**SELF STABILIZING WHEEL CHAIR**

***Abstract:*** There exist mechanical devices for which it is important to retain a constant position, or a constant direction regardless of their space fitting. Classic applications of such systems include a camera stabilization in moving systems, a platform stabilization for mobile guidance systems, a platform for the patients in the wheel chair to make him feel not falling while climbing up or down the stairs or moving from ambulance to or from, or other special cases. For this and other cases, it is appropriate to use the now widespread and relatively accurate MEMS gyroscopes and accelerometers. Their consumption and size are negligible and can be used right from the smallest models or toys. This paper reports on a particular solution build on the sensorMPU6050 and its parameters that can scan 6DOF (3 × 3 accelerometer + gyroscope). In real conditions this task is not trivial, since the signals of these sensors are quite noisy and influenced by other variables such as reaction speed, inertia, and accuracy of data interpretation, sensor drift and others. Self-stabilizing platforms can be of great use especially in the automotive, aviation, marine, robotics, and aerospace as well as in everyday life.

***Methodology:*** We are using mainly a sensor to get the inclination values of the chair (yaw, pitch, and roll). We have done this using IMU (inertial measurement unit). Now we are normalizing this values and are fed to Servo motors (180 degree rotation) to control the deviated inclination and compensate that in two dimensions.

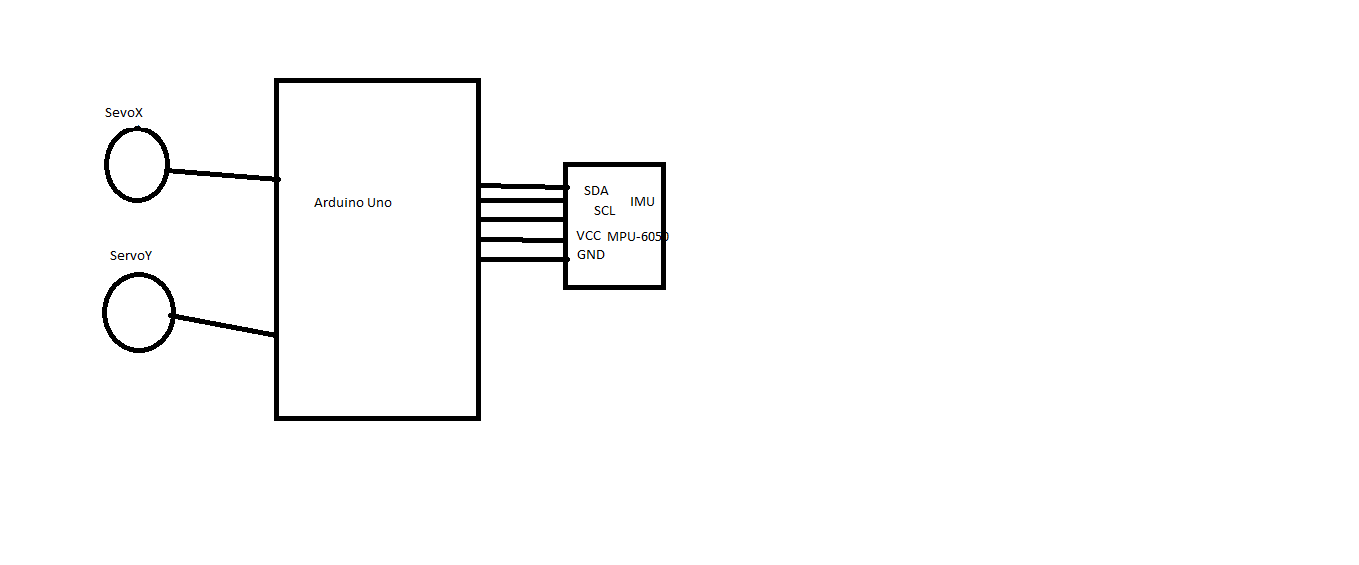
***Components Used:***

* Arduino Uno
* IMU sensor-MPU-6050
* 2 Servo motors
* Bread Board and Batteries

***Prototype of 2 Servo Motor:***



***Block Diagram:***



***Program (code):***

//Servo

#include <Servo.h>

Servo myservoY; // Roll

Servo myservoX; // Pitch

// I2Cdev and MPU6050 must be installed as libraries, or else the .cpp/.h files

// for both classes must be in the include path of your project

#include "I2Cdev.h"

