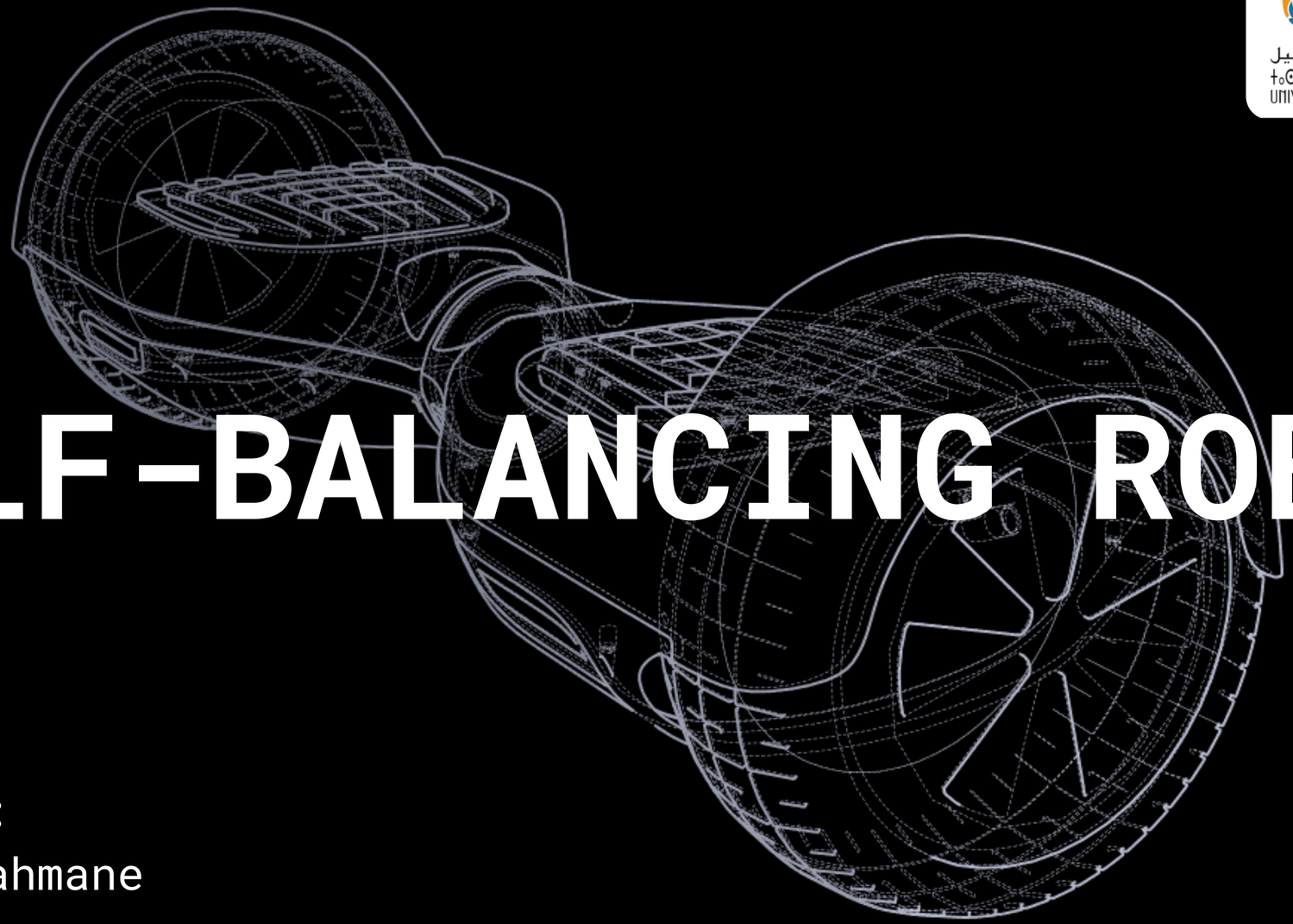


SELF-BALANCING ROBOT

Presented by:
Aarab Abderrahmane
Fajri Walid

PRESENTATION FLOW

1- Introduction

- What is a SBR?
- 2 wheeled SBR

2- Theory

- Mechanical System
- IMU
- Control System (PID)

3- Demonstrator

- Components
- Assembly
- Software

4- Conclusion

- Showcase
- Conclusion
- Perspectives

INTRODUCTION

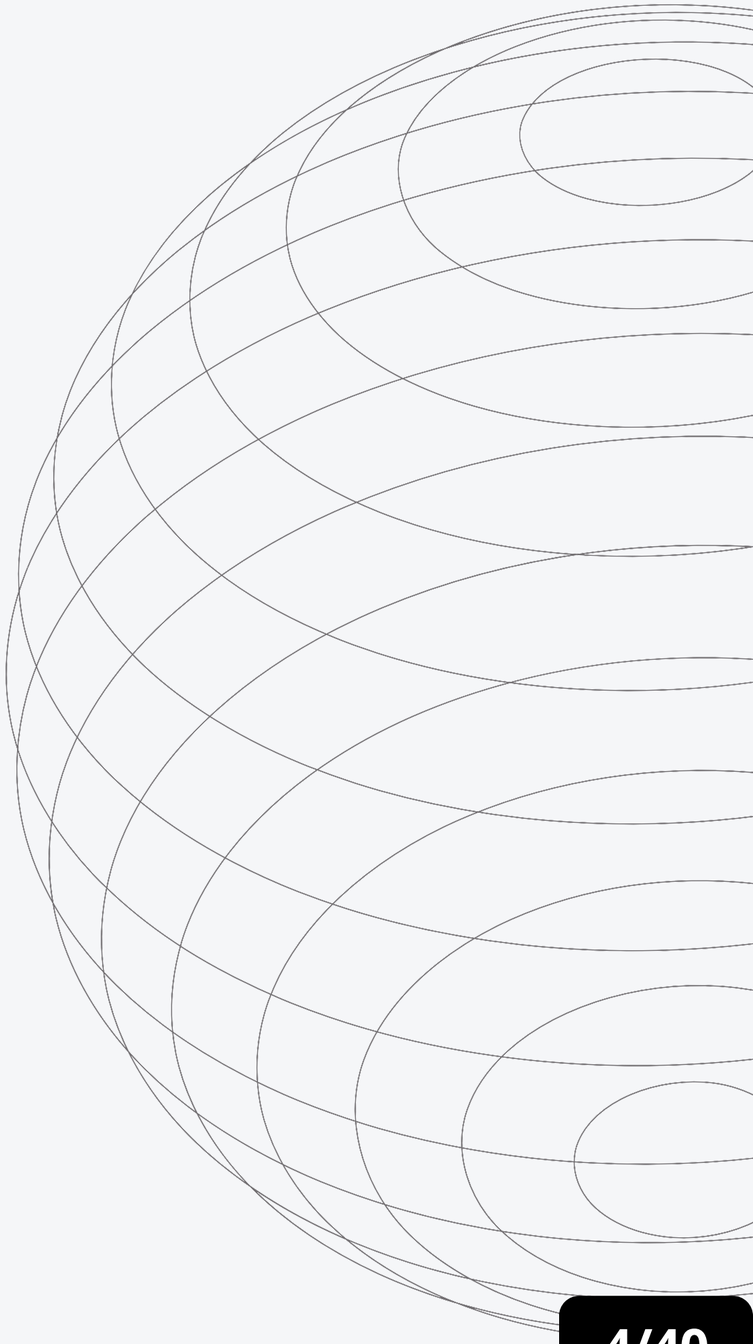
WHAT IS A SBR?

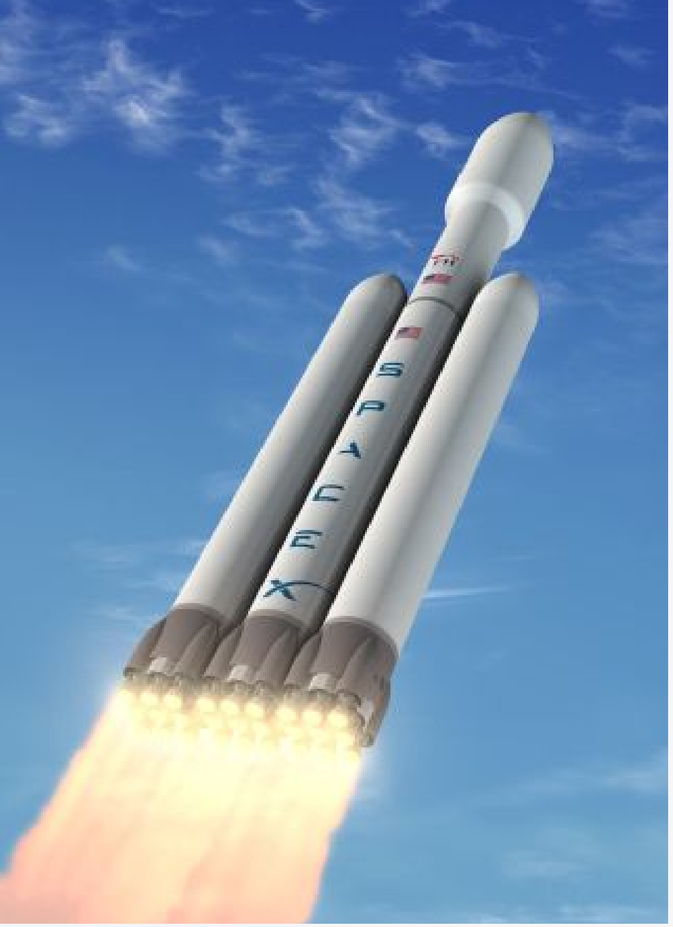
A **SBR** (**Self Balancing Robot**) is a robot that balances itself using a balancing system which is usually a **“closed-loop feedback control”** system. Similar self-balancing feedback control systems can be seen in many other applications. Some of the obvious examples include Segways, bipedal robots and space rockets (A few rockets have been lost due to a **faulty balancing system**).

But what many people don’t realise is that often the same type of controller is also used in a large variety of other applications which aren’t related to balance. **Proportional-integral-derivative (PID)** controllers are used by:

- Elevators to control their motion and position.
- Air-conditioning units to control the temperature within a room.
- Jet engines.

