SELF-BALANCING ROBOT

How it works?

The aim of a self-balancing robot is to balance itself on two wheels, being able to drive around without toppling over. Self-balancing robots use a "closed-loop feedback control" system; this means that real-time data from motion sensors is used to control the motors and quickly compensate for any tilting motion in order to keep the robot upright.

Inspiration: YAPE - Politecnico di Milano.

Programming Language: C++
Tools & Libraries: I2Cdev, MPU6050
IDE: Arduino IDE

Electronic components:

- Arduino UNO board
- Power supply/battery
- Motor Shield up to 4 motors L239D arduino compatible (motor controller)
- Gyroscope and accelerometer module (Adafruit MPU-6050 6-DoF Accel and Gyro Sensor)
- 4 high torque gearbox motor
- Wires
- Power switch on off

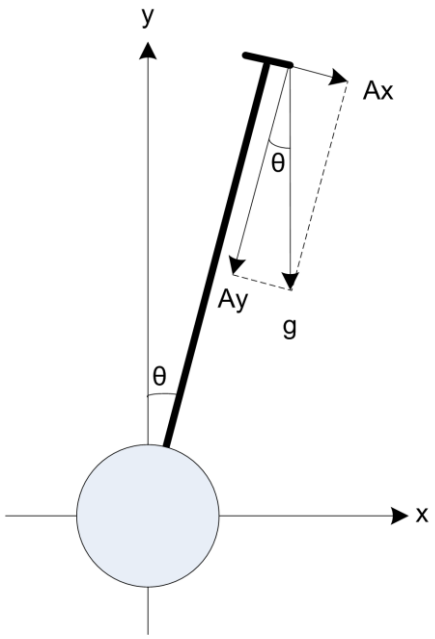
Mechanical components:

- Grippy rubber material for wheels (optional)
- 6x aluminium servo horns
- 8x 10mm diameter bearings (5mm internal diameter) x 4mm depth
- 12x 10mm M3 bolts
- 12x 15mm M5 bolts
- 4x 30mm M5 bolts
- 16x 15mm M4 bolts
- 20x M5 lock nuts and washers

ACCELEROMETER-GYROSCOPE MODULE

Goal: adjust the wheels' position so that the inclination angle remains stable at a predetermined value (the angle when the robot is balanced). When the robot starts to fall in one direction, the wheels should move in the falling direction to correct the inclination angle.

How to use an accelerometer to measure the inclination angle?[1]



The inclination angle can be calculated as:

$$\theta = tan^{-1}\frac{A_x}{A_y} = sin^{-1}\frac{A_x}{\sqrt{A_x^2 + A_y^2}} = sin^{-1}\frac{A_x}{g}$$

In the equations above, Ax is the accelerometer reading along its x axis and Ay is the accelerometer reading on its y axis. When the robot is stationary, g is the gravitational constant (which translates into the accelerometer reading on the y axis when the accelerometer is lying flat). In this inverted pendulum application, we are only interested in calculations where the inclination angle is small since our goal is to ensure that the deviation from equilibrium (typically equilibrium is reached when the inclination angle is close to 0 depending on the weight distribution of the robot) is as small as possible. So we can further simply the above equation for small inclination angles:

$$\theta \approx sin(\theta) = \frac{A_x}{g}$$

By only measuring the *x* axis reading of the accelerometer we can get a rough estimate of the inclination angle. Of course, this is under the assumption that the robot is standing still. In reality, when the robot is not in balance it will accelerate towards the direction it is falling and thus the *x* axis reading of the accelerometer will be slightly more than *Ax* due to the acceleration. At the same time, the *y* axis reading

[1] Source: http://www.kerrywong.com/2012/03/08/a-self-balancing-robot-i/

will be slightly less than the reading in standstill condition (*Ay*). As a result, the combined vector will deviate from *g*. But when the accelerometer is placed near the center of gravity of the robot, the acceleration along the *x* axis is small in near-equilibrium conditions. So the above equation will exhibit some small error, but it remains a good approximation of the inclination angle.
Accelerometer tends to be very sensitive to the accelerations introduced due to movement or vibration and thus the sensor readings will contain some level of noise, which can not be removed easily. So by relying on accelerometer readings alone, we can not get a reliable inclination angle estimate. To address this issue, we will have to rely on some other measurements.

