SELF-BALANCING ROBOT

Instructable appendix

Rik van Haaften, Sander de Lange

TABLE OF CONTENTS

INTRODUCTION

As the title of this project suggests we are going to go through the process of building a self-balancing robot. The principle of something balancing itself is, in physics terms, an inverted pendulum. It is important to realize that because this is an inverted pendulum the system of the robot is inherently unstable. So the project is more about something which can best approach stability as opposed to making a robot which is stable, because it cannot be stable.

ACHIEVING BALANCE

The robot uses a PID controller to balance itself. A PID controller powers the motors based on a mathematical formula. The letters *P*, *I* and *D* stand for *proportional*, *integral* and *differential*.

The proportional component works as simple as multiplying the error by a gain value. Finding the right gain value is absolute key to properly setting up *any* component of the PID controller. In Arduino code, the proportional component is written like this:

```
P = kp * (targetValue - currentValue);
```

Then there is the integral component. An integral correction slowly increases over time, as it is the summation of all errors since the last stable state. This summation or errors is then multiplied by a gain value. This is a simple form of an integral component in code:

```
currentError = targetValue - currentValue;
totalError += currentError;
I = ki * totalError;
```

The differential component calculates the change in error, which is then—just like the other components—multiplied by a gain value. This is what the differential component looks like when written in Arduino code:

```
currentError = targetValue - currentValue;
D = kd * (currentError - lastError);
lastError = currentError;
```

When these three components are put together in one summation, you get the basic form of a PID controller:

```
currentError = targetValue - currentValue;
totalError += currentError;

pid_output = kp * currentError + ki * totalError
       + kd * (currentError - lastError);

lastError = currentError;
```

If you have looked at the PID controller code of the self-balancing robot, you may have noticed that the controller uses not three, but four components. This is because the controller is based on the nBot Balancing Robot[1] algorithm.

[1] http://www.geology.smu.edu/~dpa-www/robo/nbot/

The nBot algorithm uses the following four components, of which two are proportional and two are differential:

- angle;
- angle derivative (angular velocity);
- wheel position;
- wheel position derivative (wheel velocity).

We changed the wheel position from a proportional component to an integral one, as this makes the program keep track of the robot's displacement from its original position. As a result, the robot will try to return to its original position after giving it a push, without the need for a lot of extra code.

A problem with this approach has to do with the gain value. A higher gain will get the robot closer to the original position, but will make the act of balancing itself a bit sloppier. We chose for a relatively low gain, because the priority of the robot is its ability to be controlled and move—and therefore not sticking to a certain position.

CODE

Low level

In our code we decided to use the addresses of the chip functions as opposed to calling the pin number and using `digitalWrite`. We did this as we discovered that calling the pin numbers took longer and was higher level in coding whilst calling the address is lower level and takes less time. Because every chip has its own chip addresses the manufacturers usually make a register chart with all addresses and their descriptions.

The other thing we used at a lower-than-usual programming level were the Arduino's timer registers. A timer register is a component on the chip which is connected to the master clock. The ATmega328(P) has three timers. They are configurable, but you should generally not touch the first register as this one is set, by default, to count the number of milliseconds the program has been running for. Modifying the first timer may make your code unpredictable, as certain operations (such as `millis` and `delay`) will not count in whole milliseconds anymore.

The timers are used to trigger code execution at specific time intervals. In case of our self-balancing robot, this interval has been set to 4 milliseconds. Because timers run independently from the code itself, it is important to note that one balancing sequence should be completed in less than 4 milliseconds. Otherwise, the next loop will start before the previous one has completed.

BUILDING THE ROBOT

Process

We divided building the robot into two parts: first we tried to get all the electronics to work and have an idea of why and how it works (since we gathered snippets of code from existing projects), and the second part was centred more around designing the best exterior for the robot. This allowed us to more efficiently work on the project as we were both exploring the do and don'ts of the project.

Pitfalls

When doing this project—especially when you have got a deadline—it is important to realize the time it takes to ship the products, as being surprised by how long it can take to ship items can really put you in a rough spot. We have personally experienced this when we ordered some parts to arrive before our project week, but it took quite a while for them to be shipped.

