Arduino/Genuino 101 CurieIMU Orientation Visualiser

This tutorial demonstrates how to make use the Genuino 101's on-board 6-axis accelerometer/gyro to read the X, Y, and Z values of both the accelerometer and the gyroscope. While the accelerometer is able to determine the orientation of the board, the gyroscope measures the angular velocity of the board. Together, the accelerometer and the gyroscope form an Inertial Monitoring Unit (IMU) which can be used to precisely identify the orientation of the board. Madgwick's filter algorithm is used in this example to calculate four quarternians from the 6 axes' values. The quarternions are then used to calculate Euler angles Pitch, Yaw, and Roll, which are received by Processing and used to control the rotation of an object around the X, Y and Z axes.

Hardware Required

- Arduino/Genuino 101

The CurieImu library uses the IMU (accelerometer + gyroscope) built into the Genuino/Arduino 101.

Instructions

1. Set up the Arduino software as described in Getting Started with Arduino 101.
2. Connect the Arduino 101 to your computer.
3. Launch the Arduino software and select Arduino 101 from the Tools > Board menu.
4. Install the *Madgwick* library from Arduino IDE's library manager. To do this, open the Arduino IDE, go to "Sketch -> Include Library -> Manage Libraries". There you can search 'Madgwick' and install the library directly from there. Please see the libraries installation guide for a more detailed explanation on installing and importing libraries.
5. Download and Launch the Processing software and create a file with the Processing code shown below.
6. Change the Serial port to the one that your Arduino is using (see "Processing Sketch" section).
7. Upload the CurieImu example to your Arduino 101, making sure that the board is flat and stationery so it can perform the calibration accurately.
8. After a few seconds, run the Processing sketch, adjust the orientation of your board, and watch as the Processing sketch gives a visualisation of your board.

The Circuit

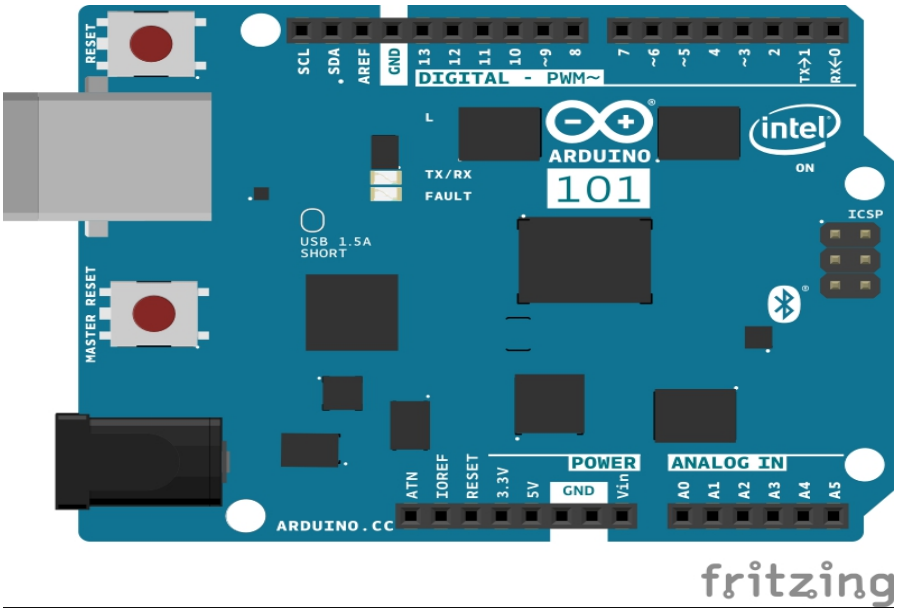

image developed using Fritzing.

How it works

The Madgwick filter algorithm is open-source and is well documented in Madgwick's information and reports. The Madgwick filter algorithm was developed by Sebastian Madgwick during his PhD in 2010, and is designed to be computationally inexpensive and efficient even at low sampling rates. The algorithm takes calibrated values from a gyroscope and accelerometer, and uses them to return four quaternions, which are 4-dimensional numbers which contain x, y, and z values to represent the axis around which rotation occurs, as well as a ω value which represents the value of rotation which occurs around the same axis. These quaternions can be used to calculate the Euler angles pitch, yaw, and roll; three angles used to describe the orientation of a rigid body in terms of x,y, and z as presented by Leonhard Euler in the 1700s. The equations (7) (8) (9) in Madgwick's Report are used to calculate the values for pitch, roll, and yaw, and their functions are included within the library.

