconstruction")

    # Connect to serial port
    ports = serial_ports()
    if ports:
        print("Available serial ports:")
        for (i,p) in enumerate(ports):
            print("%d) %s"%(i+1,p))
    else:
        print("No ports available. Check serial connection and try again.")
        print("Exiting ...")
        return
    portNo = input("Select the port to use: ")
    ser = serial.Serial(ports[int(portNo) - 1])
    ser.baudrate=57600
    ser.timeout=10

    # Reset variables
    cur_idx = 0
    mean_x = 0
    mean_y = 0
    mean_z = 0
    last_ax = 0
    last_ay = 0
    last_az = 0
    base_ex = 0
    base_ey = 0
    base_ez = 0
    base_time = 0

```

```

# Calibration constants
last_zero = [1,1,1]
first_nonzero = [0,0,0]
last_forceOn = -1
lpf_alpha = 0.9
hpf_alpha = 0.99
cal_pow = 2
failcount = 0

ser.flush()

# Calibration stage
calibrated = 0
if (sys.argv[2] == 'c'):
    calibrated = 1

while(calibrated != 1):
    try:
        line = (ser.read(12))
        c = line[1]
        # Print calibration status
        print(str((c & 0xc0) >> 6) + str((c & 0x30) >> 4) + str((c & 0x0c) >> 2) +
              str(c & 0x03))
        if ((c & 0x3f) == 0x3f): calibrated = 1
    except:
        pass

print("Recording_in_3...")
time.sleep(1)
print("2...")
time.sleep(1)
print("1...")
time.sleep(1)
print("Start")
ser.write('1'.encode())
ser.flushInput()
print(ser.inWaiting())

# Start plot
plt.ion()
fig = plt.figure()
figA = fig.add_subplot(111)
figA.set_xlabel('x')
figA.set_ylabel('y')

# Throw away first 20 data points
for i in range(20):
    line = ser.read(12)

# Main loop
while(True):
    print(cur_idx)
    ex = np.zeros(CHUNKS)
    ey = np.zeros(CHUNKS)
    ez = np.zeros(CHUNKS)

```

```

tax = np.zeros(CHUNKS)
tay = np.zeros(CHUNKS)
taz = np.zeros(CHUNKS)
count = 0

# Populate 20 data points
while count < CHUNKS:
    if (failcount > 20):
        print("Failed more than 20 times")
        exit()
    try:
        line = ser.read(12)

        # Populate force sensor and button data
        force = (line[0] >> 5) & 0x1;
        button = (line[0] >> 4) & 0x1;

        # Populate acceleration data
        ax = (line[0] & 0xf) << 8 | line[1]
        if (ax & 0x800): ax = -1*int((ax ^ 0xffff) + 1)
        ax = ax/100

        ay = line[2] << 4 | (line[3] >> 8)
        if (ay & 0x800): ay = -1*int((ay ^ 0xffff) + 1)
        ay = ay/100

        az = (line[3] & 0xf) << 8 | line[4]
        if (az & 0x800): az = -1*int((az ^ 0xffff) + 1)
        az = az/100

        # Populate Euler angle data
        ex = line[5] << 8 | line[6]
        ex = int(ex)/100

        ey = line[7] << 8 | line[8]
        if (ey & 0x8000): ey = -1*int((ey ^ 0xffff) + 1)
        ey = int(ey)/100

        ez = line[9] << 8 | line[10]
        if (ez & 0x8000): ez = -1*int((ez ^ 0xffff) + 1)
        ez = int(ez)/100

        # Populate timestamp data
        timestemp = int(line[11])
        if t != [] and timestemp+base_time - t[-1] < 0:
            base_time = base_time + 256
        timestemp = timestemp + base_time

        # Acceleration high-pass, low-pass and thresholding
        temp = float(ax)
        if (cur_idx == 0 and count == 0):
            mean_x = temp
        mean_x = mean_x * hpf_alpha + temp * (1-hpf_alpha)
        temp = temp - mean_x
        last_ax = last_ax*lpf_alpha + temp*(1-lpf_alpha)
        tax[count] = last_ax if abs(last_ax) > accel_thres/3 else 0

```

```

temp = float(ayt)
if (cur_idx == 0 and count == 0):
    mean_y = temp
mean_y = mean_y * hpf_alpha + temp * (1-hpf_alpha)
temp = temp - mean_y
last_ay = last_ay*lpf_alpha + temp*(1-lpf_alpha)
tay[count] = last_ay if abs(last_ay) > accel_thres/3 else 0

temp = float(azt)
if (cur_idx == 0 and count == 0):
    mean_z = temp
mean_z = mean_z * hpf_alpha + temp * (1-hpf_alpha)
temp = temp - mean_z
last_az = last_az*lpf_alpha + temp*(1-lpf_alpha)
taz[count] = last_az if abs(last_az) > accel_thres/3 else 0

# Convert Euler angles to deltas in radians
if (cur_idx == 0 and count == 0):
    ex[count] = 0
    ey[count] = 0
    ez[count] = 0
    base_ez = ezt
    base_ey = eyt
    base_ex = ext
else:
    ez[count] = -(ezt - base_ez)/360*2*math.pi
    ey[count] = (eyt - base_ey)/360*2*math.pi
    ex[count] = (ext - base_ex)/360*2*math.pi