#include "MPU6050\_6Axis\_MotionApps20.h"

//#include "MPU6050.h" // not necessary if using MotionApps include file

// Arduino Wire library is required if I2Cdev I2CDEV\_ARDUINO\_WIRE implementation

// is used in I2Cdev.h

#if I2CDEV\_IMPLEMENTATION == I2CDEV\_ARDUINO\_WIRE

    #include "Wire.h"

#endif

// class default I2C address is 0x68

// specific I2C addresses may be passed as a parameter here

// AD0 low = 0x68 (default for SparkFun breakout and InvenSense evaluation board)

// AD0 high = 0x69

MPU6050 mpu;

//MPU6050 mpu(0x69); // <-- use for AD0 high

/\*uncomment "OUTPUT\_READABLE\_YAWPITCHROLL" if you want to see the yaw pitch/roll angles (in degrees) calculated from the quaternions coming from the FIFO. Note this also requires gravity vector calculations.Also note that yaw/pitch/roll angles suffer from gimbal lock (for more info, see: http://en.wikipedia.org/wiki/Gimbal\_lock)

#define OUTPUT\_READABLE\_YAWPITCHROLL \*/

#define LED\_PIN 13 // (Arduino is 13, Teensy is 11, Teensy++ is 6)

bool blinkState = false;

// MPU control/status vars

bool dmpReady = false; // set true if DMP init was successful

uint8\_t mpuIntStatus;   // holds actual interrupt status byte from MPU

uint8\_t devStatus;      // return status after each device operation (0 = success, !0 = error)

uint16\_t packetSize;    // expected DMP packet size (default is 42 bytes)

uint16\_t fifoCount;     // count of all bytes currently in FIFO

uint8\_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars

Quaternion q;           // [w, x, y, z] quaternion container

VectorInt16 aa;         // [x, y, z] accel sensor measurements

VectorInt16 aaReal;     // [x, y, z] gravity-free accel sensor measurements

VectorInt16 aaWorld;    // [x, y, z] world-frame accel sensor measurements

VectorFloat gravity;    // [x, y, z] gravity vector

float euler[3]; // [psi, theta, phi] Euler angle container

float ypr[3]; // [yaw, pitch, roll] yaw/pitch/roll container and gravity vector

// packet structure for InvenSense teapot demo

uint8\_t teapotPacket[14] = { '$', 0x02, 0,0, 0,0, 0,0, 0,0, 0x00, 0x00, '\r', '\n' };

/\*INTERRUPT DETECTION ROUTINE   \*/

volatile bool mpuInterrupt = false; // indicates whether MPU interrupt pin has gone high

void dmpDataReady() {

    mpuInterrupt = true;

}

/\* INITIAL \*/

void **setup**() {

    // join I2C bus (I2Cdev library doesn't do this automatically)

    #if I2CDEV\_IMPLEMENTATION == I2CDEV\_ARDUINO\_WIRE

        Wire.begin();

        TWBR = 24; // 400kHz I2C clock (200kHz if CPU is 8MHz)

    #elif I2CDEV\_IMPLEMENTATION == I2CDEV\_BUILTIN\_FASTWIRE

        Fastwire::**setup**(400, true);

    #endif

//Attach servo

  myservoY.attach(9); // Attach Y servo to pin 9

  myservoX.attach(10);// Attach X servo to pin 10

    // initialize serial communication

    // (115200 chosen because it is required for Teapot Demo output, but it's

    // really up to you depending on your project)

**Serial**.begin(115200);

    while (!**Serial**); // wait for Leonardo enumeration, others continue immediately

    // NOTE: 8MHz or slower host processors, like the Teensy @ 3.3v or Ardunio

    // Pro Mini running at 3.3v, cannot handle this baud rate reliably due to

    // the baud timing being too misaligned with processor ticks. You must use

    // 38400 or slower in these cases, or use some kind of external separate

    // crystal solution for the UART timer.