We call the Arduino/Genuino 101's on-board IMU to create a 3D representation of the board in Processing, which will move as the board does. This is achieved with the values for Euler angles pitch, roll and yaw obtained by the Madgwick filter algorithm. These values can then be sent via Serial to Processing and used as angle arguments for Processing's rotateX(), rotateY(), and rotateZ() functions. These functions take on argument, an angle in radians, and rotate the object they correspond to around their respective axis by the angle value.

Arduino Sketch

The sketch uses functions inside the CurieImu library to get the data from the accelerometer/gyro.

In order to see a 3D representation in Processing, the Arduino sketch must incorporate two main functionalities; using the IMU data and algorithm to calculate yaw, pitch and roll values, and enabling serial communication in a handshake fashion in order to send those values to Processing.

A calibration process is implemented during the setup, which can again be used or not by setting the variable `calibrateOffsets` to 1 or 0 respectively. The calibration process is then excecuted `if calibrateOffsets > 0` and reads the initial values of the board and executes internal calibration to generate an offset compensation value for each axis. Note that the user must keep the board perfectly still and resting horizontally as shown in Section 5.2 of the BMI160 Data Sheet.

First, we must create a Madgwick object to access the functions from the Madgwick class in the library. Here, we call it *filter*:

```
Madgwick filter;
```

The previous version of the Madgwick algorithm had the sampling frequency set to 512Hz implying a sample rate of 1 every 0.00195 seconds; however, in actuality the sample rate should be calculated during each iteration and passed to the algorithm. In addition, the sampling of the Accelerometer and Gyro's should be adjusted to account for the expected sampling rate. If you do not, you will start seeing a very large yaw drift in your data. The second adjustment is that the filter gain specified by beta has now been exposed so you can adjust this from the Arduino sketch as opposed to modifying the filter library. The third adjustment is that quaternions have been exposed at the sketch level. The last change is the current term, factor, has been replaced with the Gyro Sensitivity, which is taken from the BMI-160 datasheet. The last adjustment is a algorithm coding correction. For an explanation see the [Madgwick IMU/AHRS and Fast Inverse Square Root] thread at DIY Drones.

In the header of the sketch the gyro sensitivity is set:

[Get Code]

```
/*
 * Full Scale Range   | LSB Sensitivity
 * -------------------+----------------
 * +/- 125  degrees/s | 262.4 LSB/deg/s
 * +/- 250  degrees/s | 131.2 LSB/deg/s
 * +/- 500  degrees/s | 65.5  LSB/deg/s
 * +/- 1000 degrees/s | 32.8  LSB/deg/s
 * +/- 2000 degrees/s | 16.4  LSB/deg/s : use 20.5
 */
#define gyro_sensitivity 20.5f
```

[Get Code]

I found that this may need to be adjusted slightly to reduce yaw drift.

We can also adjust the Gyro and Accelerometer ranges using the following commands after the CurieImu.initialization:

```
  CurieImu.setFullScaleAccelRange(BMI160_ACCEL_RANGE_2G);
//sets the accelerometer range to +/-2g
  CurieImu.setAccelRate(BMI160_ACCEL_RATE_100HZ);       //sets
the accelerometer sample frequency to 100Hz

  CurieImu.setFullScaleGyroRange(BMI160_GYRO_RANGE_2000);
//sets the gryo range to +/-2000 deg/sec/sec
  CurieImu.setGyroRate(BMI160_GYRO_RATE_100HZ);         //sets
the gryo range to +/-2g
```

[Get Code]

The filter gain is set by adjusting the beta term,

```
  filter.beta = 0.07;  //use 0.05 for gyro range of 500
```

[Get Code]

[Get Code]

We can then 'get' accelerometer and gyroscope data using the following functions from CurieIMU library:

```
ax = CurieImu.getAccelerationX();
ay = CurieImu.getAccelerationY();
az = CurieImu.getAccelerationZ();
gx = CurieImu.getRotationX();
gy = CurieImu.getRotationY();
gz = CurieImu.getRotationZ();
```

[Get Code]

The last change is that currently the gyro values are passed to the filter in deg/s/s where the algorithm requires that they be passed in radians/s/s.

```
  filter.updateIMU(val_cal[3]*deg2rad, val_cal[4]*deg2rad,
val_cal[5]*deg2rad, val_cal[0], val_cal[1], val_cal[2]);
```

[Get Code]

Where the val_cal terms are the gyro and accelerometers calibrated values:
val_cal[0] = CurieImu.getAccelerationX();
val_cal[1] = CurieImu.getAccelerationY();
val_cal[2] = CurieImu.getAccelerationZ();
val_cal[3] = CurieImu.getRotationX()/gyro_sensitivity;
val_cal[4] = CurieImu.getRotationY()/gyro_sensitivity;
val_cal[5] = CurieImu.getRotationZ()/gyro_sensitivity;

These are used later to pass floats to the Processing sketch included in the library.