How to use a gyroscope to measure the inclination angle?[2]
Gyroscope can measure the rate at which the rotation is taking place. And the rotation angle for a given time interval is governed by:

$$\theta(t) = \int_{t_1}^{t_2} G(t)dt$$

Where *G(t)* is the gyroscope reading with respect to the rotation direction. When the time interval is small enough, the gyroscope reading can be treated as a constant and thus the above equation can be approximated as:

$$\theta(t) \approx \theta(t_1) + G(t)(t_2 - t_1) = \theta(t_1) + G(t)\Delta t$$

Unlike the accelerometer, gyroscope measurement is largely immune to non-angular movement and thus far less susceptible to vibrations and lateral accelerations mentioned previously. But since the angular measurement is cumulative, any minute error in measurements will manifest over time which causes the estimated angle to deviate from the true value. This is the so-called drifting effect. The gyroscope alone cannot be used to reliably measure the inclination angle either.

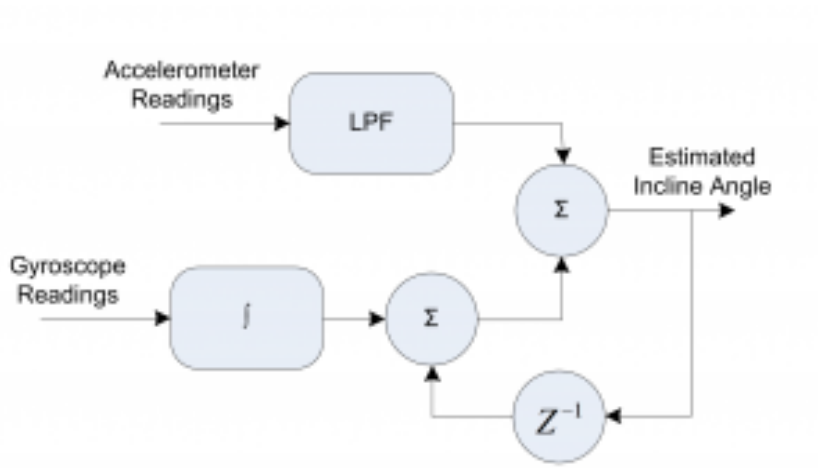
Sensor fusion

To address the issue of measurement noises and the limitations of measurements by either the accelerometer or gyroscope alone, we will need to combine the readings from both the accelerometer and the gyroscope in a meaningful way so that we could use the strengths from both sensors to obtain a more accurate result than either measurement alone could provide. Since accelerometers can provide accurate angle calculations when there is no acceleration and gyroscope can provide accurate short-time angle measurements. These complementary traits are ideal candidates for sensor fusion.
Like many processes in the physical world, sensor readings from the accelerometer and the gyroscope can be modeled as the true measurements with added white Gaussian noise (AWGN). Many filters in the least mean square filters family are suitable for this kind of stochastic process.