Of course rockets use significantly more complex controllers than air-conditioners, but the underlying **principle** is still the same: **How to adjust the system in order to get as close to the desired target value (be it temperature, angle, or position) as possible.**





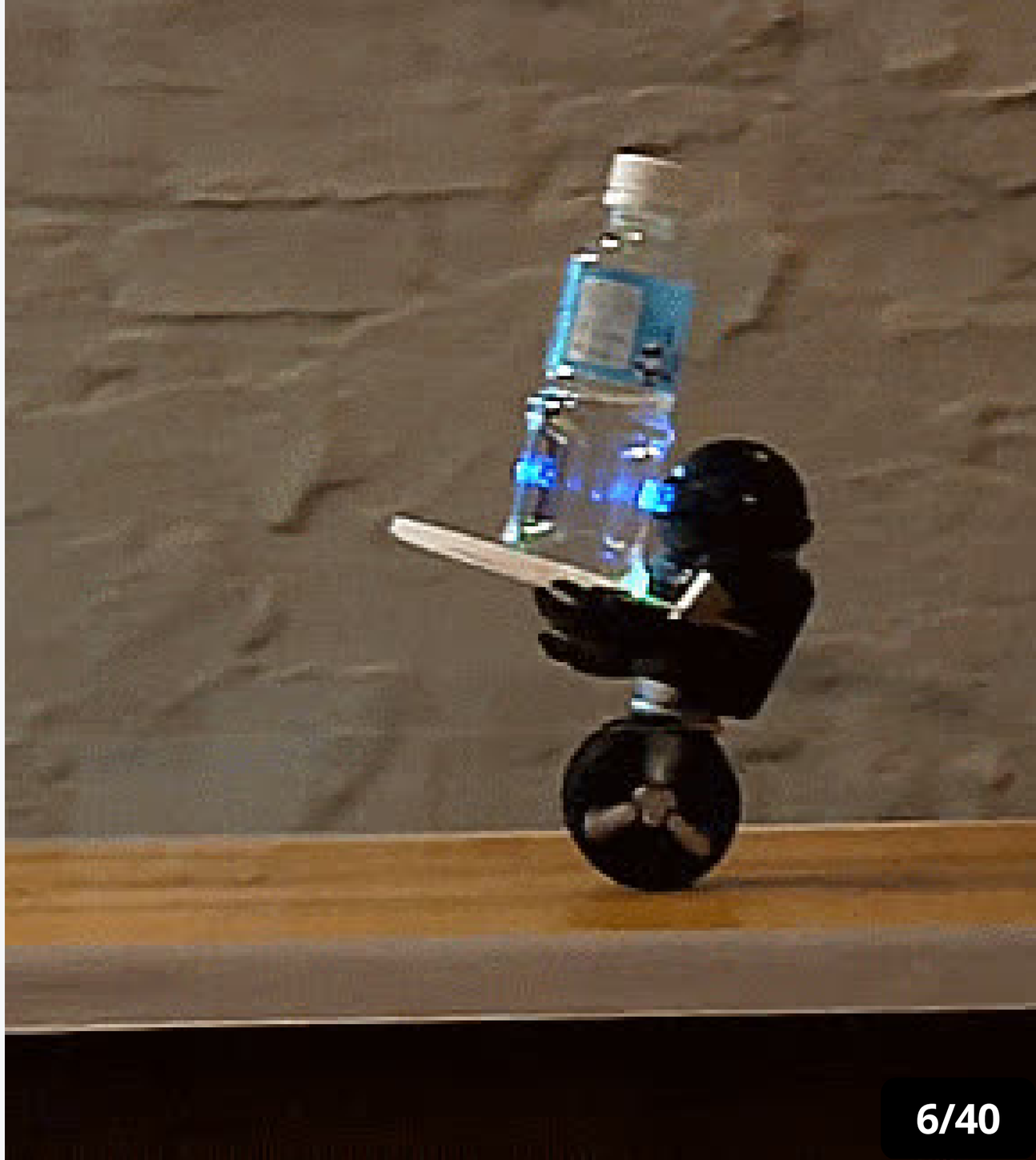
EXAMPLES OF SELF-BALANCING SYSTEMS

2 WHEELED SBR

DEFINITION:

The aim of a **2 wheeled SBR** **(self-balancing robot)** is to balance itself on two wheels, being able to drive around without toppling over.

This robot has many advantages over traditional **four wheeled robots** as it helps in taking sharp turns and navigating through tighter areas thus, serving as an essential machine for various industrial applications.



2 WHEELED SBR

PRINCIPLE:

The principle of a **2 wheeled SBR** is similar to that of the human body. The human body is an **inverted pendulum** balancing the upper body around the ankles.

It uses 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**.

2 WHEELED SBR

ADVANTAGES:

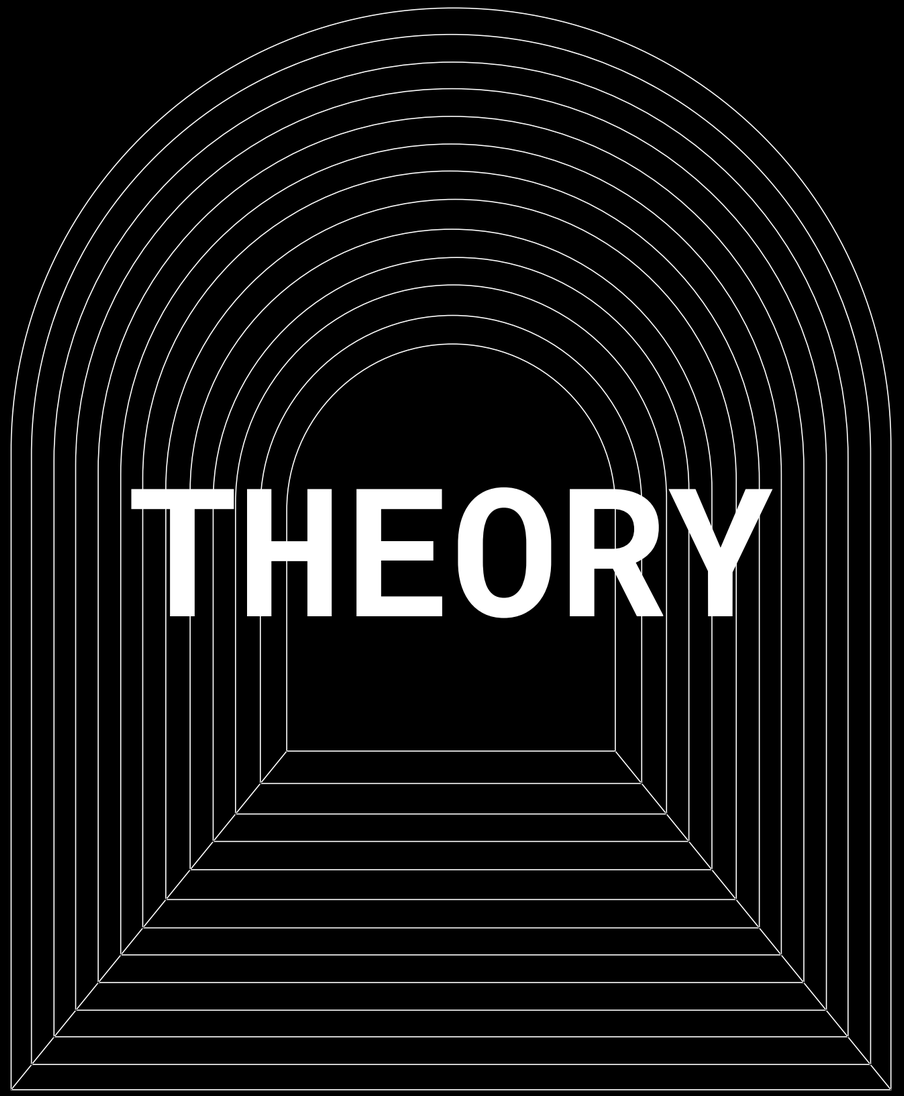
Two wheeled SBRs have advantages compared to other configurations, :

- They are mechanically simple,
- They have a zero turn radius,
- They have superior stability in inclines since they lean into the incline.

This could make **two wheeled SBRs** useful for automated package deliveries.

DISADVANTAGES:

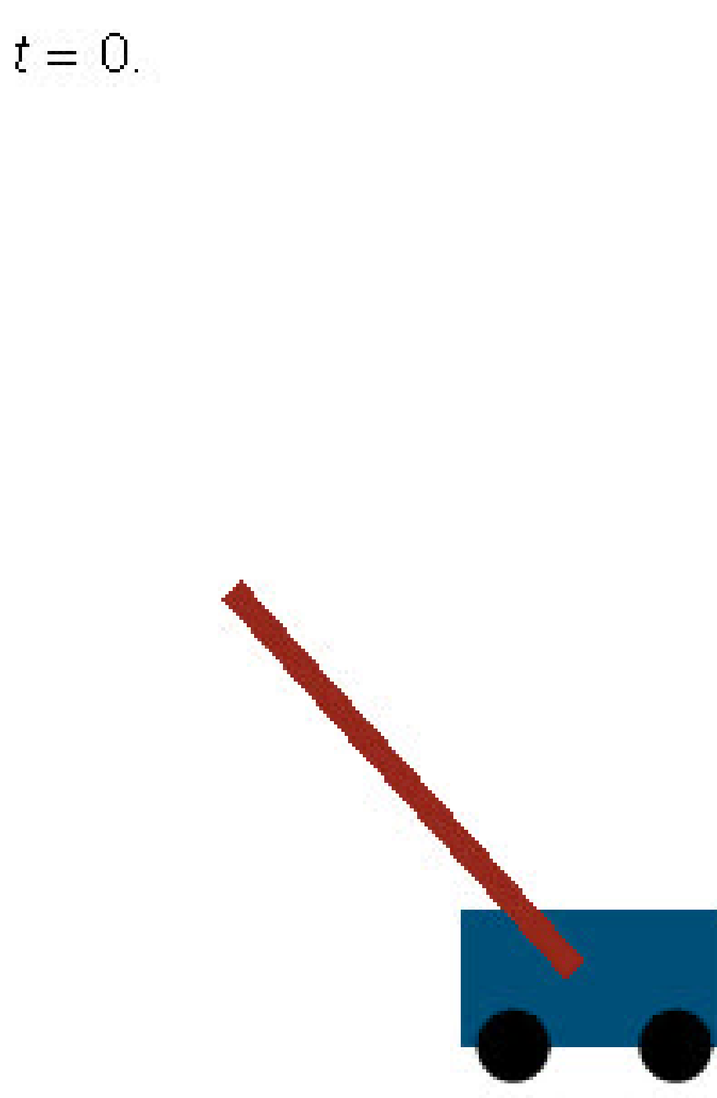
The main drawback of **2 wheeled SBRs** is poor navigation over obstacles and rough terrain.