As a bonus the library i includes Madgwick's implementation of Robert Mayhony's 'DCM filter' in quaternion form as referenced in his paper and website. For the Mahony filter everything remains the same except for:

1) *Creating the Mahony AHRS algorithm:*
    a. *Include file becomes*
        #include <MahonyAHRS.h> instead of #include <MadgwickAHRS.h>
    b. *Declaring the filter object becomes*
        Mahony filter instead of Madgwick filter
2) *Filter gains become*
    a. *filter.twoKp = (2.0f * 1.25f);*
    b. *filter.twoKi = (2.0f * 0.1f);*

    These are adjustable gains that control the AHRS performance including yaw drift. As a guide the following guidance can be used:

    In a post on DIY Drones (http://diydrones.com/forum/topics/freeimu-firmware-on-arduimu?commentId=705844%3AComment%3A1010680) by Seb Madgwick in 2011 he suggested the following for tuning Kp and Ki: Leave Ki as 0 and start with a Kp value of 5. You will want to reduce your Kp value from this by 10 or even 100 times when tuning. The lowest value of Kp you can use is dependent on: gyroscope bias calibration errors, and gyroscope sensitivity calibration errors and expected angular dynamics of application (coupled characteristics).

The quaternions can be accessed by simply referring to the class element as follows:

```
q0 = filter.q[0];
q1 = filter.q[1];
q2 = filter.q[2];
q3 = filter.q[3];
```

We can then use the function getYaw, getRoll and getPitch from the Madgwick library.

```
yaw = filter.getYaw();
roll = filter.getRoll();
pitch = filter.getPitch();
```

[Get Code]

~~As seen in the code, the gyroscope values have been scaled down by a variable factor so that they fit into a range which works well with the algorithm. Without this scaling, the values which are inputted to the function are too high and the visualisation of the movement of the board becomes very sensitive to small changes of the Arduino's position, intepreting a slight change as an great change and causing the 'virtual' board to spin. The variable integer 'factor' can be adjusted to experiment with and improve the performance of the dynamic representation, and should also be increased with an increase in baud rate.~~

The Serial communication can then be handled with the following block. Firstly, this checks for an incoming value of "s" from the serial which is sent by processing at the end of each loop. This ensures that the Arduino sketch does not send the values more often than Processing can process them which would result an extremely laggy visualisation. If an "s" is received, the values are sent by serial, each seperated with a comma and ending in a new line so that that each message can be parsed easily in Procesing. The full code can be found at the bottom of the page.

```
if (Serial.available() > 0) {
    int val = Serial.read();
    if(val == 's')
    {
      Serial.print(yaw);
      Serial.print(",");
      Serial.print(pitch);
      Serial.print(",");
      Serial.println(roll);
    }
   }
```

[Get Code]

Note that the serial prints for gx, gy, gz, ax, az, ay are left in loop in comments for debugging and must be commented whilst communicating with Processing.

Processing Sketch

If you haven't already, the first thing to do is to download the latest version of Processing from processing.org. Processing is a language similar to Arduino which allows the user to draw dynamic imagery in the familiar `void setup()` and `void loop()` structure. For more information on using Processing, please visit their Getting Started guide.

The processing code receives incoming data from the serial port which is parsed and assigned to floats `yaw`, `pitch`, and `roll`, which are then used as arguments in `RotateX()`, `RotateY()`, and `RotateZ()` functions. In each iteration of loop(), Processing reads and parses the serial information, draws a 3D Arduino shape with the parsed values and then sends a "s" via serial to indicate to Arduino that it is ready for more information.