Kalman filter is one such adaptive filter that can be used to filter the sensor data from accelerometer and gyroscope. Even though the implementation of Kalman filter is quite straightforward, in order for the filter to be optimal we need to know the precise underlying model of the system and we also need to be able to reliably estimate the noise covariance matrices. Without any reliable information on these

[2] Source: http://www.kerrywong.com/2012/03/08/a-self-balancing-robot-i/

parameters, the filter may still work but will not be in an optimal sense. Simpler methods can be used in situations where these parameters are unknown and can still achieve reasonably good results.
So, to keep the implementation simple, I used the method illustrated below where the estimated value is a linear combination of the filtered measurements from both the accelerometer and the gyroscope. Each sensor reading is multiplied by a fixed gain:

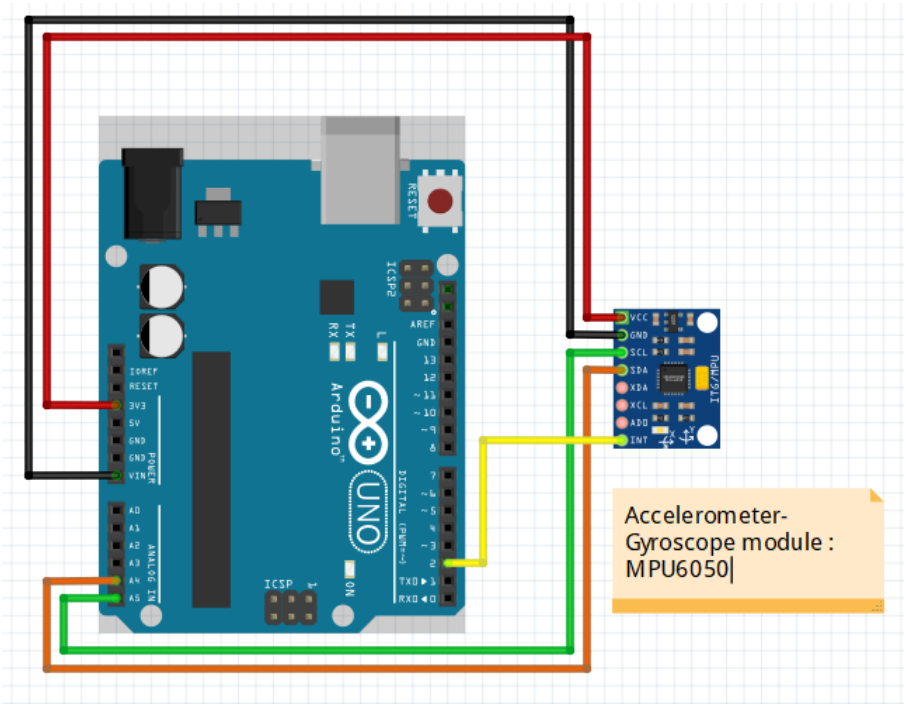


From the diagram above, you can see that in order to make the accelerometer readings more reliable, the readings are passed through a low pass filter (e.g. averaging over time) to smooth out any sudden change in values. And the gyroscope readings are integrated and then added to the previous estimate to give the current inclination angle reading. Each of the components is weighted and then added together to give the final estimate.

The control loop

We now need to control the motor accordingly using this estimate so that the robot can remain balanced. The simplest method one may attempt to employ is to rotate the wheels in the falling direction until the inclination angle reaches the value at which the robot is in balance. Heuristically, the wheel rotation speed should be proportional to the inclination angle so that the robot moves smoothly. This technique is effectively the simplest PID control with both the *I* and the *D* terms set to zero.

The MPU-6050 uses I2C to communicate with the microcontroller, so I started by connecting up the pins as shown in the schematics: the SDA line connects to the Analog pin 4, the SCL to Analog pin 5, power input to the 3.3v pin and the ground to the GND pin.



MANIPULATION OF THE DATA

$$tan^{-1}\frac{A_x}{A_y} = sin^{-1}\frac{A_x}{\sqrt{A_x^2 + A_y^2}} = sin^{-1}\frac{A_x}{g}$$

CALIBRATING THE SENSOR

The MPU-6050 needs to be calibrated before it is used for the first time. What we want to do is remove the zero-error; this refers to when the sensor records a small angle even though it is totally level. This error can be removed by applying an offset to the raw accelerometer and gyroscope sensor readings. The offset needs to be adjusted until the gyroscope readings are zero (no rotation) and the accelerometer records the acceleration due to gravity pointing directly downwards.