# Save timestamp and force sensor data
t = np.append(t, timetemp)
forceSensor = np.append(forceSensor, force)

# If button is pressed, clear figure
if(button):
    figA.cla()

count = count + 1

except:
    failcount = failcount + 1
    pass

# Convert IMU frame of reference to real coordinates
sex = np.sin(ex)
sey = np.sin(ey)
sez = np.sin(ez)
cex = np.cos(ex)
cey = np.cos(ey)
cez = np.cos(ez)

ax = np.append(ax, cey*cez*tax + (sex*sey*cez - cex*sez)*tay + (cex*sey*cez +
sex*sez)*taz)
ay = np.append(ay, cey*sez*tax + (sex*sey*sez + cex*cez)*tay + (cex*sey*sez -
sex*cez)*taz)

```



```

az = np.append(az, -sey*tax + sex*cey*tay + cex*cey*taz)

# Process absolute acceleration data
for i in range(cur_idx, cur_idx+CHUNKS):
    if (i == 0):
        vx = np.array([0])
        vy = np.array([0])
        vz = np.array([0])
        x = np.array([0])
        y = np.array([0])
        z = np.array([0])
        v_anchor = np.array([1])
        continue

    timestep = (t[i] - t[i-1])*15/1000

    # Update velocity
    vx = np.append(vx, vx[i-1]+ax[i-1]*timestep)
    vy = np.append(vy, vy[i-1]+ay[i-1]*timestep)
    vz = np.append(vz, vz[i-1]+az[i-1]*timestep)

    # Update position
    x = np.append(x, x[i-1]+vx[i-1]*timestep)
    y = np.append(y, y[i-1]+vy[i-1]*timestep)
    z = np.append(z, z[i-1]+vz[i-1]*timestep)

    # Calibrate x velocity if more than accel_time_lim with no x accel
    if (abs(ax[i]) <= accel_thres):
        if first_nonzero[0] != -1 and last_zero[0] == -1:
            last_zero[0] = i
        else:
            last_zero[0] = -1
            if first_nonzero[0] == -1:
                first_nonzero[0] = i

    if (last_zero[0] != -1 and (t[i] - t[last_zero[0]]) > accel_time_lim*1000/15):
        :
        for j in range(first_nonzero[0], last_zero[0]+1):
            timestep = (t[j] - t[j-1])*15/1000
            vx[j] = vx[j] - ((j-first_nonzero[0])/(last_zero[0]-first_nonzero[0]))**
                cal_pow*vx[last_zero[0]]
            x[j] = x[j-1] + vx[j-1]*timestep
        for j in range(last_zero[0], i+1):
            vx[j] = 0
            x[j] = x[j-1]
        first_nonzero[0] = i-1
        last_zero[0] = -1

    # Calibrate y velocity if more than accel_time_lim with no y accel
    if (abs(ay[i]) <= accel_thres):
        if first_nonzero[1] != -1 and last_zero[1] == -1:
            last_zero[1] = i
        else:
            last_zero[1] = -1
            if first_nonzero[1] == -1:

```

```

        first_nonzero[1] = i

    if (last_zero[1] != -1 and (t[i] - t[last_zero[1]]) > accel_time_lim*1000/15):
        :
        for j in range(first_nonzero[1], last_zero[1]+1):
            timestep = (t[j] - t[j-1])*15/1000
            vy[j] = vy[j] - ((j-first_nonzero[1])/(last_zero[1]-first_nonzero[1]))**
                cal_pow*vy[last_zero[1]]
            y[j] = y[j-1] + vy[j-1]*timestep
        for j in range(last_zero[1], i+1):
            vy[j] = 0
            y[j] = y[j-1]
        first_nonzero[1] = i-1
        last_zero[1] = -1

    # Calibrate z velocity if more than accel_time_lim with no z accel
    if (abs(az[i]) <= accel_thres):
        if first_nonzero[2] != -1 and last_zero[2] == -1:
            last_zero[2] = i
        else:
            last_zero[2] = -1
            if first_nonzero[2] == -1:
                first_nonzero[2] = i

    if (last_zero[2] != -1 and (t[i] - t[last_zero[2]]) > accel_time_lim*1000/15):
        :
        for j in range(first_nonzero[2], last_zero[2]+1):
            timestep = (t[j] - t[j-1])*15/1000
            vz[j] = vz[j] - ((j-first_nonzero[2])/(last_zero[2]-first_nonzero[2]))**
                cal_pow*vz[last_zero[2]]
            z[j] = z[j-1] + vz[j-1]*timestep
        for j in range(last_zero[2], i+1):
            vz[j] = 0
            z[j] = z[j-1]
        first_nonzero[2] = i-1
        last_zero[2] = -1

    # Compute tip position from IMU position
    tipx = np.append(tipx, x[cur_idx:]*scale_factor[0] - len_pen*sey)
    tipy = np.append(tipy, y[cur_idx:]*scale_factor[1] - len_pen*sex)

    # Plot if we go from force sensor on to off
    if (last_forceOn != -1):
        first_zero_f = None
        for idx in range(cur_idx, cur_idx+CHUNKS):
            if forceSensor[idx] == 0:
                first_zero_f = idx
                break
        if (first_zero_f):
            figA.plot(tipx[last_forceOn:first_zero_f-5], tipy[last_forceOn:first_zero_f-5], 'blue')
            plt.axis('equal')
            last_forceOn = -1
            fig.canvas.draw()

```

```

if (last_forceOn == -1):
    first_zero_f = None
    for idx in range(cur_idx, cur_idx+CHUNKS):
        if forceSensor[idx] == 1:
            first_zero_f = idx
            break
    if (first_zero_f):
        last_forceOn = first_zero_f

cur_idx = cur_idx + CHUNKS

# Time limit of 10000 samples
if (cur_idx >= 10000):
    break

# Keep plot on
plt.show()

run()

```