To enable Processing to read from the same port that Arduino is sending to, myPort needs to be changed to your serial port's name. In setup(), this is the Second parameter of Serial.

```
myPort = new Serial(this, Serial.list()[2], 9600);
```

[Get Code]

The correct port can be found by using the list() function from the Serial class. The number inside the square brackets refers to the number of the serial port, and will be 0, 1, 2, etc. *If in doubt, you can print a list of your available serial ports in a separate sketch to determine this number.*

The function serialEvent() is then used to receive and parse data.

```
void serialEvent()
{
  message = myPort.readStringUntil(13);
  if (message != null) {
    ypr = split(message, ",");
    yaw = float(ypr[0]);
    pitch = float(ypr[1]);
    roll = float(ypr[2]);
  }
}
```

[Get Code]

This reads from the serial port until ASCII character 13 (new line) and then uses the split() function to seperate the values using the comma character. Since we know that we sent from Arduino in the order yaw, pitch, roll, we can then convert each string to a float and assign them to the first three values in String array ypr[]. The strings are then converted into floats and stored in float variables. The full Arduino and Processing sketches can be seen below.

Code

Arduino Code

```
/*
  ============================================
  Example sketch for CurieImu library for Intel(R) Curie(TM)
devices.
  Copyright (c) 2015 Intel Corporation.  All rights reserved.

  Based on I2C device class (I2Cdev) demonstration Arduino
sketch for MPU6050
  class by Jeff Rowberg: https://github.com/jrowberg/i2cdevlib

  ============================================
  I2Cdev device library code is placed under the MIT license
  Copyright (c) 2011 Jeff Rowberg

  Permission is hereby granted, free of charge, to any person
obtaining a copy
  of this software and associated documentation files (the
"Software"), to deal
  in the Software without restriction, including without
limitation the rights
  to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell
  copies of the Software, and to permit persons to whom the
```

```
Software is
  furnished to do so, subject to the following conditions:

  The above copyright notice and this permission notice shall be
included in
  all copies or substantial portions of the Software.

  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR
  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE
  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES
OR OTHER
  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM,
  OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN
  THE SOFTWARE.
  ================================================

  Genuino 101 CurieIMU Orientation Visualiser
  Hardware Required:
  * Arduino/Genuino 101

  Modified Nov 2015
  by Helena Bisby <support@arduino.cc>
  This example code is in the public domain
  http://arduino.cc/en/Tutorial/Genuino101CurieIMUOrientationVis
ualiser
*/

#include "CurieImu.h"
#include "MadgwickAHRS.h"

Madgwick filter; // initialise Madgwick object
int ax, ay, az;
int gx, gy, gz;
float yaw;
float pitch;
float roll;
int factor = 800; // variable by which to divide gyroscope
values, used to control sensitivity
// note that an increased baud rate requires an increase in
value of factor
```

```
int calibrateOffsets = 1; // int to determine whether
calibration takes place or not


void setup() {
  // initialize Serial communication
  Serial.begin(9600);

  // initialize device
  CurieImu.initialize();

  // verify connection
  if (!CurieImu.testConnection()) {
    Serial.println("CurieImu connection failed");
  }


  if (calibrateOffsets == 1) {
    // use the code below to calibrate accel/gyro offset values
    Serial.println("Internal sensor offsets BEFORE
calibration...");
    Serial.print(CurieImu.getXAccelOffset());
Serial.print("\t");
    Serial.print(CurieImu.getYAccelOffset());
Serial.print("\t");
    Serial.print(CurieImu.getZAccelOffset());
Serial.print("\t");
    Serial.print(CurieImu.getXGyroOffset()); Serial.print("\t");
    Serial.print(CurieImu.getYGyroOffset()); Serial.print("\t");
    Serial.print(CurieImu.getZGyroOffset()); Serial.print("\t");
    Serial.println("");

    // To manually configure offset compensation values, use the
following methods instead of the autoCalibrate...() methods
below
    //    CurieImu.setXGyroOffset(220);
    //    CurieImu.setYGyroOffset(76);
    //    CurieImu.setZGyroOffset(-85);
    //    CurieImu.setXAccelOffset(-76);
    //    CurieImu.setYAccelOffset(--235);
    //    CurieImu.setZAccelOffset(168);

    //IMU device must be resting in a horizontal position for
the following calibration procedure to work correctly!

    Serial.print("Starting Gyroscope calibration...");
    CurieImu.autoCalibrateGyroOffset();
```

```
    Serial.println(" Done");
    Serial.print("Starting Acceleration calibration...");
    CurieImu.autoCalibrateXAccelOffset(0);
    CurieImu.autoCalibrateYAccelOffset(0);
    CurieImu.autoCalibrateZAccelOffset(1);
    Serial.println(" Done");

    Serial.println("Internal sensor offsets AFTER
calibration...");
    Serial.print(CurieImu.getXAccelOffset());
Serial.print("\t");
    Serial.print(CurieImu.getYAccelOffset());
Serial.print("\t");
    Serial.print(CurieImu.getZAccelOffset());
Serial.print("\t");
    Serial.print(CurieImu.getXGyroOffset()); Serial.print("\t");
    Serial.print(CurieImu.getYGyroOffset()); Serial.print("\t");
    Serial.print(CurieImu.getZGyroOffset()); Serial.print("\t");
    Serial.println("");
    Serial.println("Enabling Gyroscope/Acceleration offset
compensation");
    CurieImu.setGyroOffsetEnabled(true);
    CurieImu.setAccelOffsetEnabled(true);
  }
}

void loop() {
  // read raw accel/gyro measurements from device
  ax = CurieImu.getAccelerationX();
  ay = CurieImu.getAccelerationY();
  az = CurieImu.getAccelerationZ();
  gx = CurieImu.getRotationX();
  gy = CurieImu.getRotationY();
  gz = CurieImu.getRotationZ();

  // use function from MagdwickAHRS.h to return quaternions
  filter.updateIMU(gx / factor, gy / factor, gz / factor, ax,
ay, az);

  // functions to find yaw roll and pitch from quaternions
  yaw = filter.getYaw();
  roll = filter.getRoll();
  pitch = filter.getPitch();

  // print gyro and accel values for debugging only, comment out
when running Processing
  /*
```

```
  Serial.print(ax); Serial.print("\t");
  Serial.print(ay); Serial.print("\t");
  Serial.print(az); Serial.print("\t");
  Serial.print(gx); Serial.print("\t");
  Serial.print(gy); Serial.print("\t");
  Serial.print(gz); Serial.print("\t");
  Serial.println("");
  */

  if (Serial.available() > 0) {
    int val = Serial.read();
    if (val == 's') { // if incoming serial is "s"
      Serial.print(yaw);
      Serial.print(","); // print comma so values can be parsed
      Serial.print(pitch);
      Serial.print(","); // print comma so values can be parsed
      Serial.println(roll);
    }
  }
}
```