To calibrate it I used the sketch in the calibration folder.
The original calibration sketch can be found on the I2Cdev library forum. Upload the sketch and open up the serial monitor in the Arduino IDE, setting the baud rate to 115200. To start calibration, place the accel-gyro module in a flat and level position and send any character in the serial monitor. The program will make an average of a few hundred readings and display the offsets required to remove zero error.

PID Controller

PID stands for Proportional, Integral and Derivative, referring to the mathematical equations used to calculate the output of the controller. Mathematically, the PID controller can be described by the following formula:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt}$$

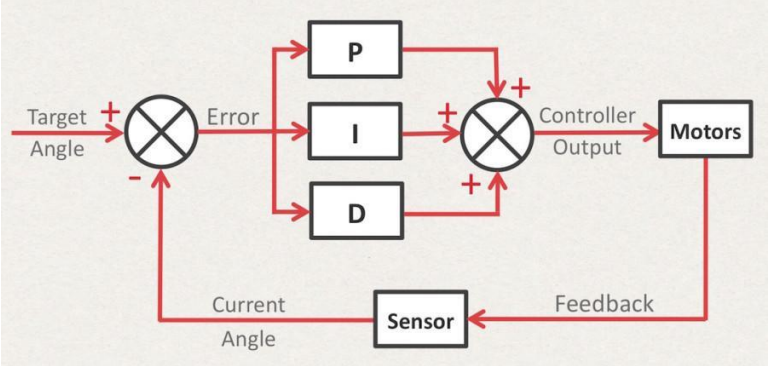
In this formula, the output of the controller *u(t)* is determined by the sum of three different elements, each dependent on the error *e(t)*. The error is the difference between the target value and the current value (measured by the sensor).
The first addendum is the proportional component, which takes the current error value and multiplies it by a constant number (*Kp*). For the self-balancing robot, this simply takes in the current angle of the robot and makes the motors move in the same direction as the robot is falling.
The second addendum is used to accumulate any errors over time, and multiplies this accumulated value by a constant number (*Ki*). For example if the robot tends to fall over to one side, it knows that it needs to move in the opposite direction in order to keep on target and to prevent drifting left or right.
Finally, the derivative component is responsible for dampening any oscillations and ensures that the robot does not overshoot the target value. Each time the controller is called, this term calculates the change in the error value and multiplies it by a constant number (*Kd*). Often this is simplified to calculate only the change in the current sensor value, rather than the change in error. If the target position remains constant, this gives the same result.

Summing all three of these terms together then gives us an output value which we can send to the motors of the robot. However, before the controller can successfully balance the robot, the three constants (*Kp*), (*Ki*) and (*Kd*) need to be tuned to suit our specific application. By increasing or decreasing the values of these constants, we can control how much each of the three components of the controller contributes to the output of the system.

The formula I used above is called the “Independent” PID equation:

$$u(t) = K_c e(t) + \frac{K_c}{T_i} \int_0^t e(t)dt + K_c T_d \frac{de(t)}{dt}$$

The term (*Kc*) is known as controller gain, (*Ti*) the integral time and (*Td*) the derivative time. This format can be useful when using some automated tuning techniques, however the final result is the same.



Implementing the Controller

The controller can be used by calling the "pid" function at regular intervals, providing the target position and the current position (as recorded by the sensor) as parameters of the function. The function then outputs the calculated result of the controller.