It is also important to realize the budget you are willing to spend and what kind of products you want to use in building your robot, of course when you spend more money you get better parts, but it is also important to realize a certain balance within the project, so that you will not run into any straining bottlenecks. You would not want to spend half you budget on better motors only to later find out you need to buy an additional battery or IMU.

A budgeting problem we have experienced in this project was with the battery of our robot, we had one lying around so we thought we would just use that one and be done with it, however it appeared to be insufficient, causing our budget to go over our predicted amount and it also increased the amount of time we needed to finish our project.

Other than money or time related pitfalls it should also be taken into consideration how strong the materials used are. And what components will endure the most stress. It is safe to say that the circuit boards and the battery should not be under any stress really, and that the sides of the robot should be sturdy. As there is a very present risk of the robot falling over, and it would be a shame if falling down a few times would destroy the main body of the robot. Another tip is the connection between the wheels and the motors of the robot, as each one of them is under the stress of ½ of the mass of a well-balanced robot. Another solution, albeit we did not experiment with it, could be making a third wheel in the middle of the already present axis. This would take some of the stress away from the other two wheels.

Construction

Putting the robot together and running the code proved to be one of the easier chapters of our project. The main issue we ran into was that we misread the colour codes on the resistors of our voltage divider, which led to a way too high resistance—reducing the voltage our Bluetooth chip received to nearly zero. Other than that, it was self-explanatory; connecting the labelled pins to the labelled connectors.

Designing for balance

Earlier in this document we have talked about how the mainboard achieves balance, but ultimately achieving balance is less important than being balanced from the beginning. This is mainly because the more effort we put into making a balanced robot the less effort the robot must put into keeping itself balanced. Which in turn heavily ties into the dynamics of an inverted pendulum.

Within an inverted pendulum like ours we are trying to accommodate for the angular acceleration with horizontal acceleration. And if you make the distance between the centre of mass you increase the time you have before the robot falls over, as angular acceleration is largely dependent on the gravitational acceleration. And while if you take a longer distance between the wheels and the centre of mass the angle between the pendulum and the centre of mass will not change, the distance between the centre of mass and the ground will. This does not come without downsides though, as the overall mass of the robot correlates with the length of the robot. And the heavier the robot, the more torque is required from the motors. Which in turn means that to make our robot as stable as possible we need to get the centre of mass as high as possible without over encumbering the motors. In short: the better the motors, the higher you can make your robot, the better the robot will be balanced.

Prototypes and trial and error

While we were putting the components of our robot together we encountered some hurdles. The main one being that we simultaneously needed a prototype to test our code on, whilst the code we were testing might not work on the final shape of the robot. At one point we were trying to use cardboard wheels, which did not give us the grip required to actively figure out if the code was functional. The prototype design was a bit of a paradox, because we were not able to figure out what shape to build our prototype to fit the code, we were not able to fit the code to the body of the prototype.

Luckily for us there was a multitude of designs already made before we started our project, which proved to be very helpful because we could construct a prototype akin to the already finished designs, so that the code we were using could be configured to fit the shape our project was probably going to take. As if we were figuring out what way to shape our prototype through trial and error could take months.

Centre of mass

To calculate where our centre of mass would be in our robot we used the following formula:

$$Centre\ of\ mass = \frac{\Sigma(m_n * x_n)}{\Sigma(m_n)}\ for\ n = 1,2,3,4\ldots$$

In this formula m is the mass of an object in our robot and x is the distance of the object to the “beginning” of our robot, which of course is one of the sides of the robot.

While applying this formula we conjure a 2D top down image of our robot as this image does not really give us a 3D calculation. The 2D/3D problem could be solved by simply

doing the same thing again but then in the other direction. Again, we decided that our biggest shot of making this a successful project and Instructable would be to take some inspiration from people who have done similar projects before us, as we would have to go through a lot of potential variables in the design of our robot.