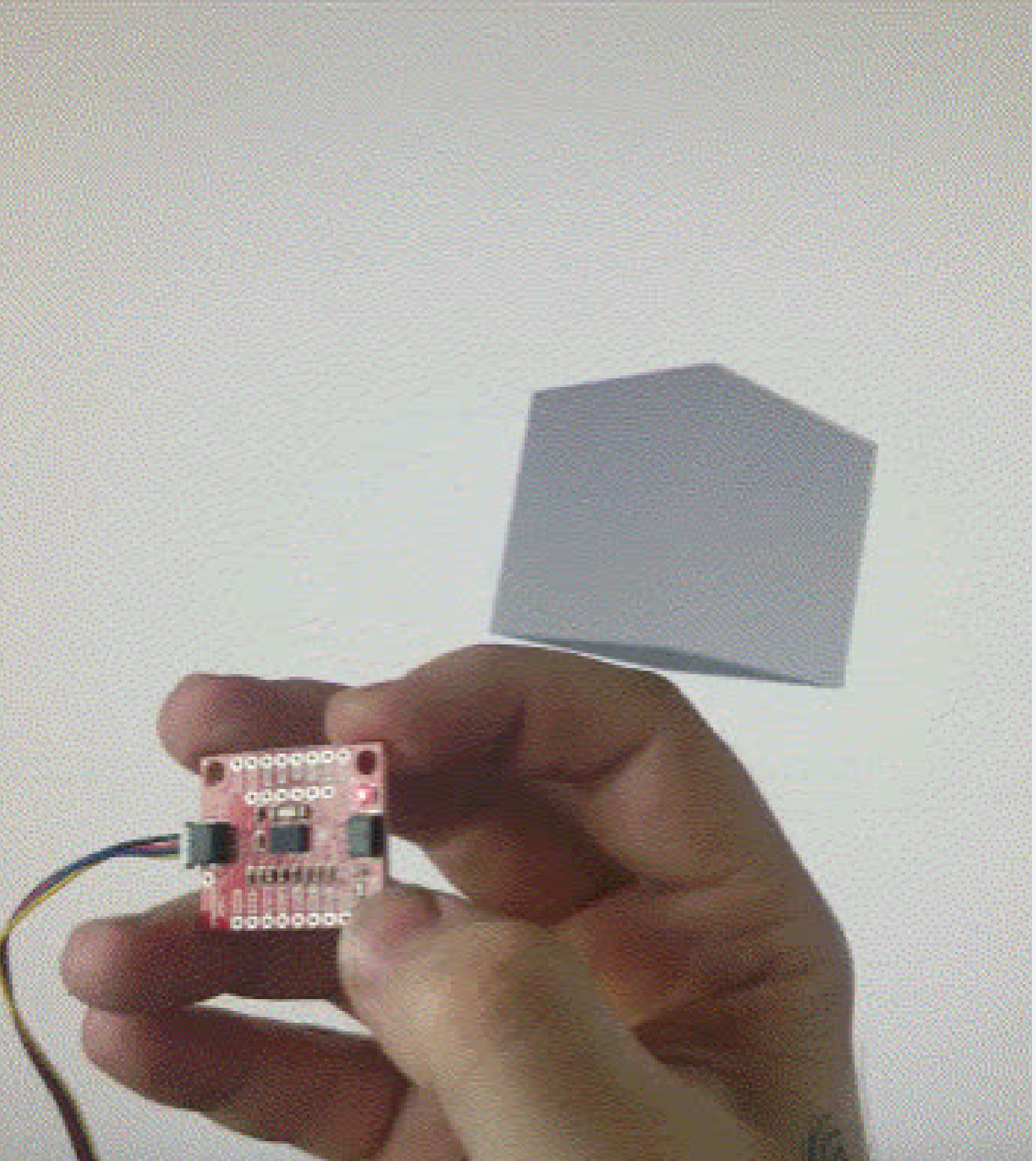

THEORY

MECHANICAL SYSTEM

The principle of creating a **2 wheeled SBR** is similar to the **inverted pendulum** principle.

When a **tilt** from the equilibrium occurs the **motors** will generate a torque that drives the wheels in the **same direction** as the tilt, the wheels will move the same distance as the center of gravity in order to **maintain balance**.





IMU

An **inertial measurement unit (IMU)** is an electronic device that measures and reports a body's **specific force**, **angular rate**, and sometimes the **orientation of the body**, using a combination of **accelerometers**, **gyroscopes**, and sometimes magnetometers.

IMUs are typically used to maneuver modern vehicles including motorcycles, missiles, aircrafts, and spacecraft, including satellites and landers. It's an essential part of our **SBR**.

CONTROL SYSTEM

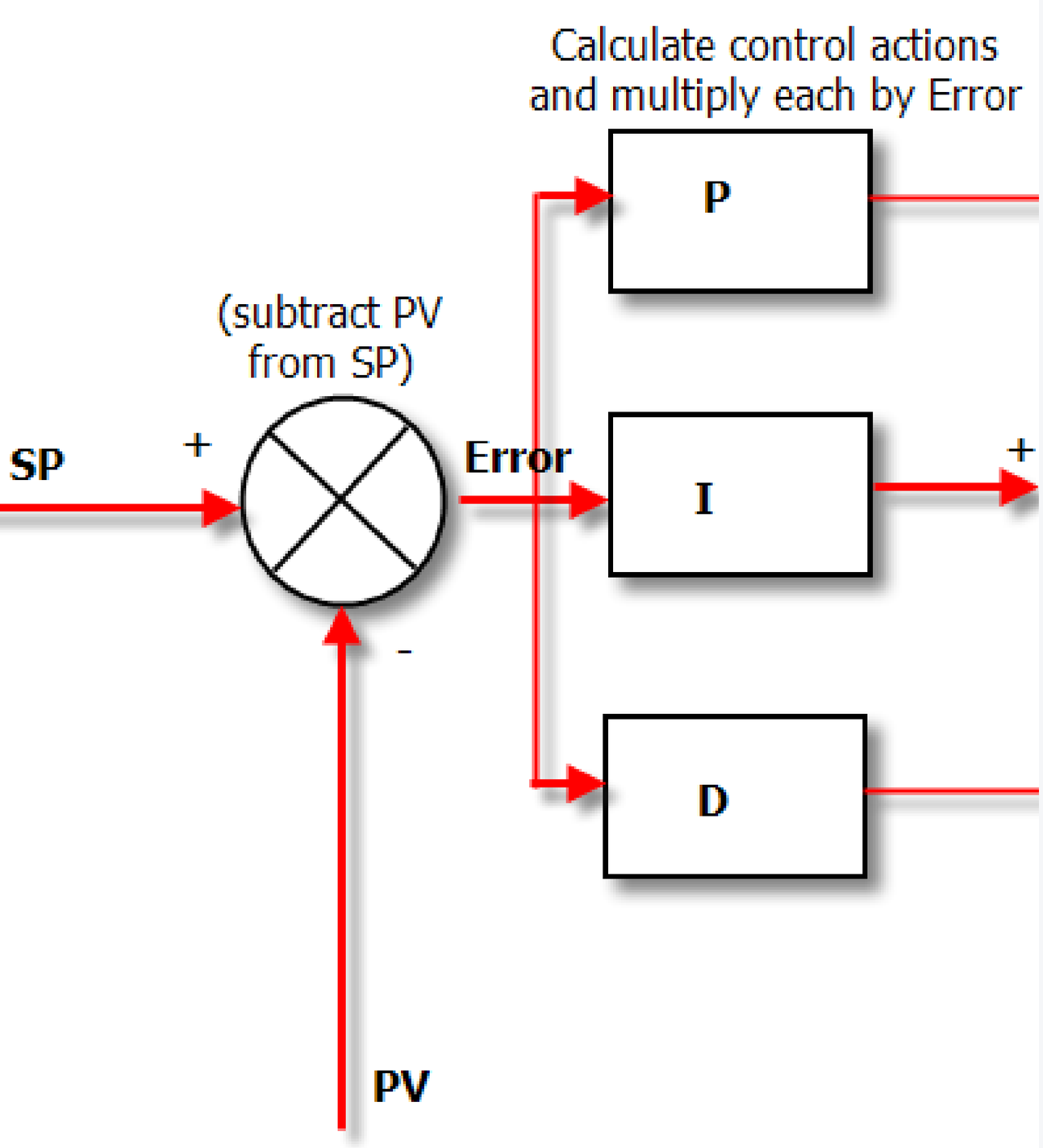
To apply the **inverted pendulum** principle to our **SBR** we first need a robust control system. A "**closed-loop feedback control**" system will get **real-time data** from the **motion sensor** (**IMU**) and use it to control the **motors** and quickly compensate for any **tilting** motion in order to keep the robot upright and **balanced**.

We'll be using a **PID** controller as our control system to stabilize the robot.

PID CONTROLLER

A **proportional–integral–derivative** controller (**PID**) is a **control loop** mechanism employing **feedback** which means that the **output** of the algorithm is fed back as **input** to improve accuracy and minimize error.

PID controllers are widely used in **industrial control systems** and a variety of other applications requiring continuously **modulated** control.