First of all, the algorithm calculates the time since the last loop was called, using the "millis()" function. The error is then calculated; this is the difference between the current value (the angle recorded by the sensor), and the target value (the angle of 0 degrees we are aiming to reach).

```
float thisTime = millis();
float dT = thisTime - lastTime;
lastTime = thisTime;
```

The PID values are then calculated and summed up to give an output for the motors. The output is then constrained to ±255 as this is the maximum PWM value that can be output to the motors of the self-balancing robot.

```
// Calculate error between target and current values
float error = target - current;

// Calculate the integral term
iTerm += error * dT;

// Calculate the derivative term (using the simplification)
float dTerm = (oldValue - current) / dT;

// Set old variable to equal new ones
oldValue = current;

// Multiply each term by its constant, and add it all up
float result = (error * Kp) + (iTerm * Ki) + (dTerm * Kd);

// Limit PID value to maximum values
if (result > maxPID) result = maxPID;
else if (result < -maxPID) result = -maxPID;

return result;
```

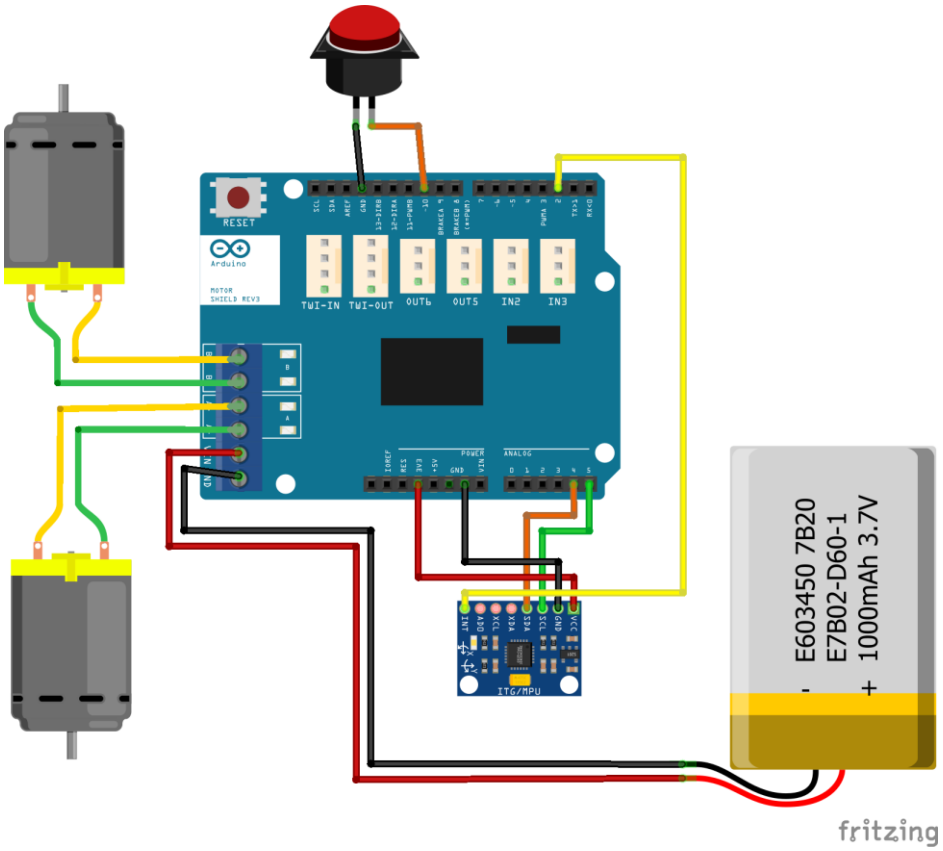
Although the PID controller code is complete, suitable PID constants still need to be found to tune the controller for your specific robot. These constants depend on things such as weight, motor speed and the shape of the robot, and therefore they can vary significantly from robot to robot.