    // initialize device

**Serial**.println(F("Initializing I2C devices..."));

    mpu.initialize();

    // verify connection

**Serial**.println(F("Testing device connections..."));

**Serial**.println(mpu.testConnection() ? F("MPU6050 connection successful") : F("MPU6050 connection failed"));

    // load and configure the DMP

**Serial**.println(F("Initializing DMP..."));

    devStatus = mpu.dmpInitialize();

    // supply your own gyro offsets here, scaled for min sensitivity

    mpu.setXGyroOffset(220);

    mpu.setYGyroOffset(76);

    mpu.setZGyroOffset(-85);

    mpu.setZAccelOffset(1788); // 1688 factory default for my test chip

    // make sure it worked (returns 0 if so)

    if (devStatus == 0) {

        // turn on the DMP, now that it's ready

**Serial**.println(F("Enabling DMP..."));

        mpu.setDMPEnabled(true);

        // enable Arduino interrupt detection

**Serial**.println(F("Enabling interrupt detection (Arduino external interrupt 0)..."));

        attachInterrupt(0, dmpDataReady, RISING);

        mpuIntStatus = mpu.getIntStatus();

        // set our DMP Ready flag so the main loop() function knows it's okay to use it

**Serial**.println(F("DMP ready! Waiting for first interrupt..."));

        dmpReady = true;

        // get expected DMP packet size for later comparison

        packetSize = mpu.dmpGetFIFOPacketSize();

    } else {

        // ERROR!

        // 1 = initial memory load failed

        // 2 = DMP configuration updates failed

        // (if it's going to break, usually the code will be 1)

**Serial**.print(F("DMP Initialization failed (code "));

**Serial**.print(devStatus);

**Serial**.println(F(")"));

    }

    // configure LED for output

    pinMode(LED\_PIN, OUTPUT);

}

/\* MAIN PROGRAM LOOP \*/

void **loop**() {

    // if programming failed, don't try to do anything

    if (!dmpReady) return;

    // wait for MPU interrupt or extra packet(s) available

    // reset interrupt flag and get INT\_STATUS byte

    mpuInterrupt = false;

    mpuIntStatus = mpu.getIntStatus();

    // get current FIFO count

    fifoCount = mpu.getFIFOCount();

    // check for overflow (this should never happen unless our code is too inefficient)

    if ((mpuIntStatus & 0x10) || fifoCount == 1024) {

        // reset so we can continue cleanly

        mpu.resetFIFO();

**Serial**.println(F("FIFO overflow!"));

    // otherwise, check for DMP data ready interrupt (this should happen frequently)

    } else if (mpuIntStatus & 0x02) {

        // wait for correct available data length, should be a VERY short wait

        while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

        // read a packet from FIFO

        mpu.getFIFOBytes(fifoBuffer, packetSize);

        // track FIFO count here in case there is > 1 packet available

        // (this lets us immediately read more without waiting for an interrupt)

        fifoCount -= packetSize;

        #ifdef OUTPUT\_READABLE\_YAWPITCHROLL

            // display Euler angles in degrees

            mpu.dmpGetQuaternion(&q, fifoBuffer);

            mpu.dmpGetGravity(&gravity, &q);

            mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);

**Serial**.print("ypr\t");

**Serial**.print(ypr[0] \* 180/M\_PI);

**Serial**.print("\t");

**Serial**.print(ypr[1] \* 180/M\_PI);

            myservoY.write(int(ypr[1] \* -180/M\_PI)+90); // Rotation around Y

**Serial**.print("\t");

**Serial**.println(ypr[2] \* 180/M\_PI);

            myservoX.write(int(ypr[2] \* 180/M\_PI)+90); // Rotation around X

        #endif

        // blink LED to indicate activity

        blinkState = !blinkState;

        digitalWrite(LED\_PIN, blinkState);