[Get Code]

Processing Code

```
import processing.serial.*;
Serial myPort;

int newLine = 13; // new line character in ASCII
float yaw;
float pitch;
float roll;
String message;
String [] ypr = new String [3];

void setup()
{
  size(600, 500, P3D);

  /*Set my serial port to same as Arduino, baud rate 9600*/
  myPort = new Serial(this, Serial.list()[2], 9600);
  textSize(16); // set text size
  textMode(SHAPE); // set text mode to shape
}

void draw()
{
  serialEvent();  // read and parse incoming serial message
  background(255); // set background to white
```

```
  translate(width/2, height/2); // set position to centre

  pushMatrix(); // begin object

  rotateX(pitch); // RotateX pitch value
  rotateY(-yaw); // yaw
  rotateZ(-roll); // roll

  drawArduino(); // function to draw rough Arduino shape

  popMatrix(); // end of object

  // Print values to console
  print(pitch);
  print("\t");
  print(roll);
  print("\t");
  print(-yaw);
  println("\t");

  myPort.write("s"); // write an "s" to receive more data from
Arduino
}

void serialEvent()
{
  message = myPort.readStringUntil(newLine); // read from port
until new line (ASCII code 13)
  if (message != null) {
    ypr = split(message, ","); // split message by commas and
store in String array
    yaw = float(ypr[0]); // convert to float yaw
    pitch = float(ypr[1]); // convert to float pitch
    roll = float(ypr[2]); // convert to float roll
  }
}
void drawArduino() {
  /* function contains shape(s) that are rotated with the IMU */
  stroke(0, 90, 90); // set outline colour to darker teal
  fill(0, 130, 130); // set fill colour to lighter teal
  box(300, 10, 200); // draw Arduino board base shape

  stroke(0); // set outline colour to black
  fill(80); // set fill colour to dark grey

  translate(60, -10, 90); // set position to edge of Arduino box
```

```
  box(170, 20, 10); // draw pin header as box

  translate(-20, 0, -180); // set position to other edge of
Arduino box
  box(210, 20, 10); // draw other pin header as box
}
```

[Get Code]

See Also

Further information on Genuino/Arduino 101
Getting Started
Arduino 101 CurieIMU Step Counter Tutorial
Arduino 101 CurieBLE Heart Rate Monitor Tutorial

Share