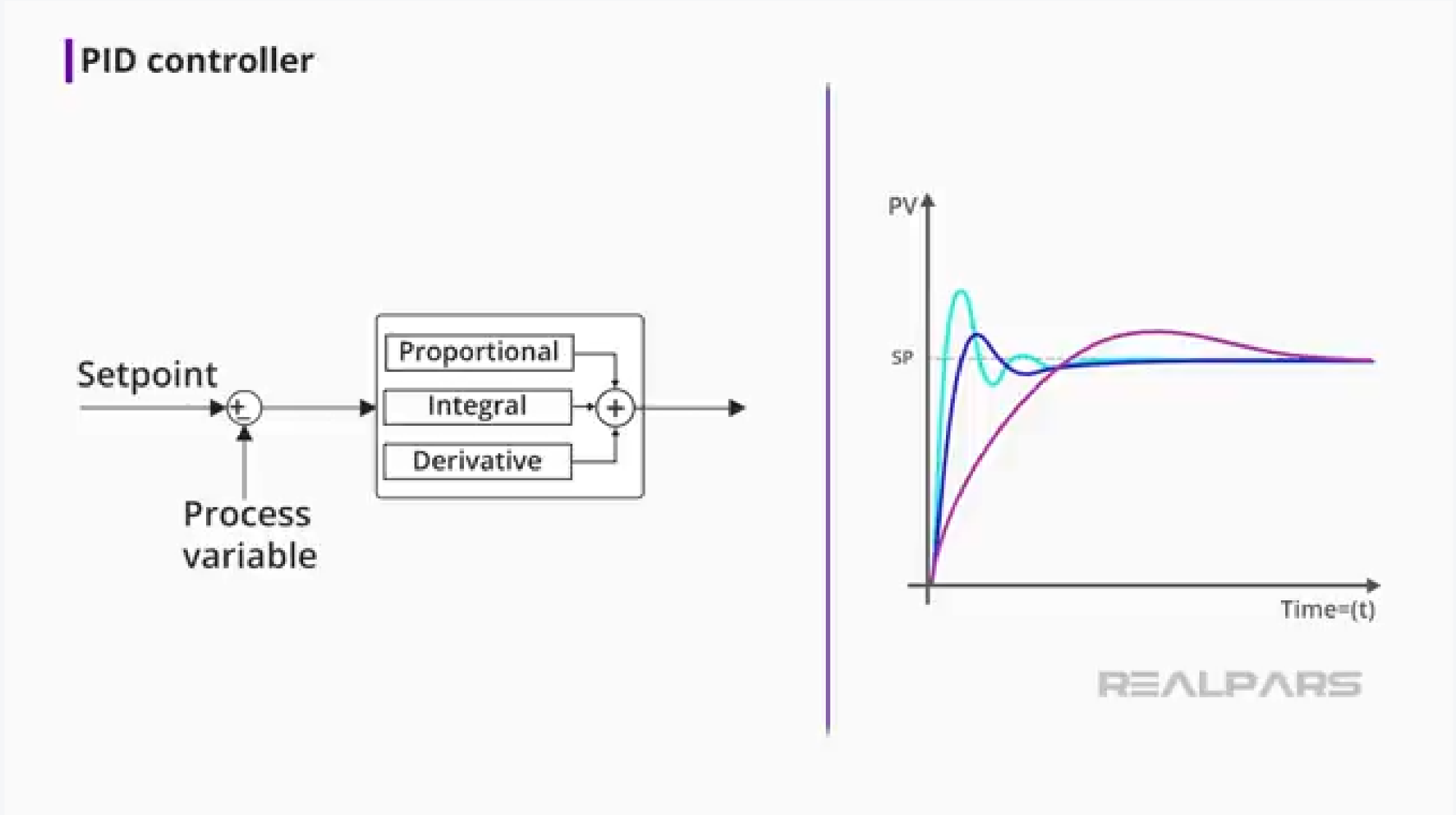
PID CONTROLLER

A **PID** controller continuously calculates an error value **e(t)** as the difference between a desired **setpoint** (**SP**) and a measured **process variable** (**PV**) and applies a correction based on

Proportional,
Integral, and
Derivative terms

(denoted **P**, **I**, and **D** respectively), hence the name.

PID CONTROLLER



PID CONTROLLER

Mathematically, the **PID** controller can be described by the following formula:

SetPoint $\mathrm{SP} = r(t)$
Process Var $\mathrm{PV} = y(t)$
Error $e(t) = r(t) - y(t)$

$$u(t) = \underbrace{K_\mathrm{p} e(t)}_{\mathbf{P}} + \underbrace{K_\mathrm{i} \int_0^t e(\tau)\,\mathrm{d}\tau}_{\mathbf{I}} + \underbrace{K_\mathrm{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t}}_{\mathbf{D}},$$

- **SP= r(t)** is the position you want your robot to be in.
- **PV= y(t)** is the current calculated position of the robot. (Calculated by the IMU)
- **e(t)** is the error experienced. (Ex: 90 - 95 = 5 Degrees)
- **Kp**, **Ki**, and **Kd** are coefficients that need to be inputted manually. (PID Tuning)

They are all added together into an output control signal **u(t)** that is passed on to the system.

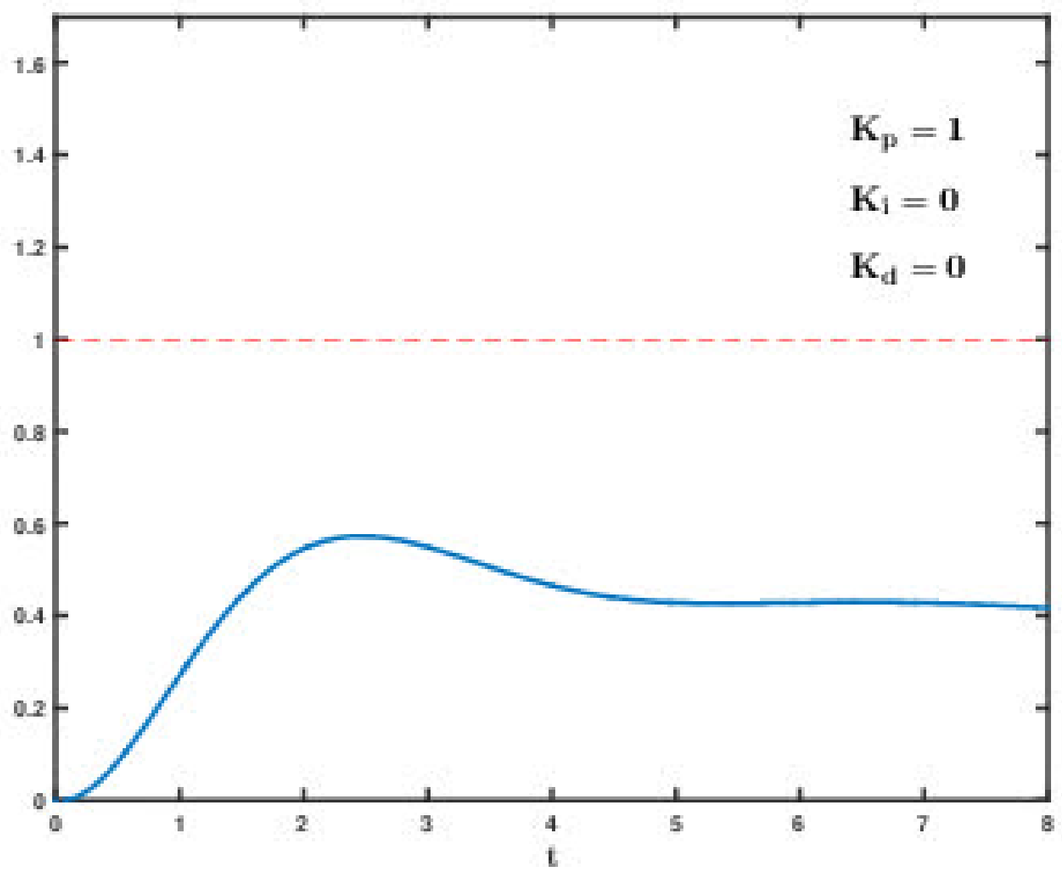
PID CONTROLLER - PID TUNING

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(\tau)\,\mathrm{d}\tau + K_\mathrm{d} \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

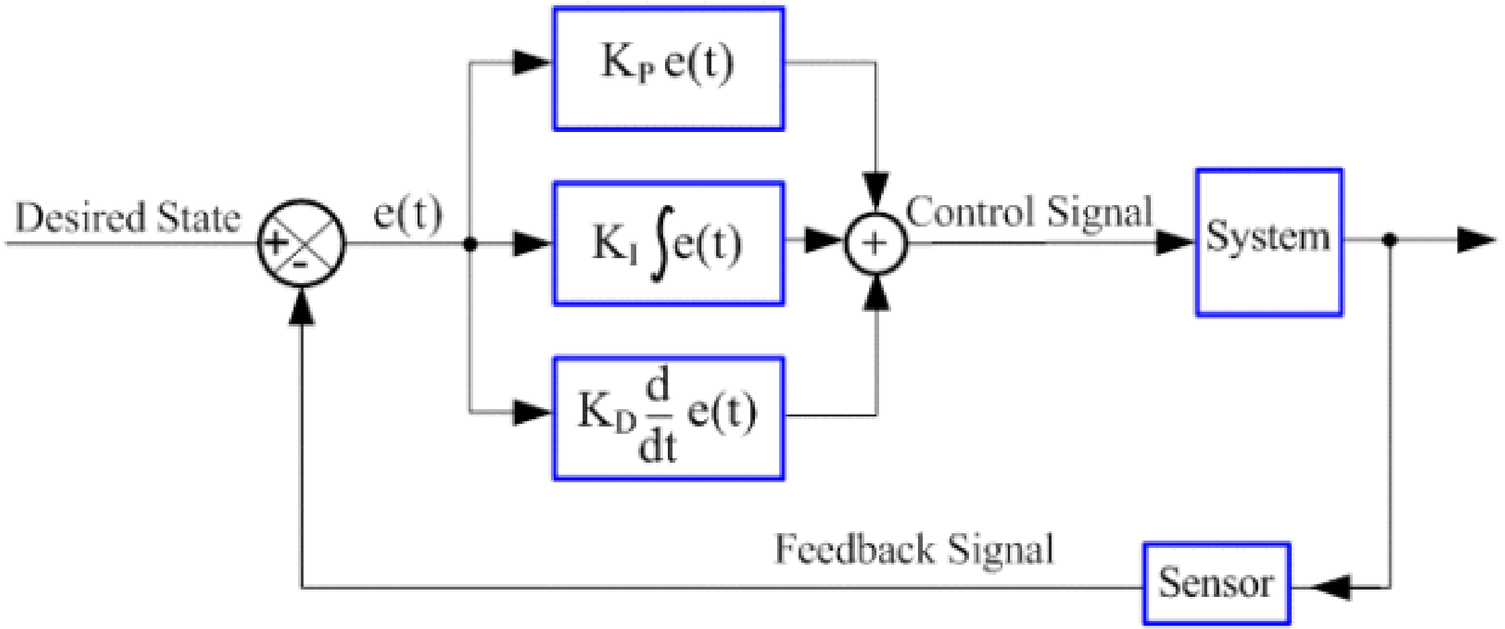
P I D

Finding the right **PID controller parameters** for our use case is what we call **PID Tuning**.

The **Kp**, **Ki**, and **Kd** 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.