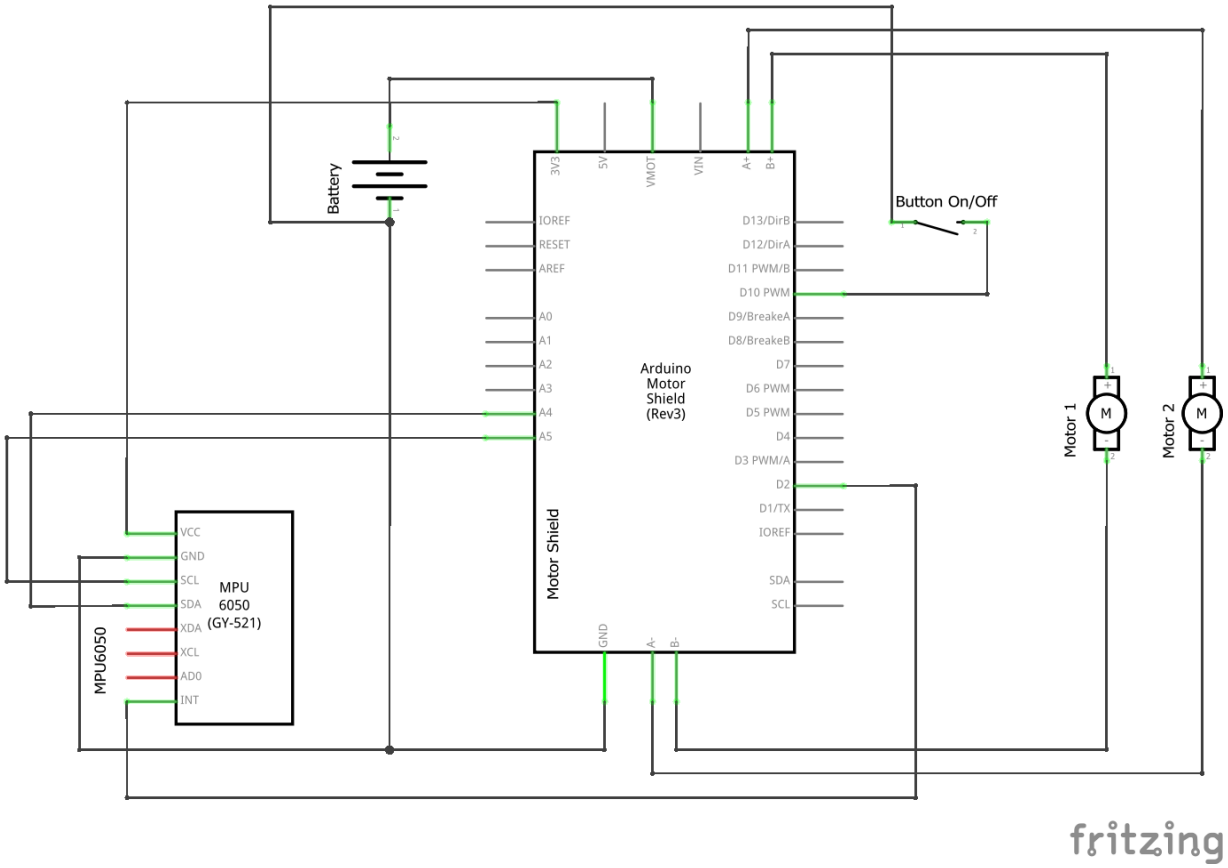
ASSEMBLY

MECHANICAL, MOTORI ECC

SENSOR POSITIONING: the positioning of the accelerometer/gyroscope module is important. It should be on top, as this is where it would record the largest amount of movement.

SCHEMAS:




Battery
Button On/Off
Arduino
Motor
Shield
(Rev3)
Motor Shield
3V3
5V
VMOT
VIN
A+
B+
IOREF
RESET
AREF
A0
A1
A2
A3
A4
A5
D13/DirB
D12/DirA
D11 PWM/B
D10 PWM
D9/BreakeA
D8/BreakeB
D7
D6 PWM
D5 PWM
D4
D3 PWM/A
D2
D1/TX
IOREF
SDA
SCL
GND
A-
B-
Motor 1
Motor 2
M
M
MPU6050
MPU
6050
(GY-521)
VCC
GND
SCL
SDA
XDA
XCL
AD0
INT
fritzing