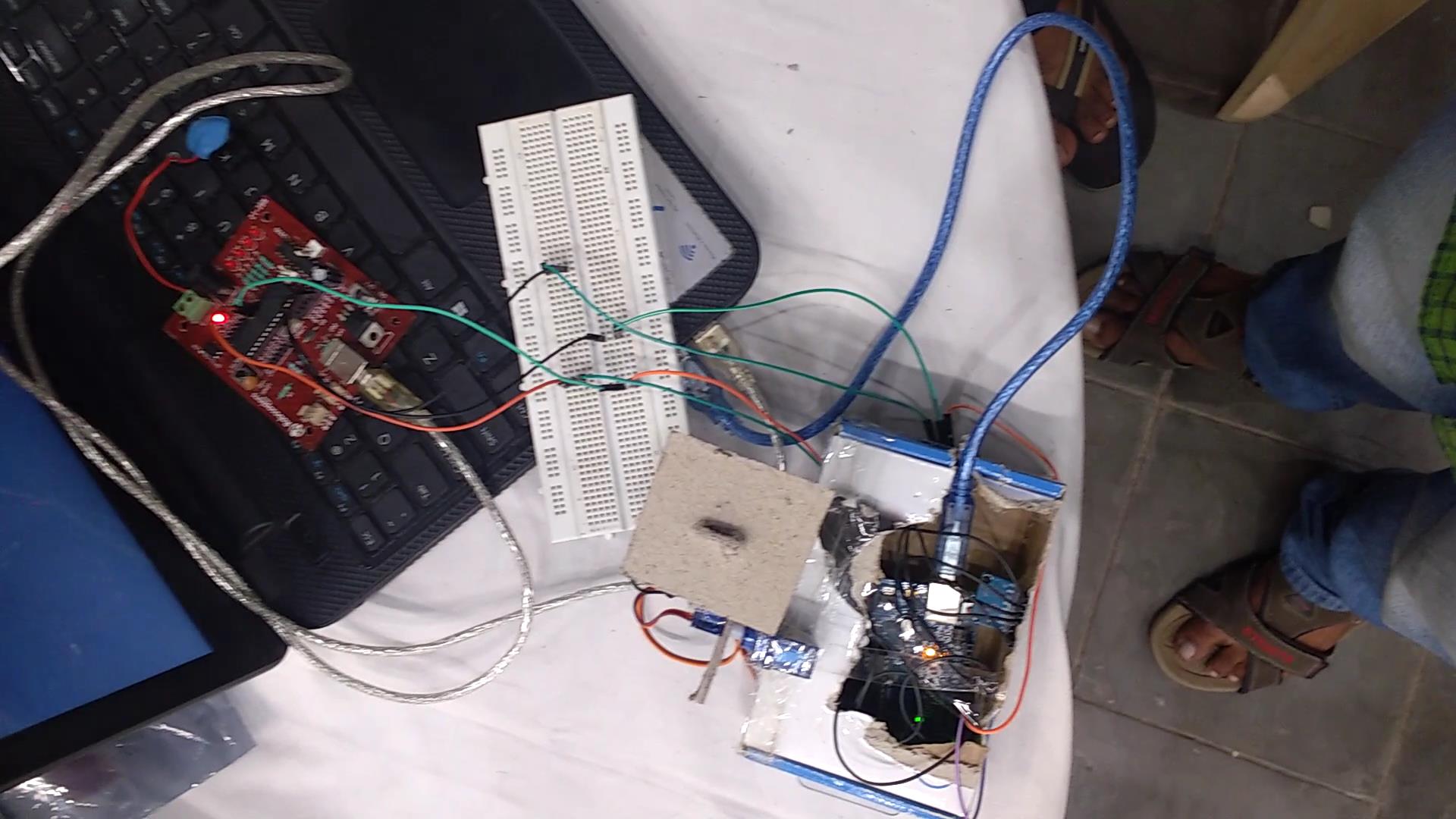
    }

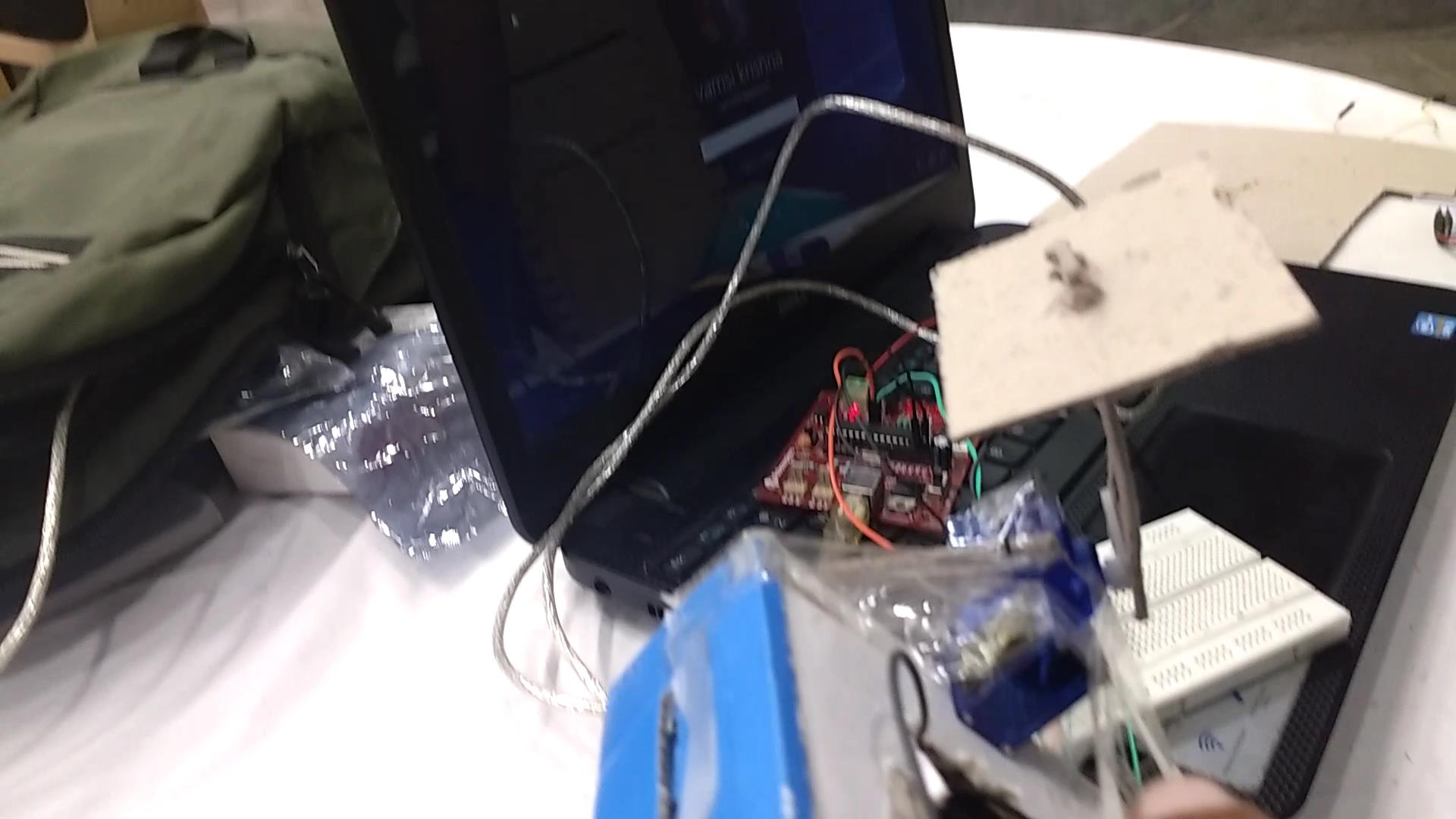
    }

***Working:*** Reading the raw values for the accelerometer and gyro is easy. The sleep mode has to be disabled, and then the registers for the accelerometer and gyro can be read. But the sensor also contains a 1024 byte FIFO buffer. The sensor values can be programmed to be placed in the FIFO buffer. And the buffer can be read by the Arduino. The FIFO buffer is used together with the interrupt signal. If the MPU-6050 places data in the FIFO buffer, it signals the Arduino with the interrupt signal so the Arduino knows that there is data in the FIFO buffer waiting to be read. A little more complicated is the ability to control a second I2C-device.  
The MPU-6050 always acts as a slave to the Arduino with the SDA and SCL pins connected to the I2C-bus.

The sensor has a "Digital Motion Processor" (DMP), also called a "Digital Motion Processing Unit". This DMP can be programmed with firmware and is able to do complex calculations with the sensor values. The DMP ("Digital Motion Processor") can do fast calculations directy on the chip. This reduces the load for the microcontroller (like the Arduino).

We are giving these DMP sensor values (ypr-yaw, pitch and roll) to Servo motor to compensate inclination. This is the working prototype screenshot.





***Screenshots of the prototype prepaid***

***Conclusion:*** This have many applications like,

* Camera stabilization
* Used in cars to feel comfortable stabilization
* Used in wheel chair of the hospitals making the patient sitting in the chair normal to the ground.