PID CONTROLLER



PID Controller Diagram

DEMONSTRATOR

DEMONSTRATOR

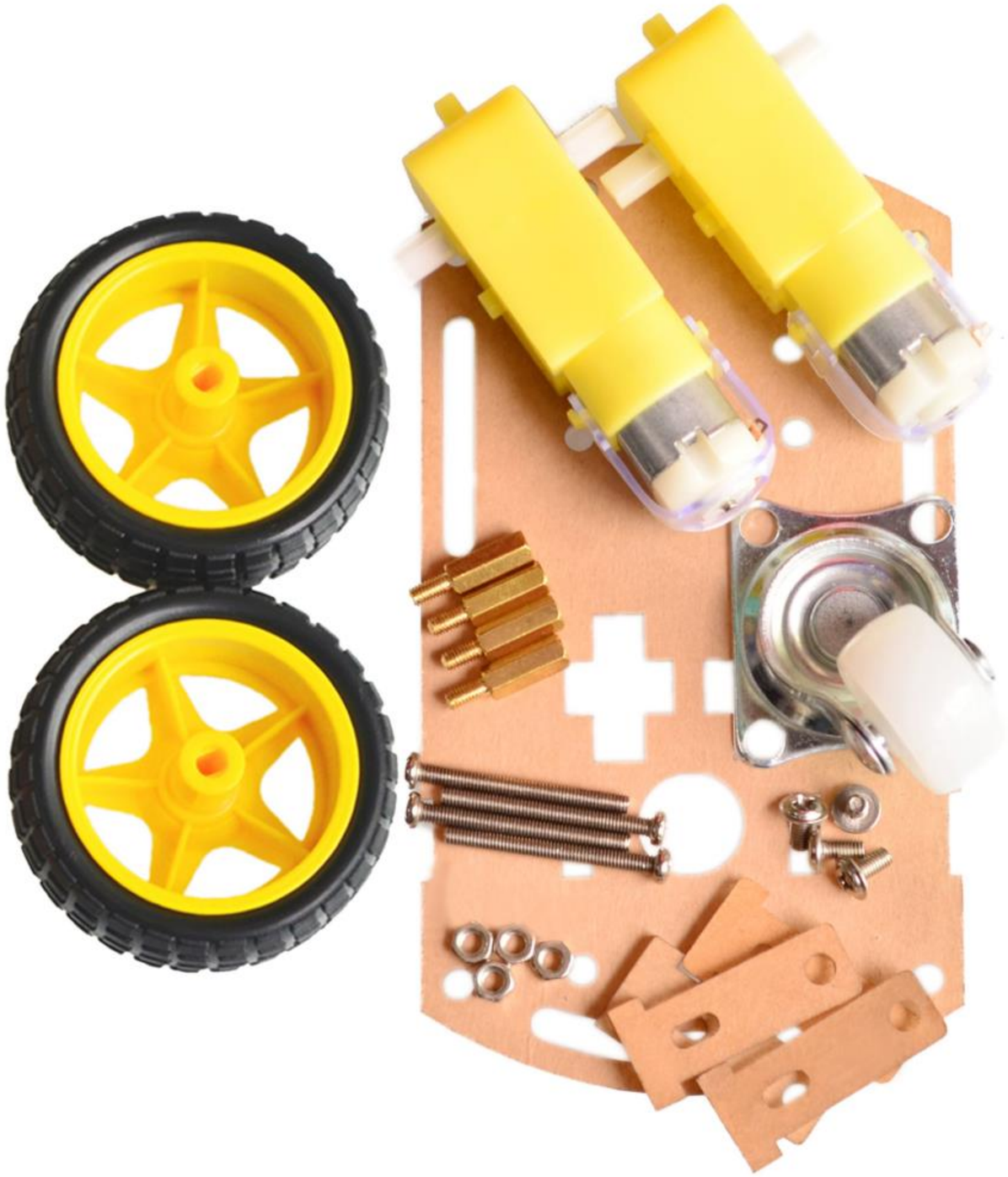
It takes several steps to build a **SBR**. The easiest part is the **hardware**. Surprisingly enough due to the inertia of an **inverted pendulum**, the robot actually finds it easier to stabilize if the frame is **very tall** with a **lot of weight on top**, instead of a small frame with a low centre of gravity!

COMPONENTS

Hardware required for our **SBR**:

- **Mechanical** Components:
  - 4x Support rods
  - 2x Wheels
  - Some type of frame

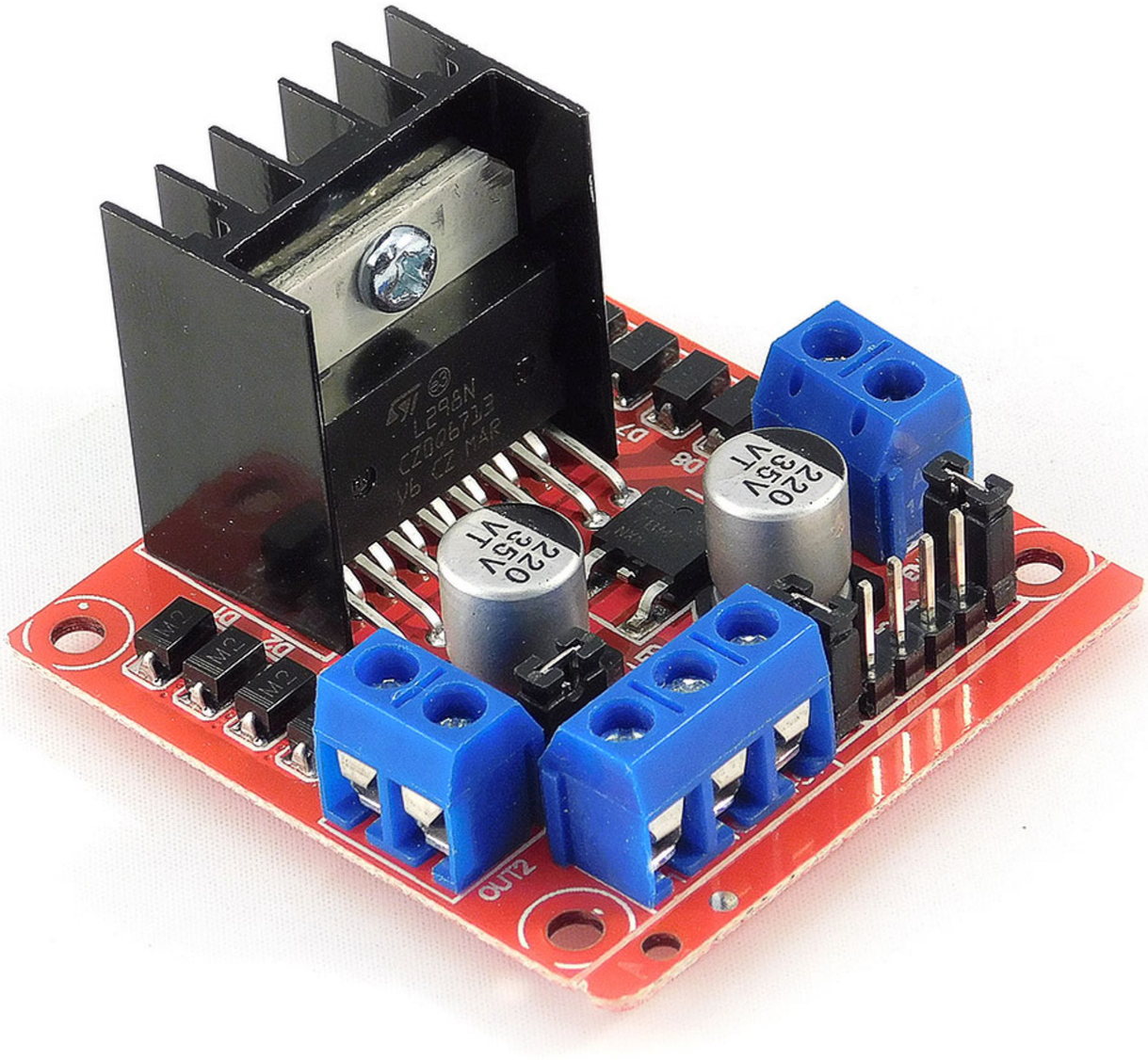
- **Electrical** Components:
  - 2x motors
  - 1x motor controller
  - 1x sensor to detect its tilt angle
  - 1x MCU (micro-controller unit)



MECHANICAL COMPONENTS

Components from RC Car kit:

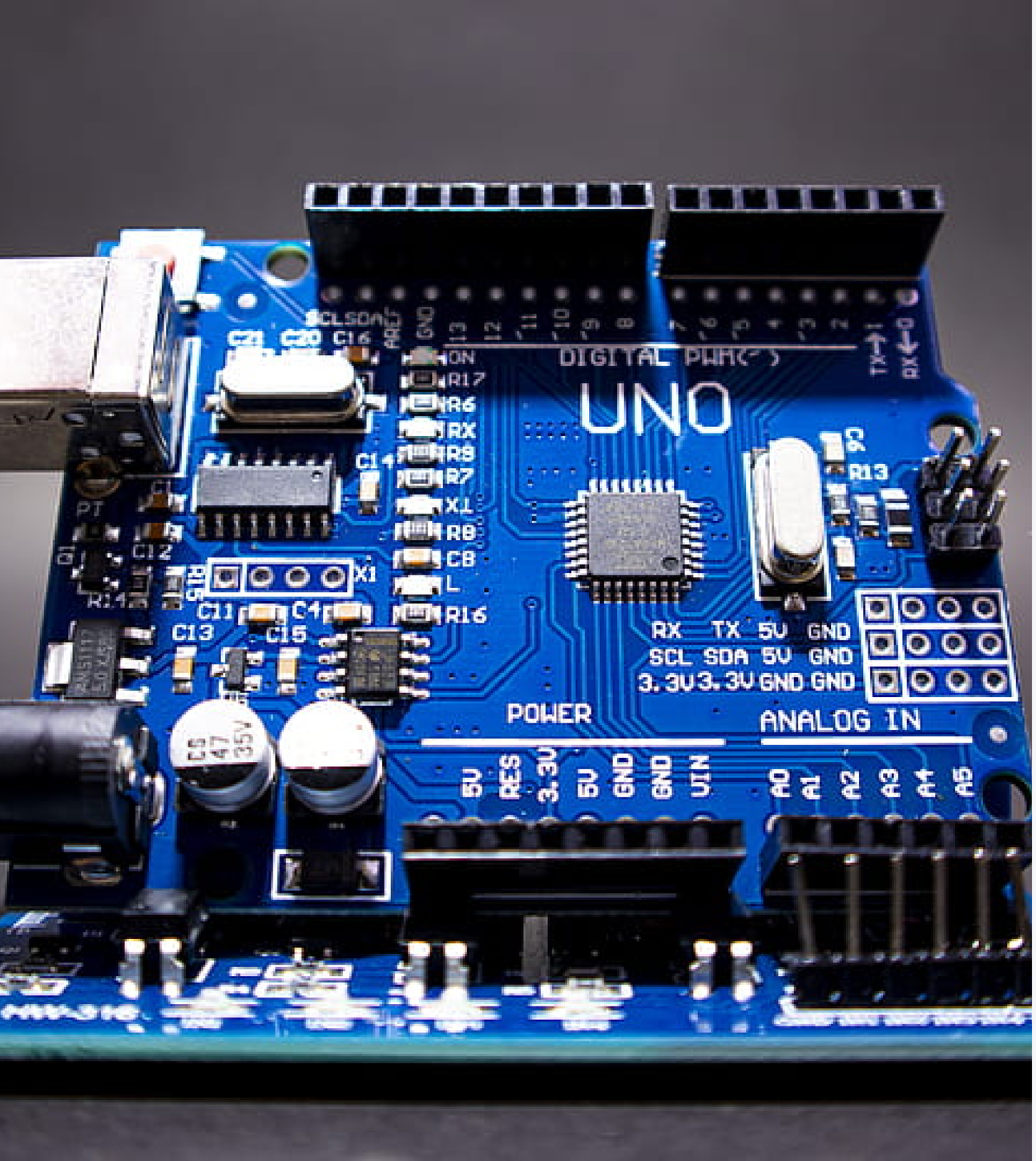
- 4x RC Car Acrylic Frames
- 2x Generic DC geared motors
- 2x Generic car wheels