Final design

What we ended up settling upon was a rhombus like shape with flat corners, like shown in our Instructable. This allowed us more than enough space around the wheels so the robot could angle itself on the ground whilst it also gave us the space to fit the parts and then still have leftover space for the wiring. We then decided we wanted to use wood as the body of our robot, as wood is quite easy to work with, and more forgiving, cheap and lighter than metal or aluminium. The main drawback being that it is less aesthetically pleasing, but the robot's aesthetics are not the main concern in this project. We screwed most of the circuit boards in place, but we also used Velcro strips to attach our battery, to easily be able to change its position and test if the robot has better stability with a higher / lower centre of mass.

EXPLICATION OF THE CODE

The code starts with `GCC optimize`. This is an instruction to the compiler to optimize the code and make it as small as possible, so that the controller's chip can execute it faster. The number `3` refers to the level of optimization, which is the highest possible.

Next up are some integers to keep track of the code timing. If the robot doesn't work properly, you can have the loop time of the code sent into another data field of the joystick app so that you can verify it stays under 4 milliseconds.

Using `define` is a handy way of noting down the registry map of the IMU, because it simplifies the programming while taking zero extra space on the controller. The compiler simply replaces the text values with the actual addresses, and the code is more readable because you know what is being called (for instance `ACCEL_XOUT_H`, which will be replaced by `0x3B`).

To control brushless motors, you need to output a sine wave to each pin with a 120-degree phase shift. The PWM lookup table is taking care of that. It eliminates the need for the controller to calculate the values every time a code loop runs, which makes the code run faster.

In the `setup()` function, the controller establishes a serial connection with the Bluetooth chip, and gives the user five seconds to put the robot on either its front or its back before the IMU will be calibrated.

To communicate with the IMU, the `Wire` function is used. Some settings, such as the sensitivity of the IMU are being set, and the robot's angle will be calibrated by taking 1024 measurements with 10 millisecond delays. The total calibration will take 10.24 seconds.

Next up is setting the pins to which the motors are connected as output pins, as well as setting the controller's timers properly to allow the PWM lookup table to be output over the pins with a correct timing.

The control loop will run every 4 milliseconds. It will check whether the power button in the controller app has been turned on, and then run an angle calculation, followed by the PID control loop. If the power button is in the 'off' position, the robot will turn the motors off and stop balancing. It will also reset the control loop.

To calculate the robot's angle, a combination of the gyroscope and accelerometer is used, because the gyroscope alone is not accurate enough and will drift over time. This is done using a complementary filter. The calibration offset is added on each measurement, too.

Before running, the program checks whether the robot is within one degree of being perfectly vertical, before running. When the control loop runs, it takes the last angle measurement and applies a correction accordingly. There are calculations for the derivatives and other loop components, each multiplied by their gain value. Bluetooth joystick input is taken into account with another calculation, to get the final motor output.

BIBLIOGRAPHY

Anderson, D. P. (2012, February 1). *nBot Balancing Robot.* Retrieved from Geophysics Research Archives: http://www.geology.smu.edu/~dpa-www/robo/nbot/bal2.txt

Anonymous. (2013, August 19). *MPU-6000 and MPU-6050 Register Map and Description Revision 4.2.* Retrieved from InvenSense: https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf

Anonymous. (2015, January). *ATmega328P Datasheet.* Retrieved from Atmel Corporation: http://www.atmel.com/images/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

Anonymous. (2016, April 15). *What PIDs do and how they do it*. Retrieved from YouTube: https://www.youtube.com/watch?v=0vqWyramGy8

Anonymous. (n.d.). *Arduino 101: Timers and Interrupts*. Retrieved from Let's Make Robots!: https://www.robotshop.com/letsmakerobots/arduino-101-timers-and-interrupts

Avery, P. (2009, March 1). *Introduction to PID control*. Retrieved from MachineDesign: http://www.machinedesign.com/sensors/introduction-pid-control

Chi Ooi, R. (2003). *Robotics & Automation Lab.* Retrieved from The University of Western Australia: http://robotics.ee.uwa.edu.au/theses/2003-Balance-Ooi.pdf

Hayes, S. (2013, September 4). *PID Math Demystified*. Retrieved from YouTube: https://www.youtube.com/watch?v=JEpWlTl95Tw