ELECTRICAL COMPONENTS

L298N Dual H-Bridge Motor Controller

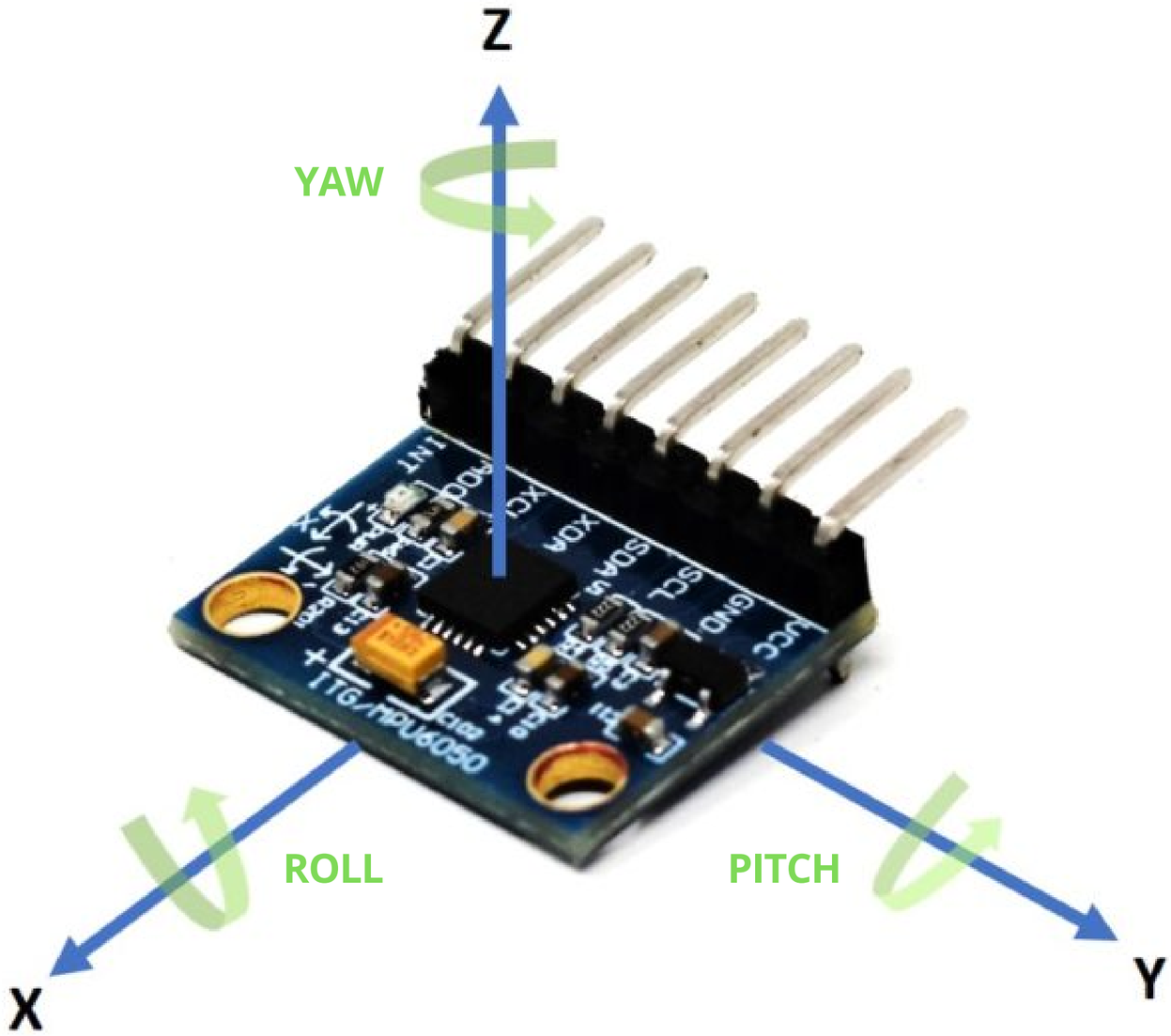
This L298N Motor Driver Module is a high power motor driver module for driving DC and Stepper Motors. This module consists of an L298 motor driver IC and a 78M05 5V regulator. L298N Module can control up to 4 DC motors, or 2 DC motors with directional and speed control.



ELECTRICAL COMPONENTS

Arduino UNO R3

Arduino UNO is a microcontroller board based on the ATmega328P. It has 14 digital input/output pins (of which 6 can be used as PWM outputs), 6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header and a reset button.



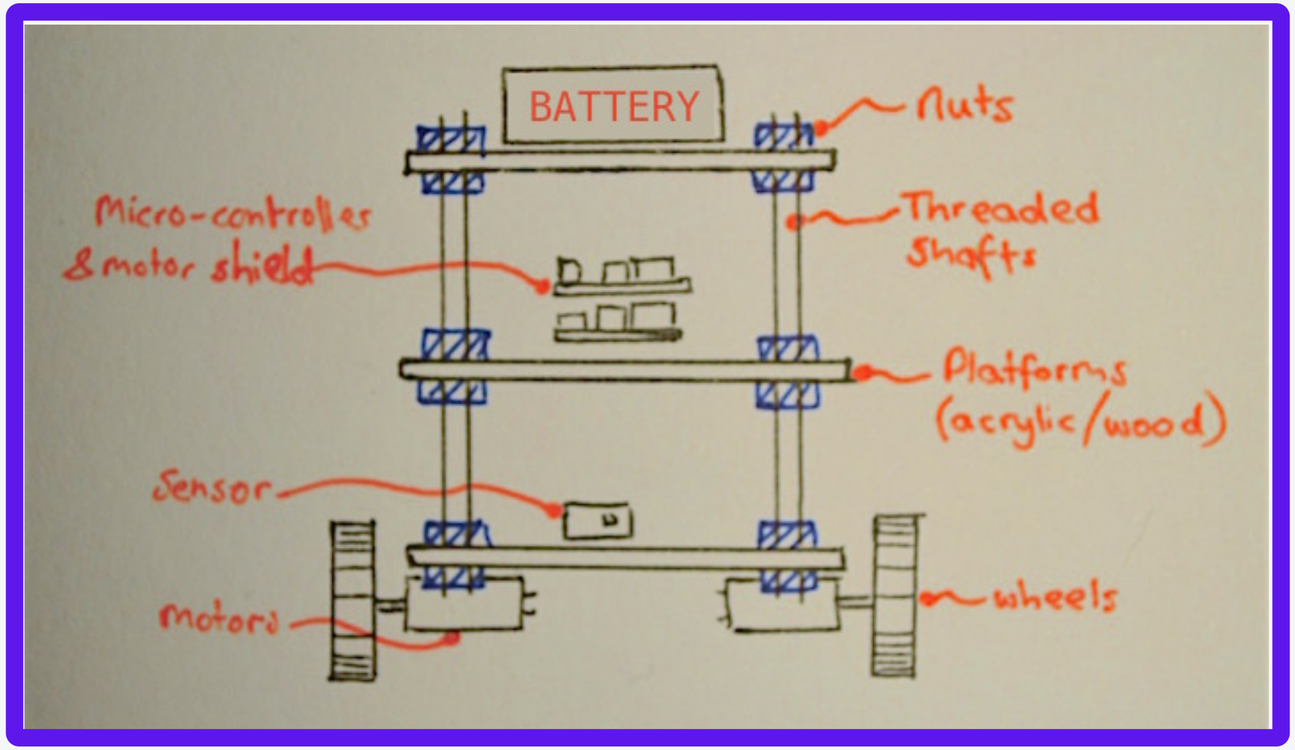
ELECTRICAL COMPONENTS

MPU6050 Accelerometer & Gyroscope

This **IMU** consists of 3-axis accelerometer and 3-axis gyroscope. It helps us to measure **velocity**, **orientation**, **acceleration**, displacement. This module uses the I2C module for interfacing with Arduino. Its main feature is that it can easily combine accelerometer and gyroscope.

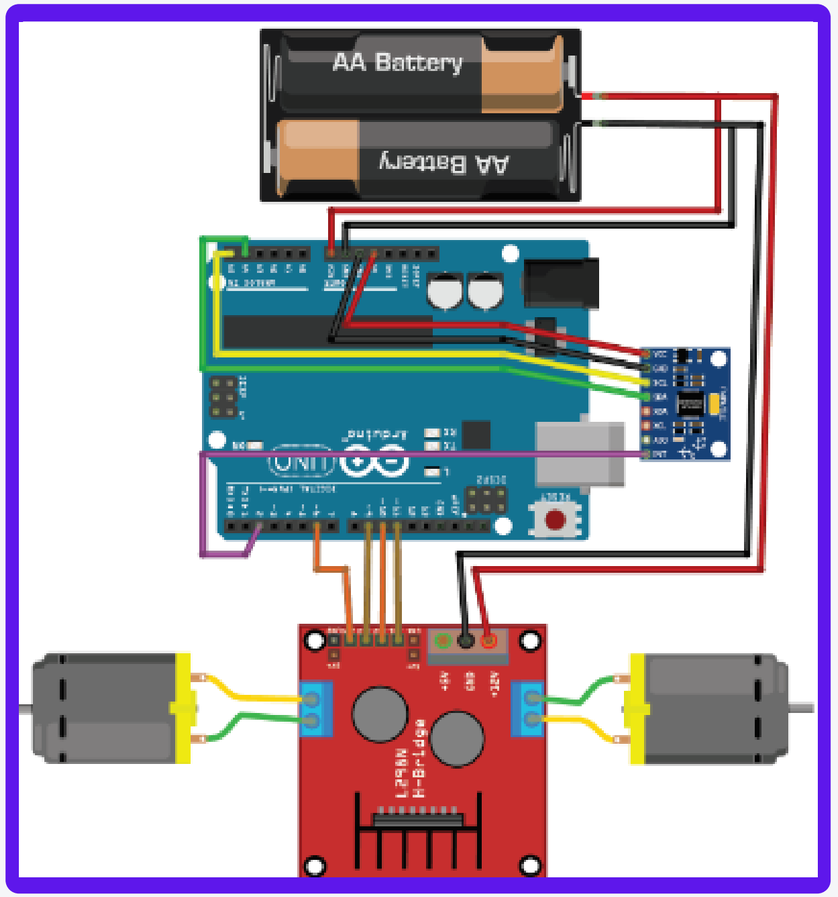
ASSEMBLY - CHASSIS

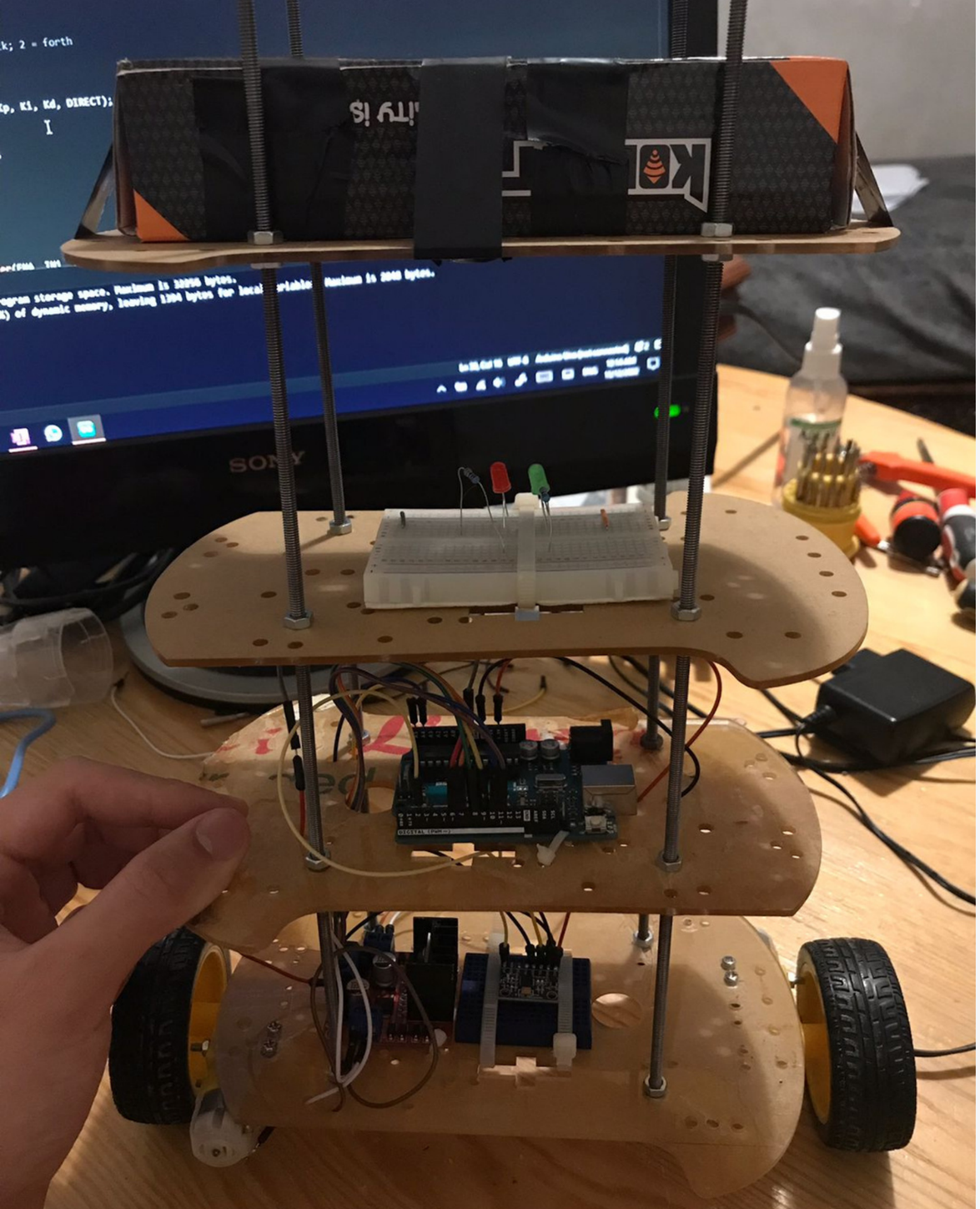
We built a SBR prototype according to a frame design which is commonly found is to use a couple of plastic/wood rectangles as platforms, and to connect these using a long threaded bolt and some nuts (as shown in the diagram below).



ASSEMBLY - SCHEMATIC

We then wire everything up using the schematic below:



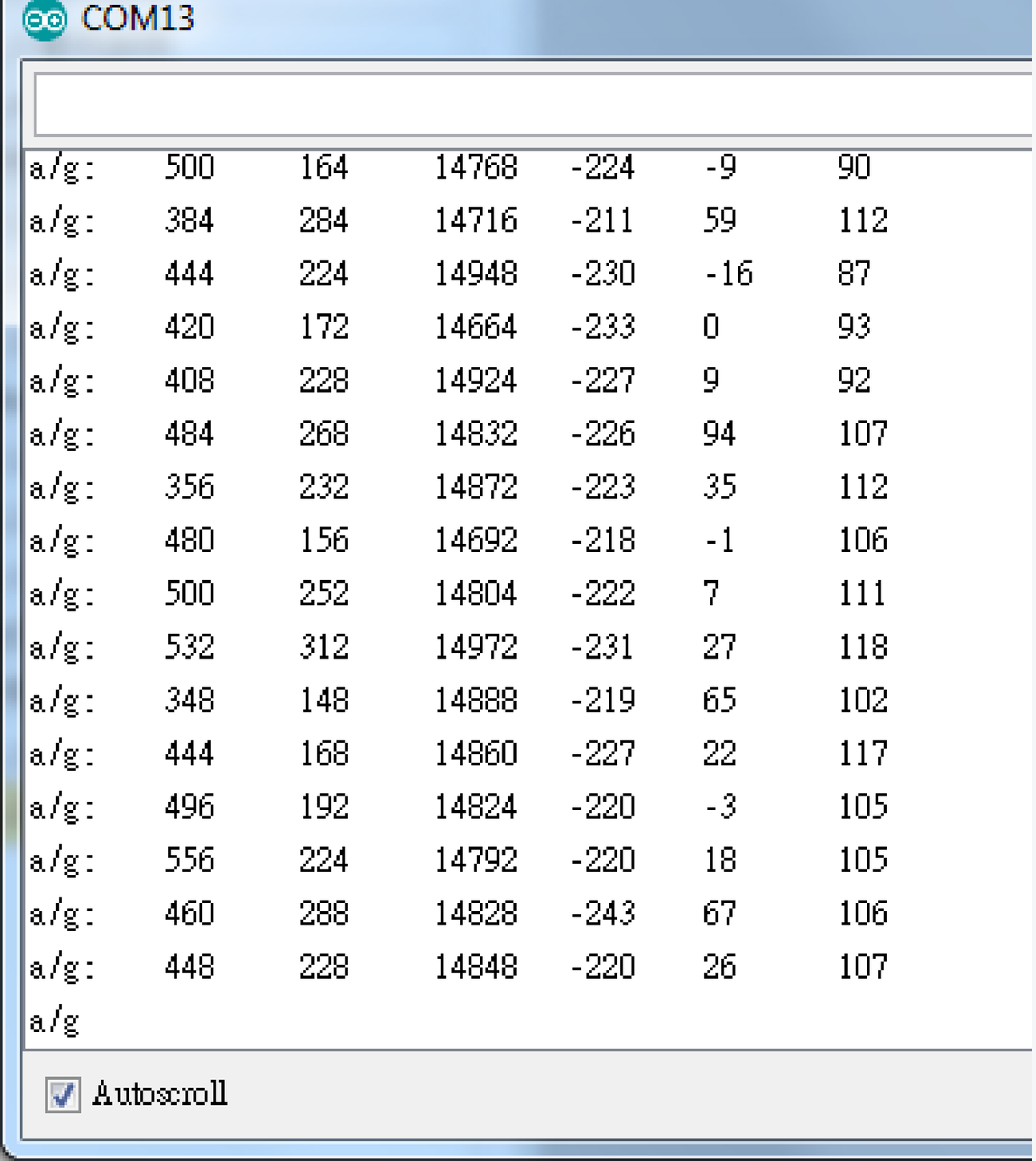


ASSEMBLY PROTOTYPE BUILD

SOFTWARE

After setting everything up and mounting the robot together, it's time for the code part of the robot.

Just like most sensors, 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.



SOFTWARE - MPU CALIBRATION

Using the calibration program from the **I2Cdev library forum** we start by placing the **MPU6050** module in a flat and level position and start the program. The program will make an average of a few hundred readings and display the offsets required to remove zero error.

```
Serial.println(F("Initializing DM
devStatus = mpu.dmpInitialize();

// supply your own gyro offsets h
mpu.setXGyroOffset(15);
mpu.setYGyroOffset(9);
mpu.setZGyroOffset(-5);
mpu.setZAccelOffset(983); // 1688

// make sure it worked (returns 0
if (devStatus == 0)
{
    // turn on the DMP, now that
    Serial.println(F("Enabling DM
    mpu.setDMPEnabled(true);

    // enable Arduino interrupt d
```


SOFTWARE - OFFSETS

We plug the **X Y Z Gyroscope offsets** we got from the calibration program into the MPU offset functions into our main balancing code.

SOFTWARE - PID SETUP

```
//PID
double originalSetpoint = 176.9;
double setpoint = originalSetpoint;
double movingAngleOffset = 0.1;
double input, output;
int moveState=0; //0 = balance; 1 = back; 2 = f
double Kp = 100; // First adjustment
double Kd = 1.2; // Second adjustment
double Ki = 750; // Third adjustment
PID pid(&input, &output, &setpoint, Kp, Ki, Kd,

double motorSpeedFactorLeft = 0.7;
double motorSpeedFactorRight = 0.7;
//MOTOR CONTROLLER
int ENA = 10;
int IN1 = 6;
```

Serial Monitor

The **PID** 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. Tuning of the controller can be done by changing the values of the (**Kp**), (**Ki**) and (**Kd**) variables at the top of the code.

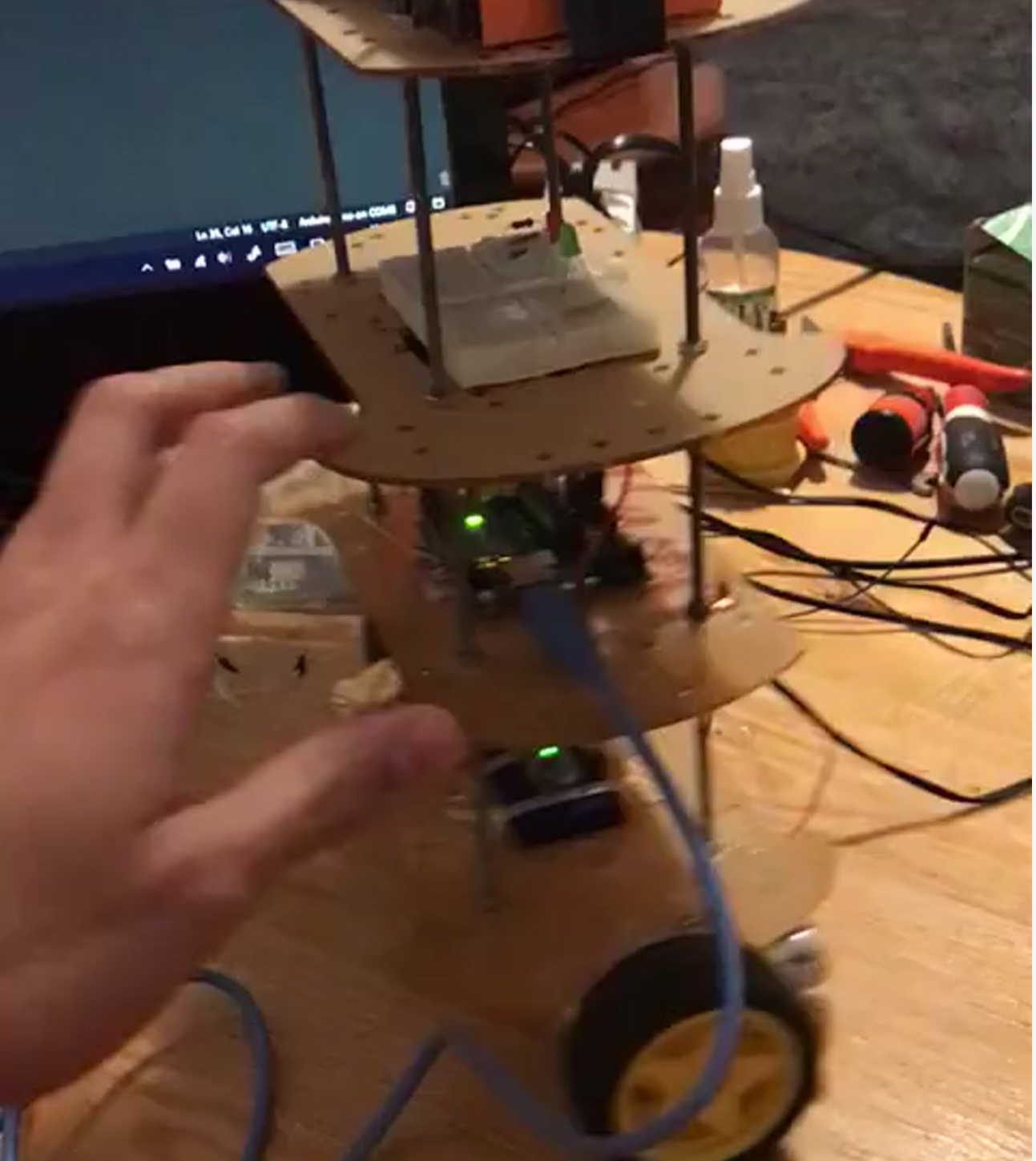
SOFTWARE - PID TUNING

```
//PID
double originalSetpoint = 176.9;
double setpoint = originalSetpoint;
double movingAngleOffset = 0.1;
double input, output;
int moveState=0; //0 = balance; 1 = back; 2 = f
double Kp = 100; // First adjustment
double Kd = 1.2; // Second adjustment
double Ki = 750; // Third adjustment
PID pid(&input, &output, &setpoint, Kp, Ki, Kd,

double motorSpeedFactorLeft = 0.7;
double motorSpeedFactorRight = 0.7;
//MOTOR CONTROLLER
int ENA = 10;
int IN1 = 6;
```

Serial Monitor

Although the **PID** controller code is complete, suitable **PID** constants still need to be found to tune the controller for our 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.



SOFTWARE - PID TUNING (KP)

0. **We set all PID constants to zero**.

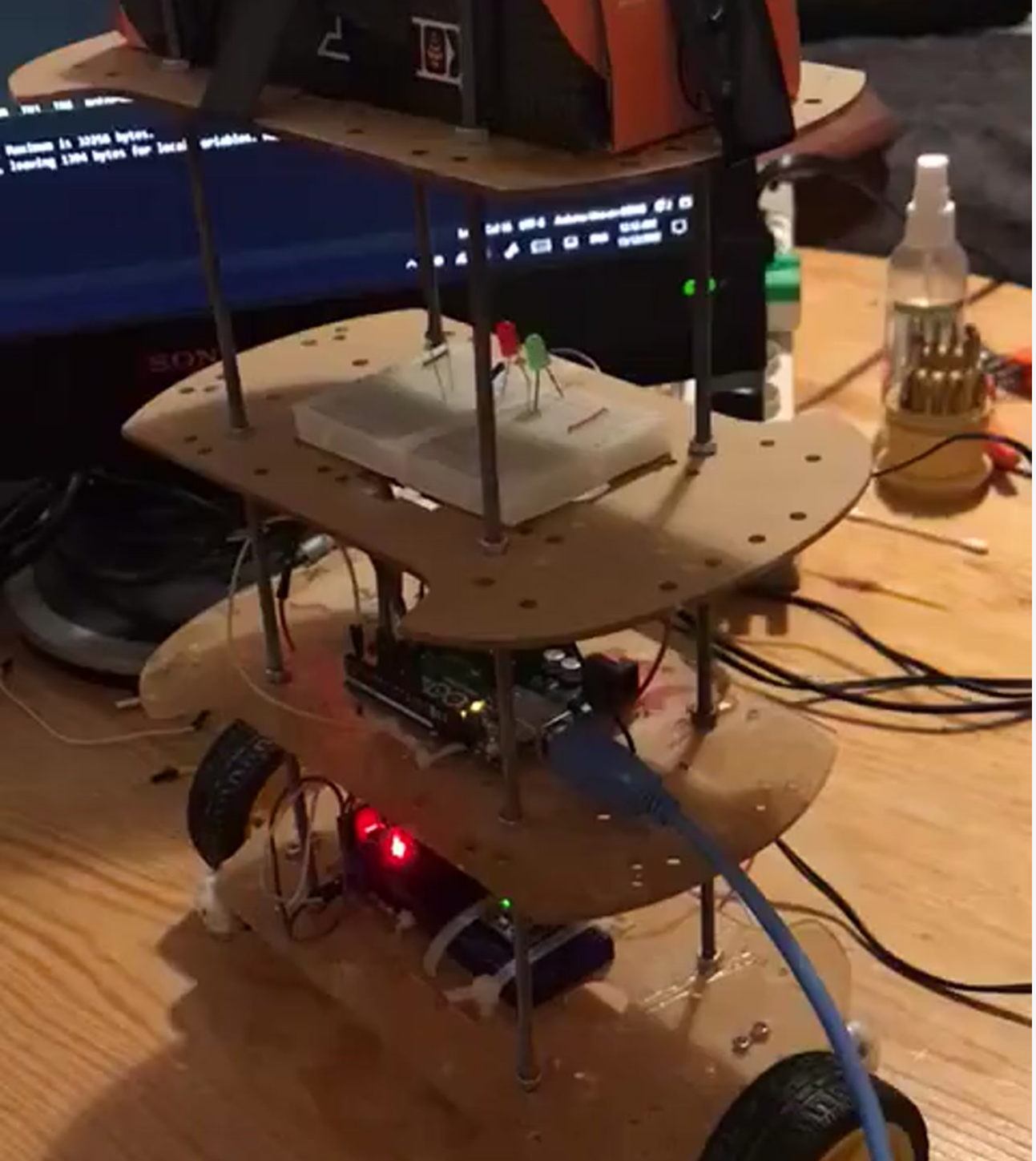
1. **Slowly increase the P-constant value.** we increase the P-constant until the robot responds **quickly** to any **tilting**, and then *just* makes the robot overshoot in the other direction.

```
Kp = 100; // First adjustment
Kd = 1.2; // Second adjustment
Ki = 750; // Third adjustment
```

SOFTWARE - PID TUNING (KD)

2. **Now increase the D-constant.** This component is a bit tricky to get right. It should be kept relatively low, as it can accumulate errors very quickly.

```
Kp = 100; // First adjustment
Kd = 1.2; // Second adjustment
Ki = 750; // Third adjustment
```



SOFTWARE - PID TUNING (KI)

3. **Raise the I-constant. A lot.** This components works against any motion, so it helps to dampen any oscillations and reduce overshooting.

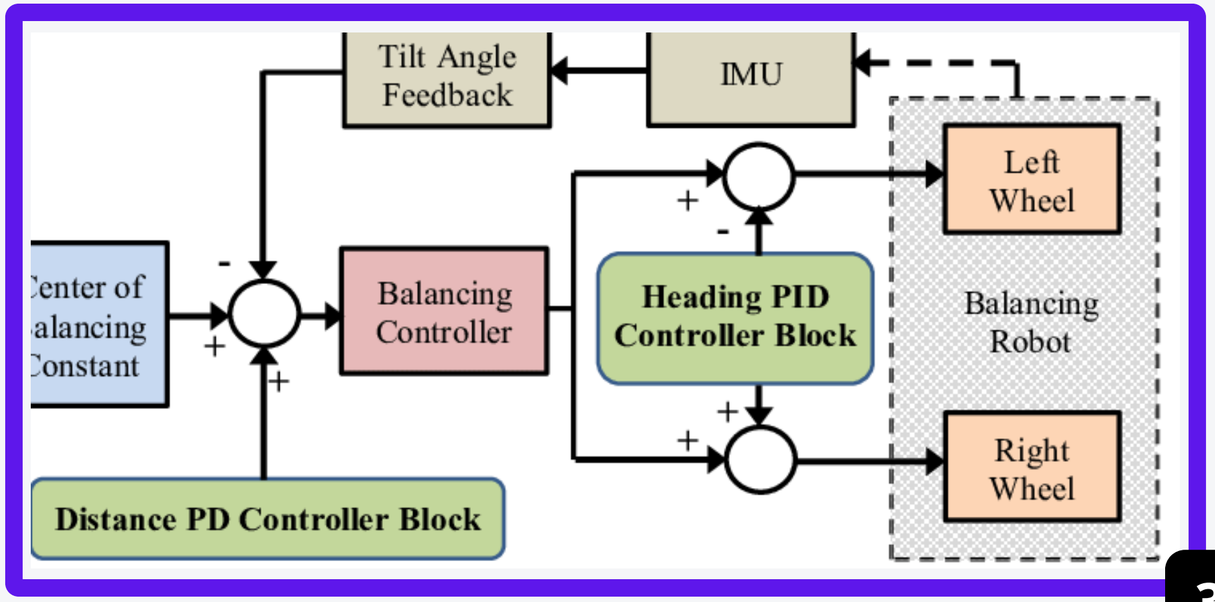
```
Kp = 100; // First adjustment
Kd = 1.2; // Second adjustment
Ki = 750; // Third adjustment
```

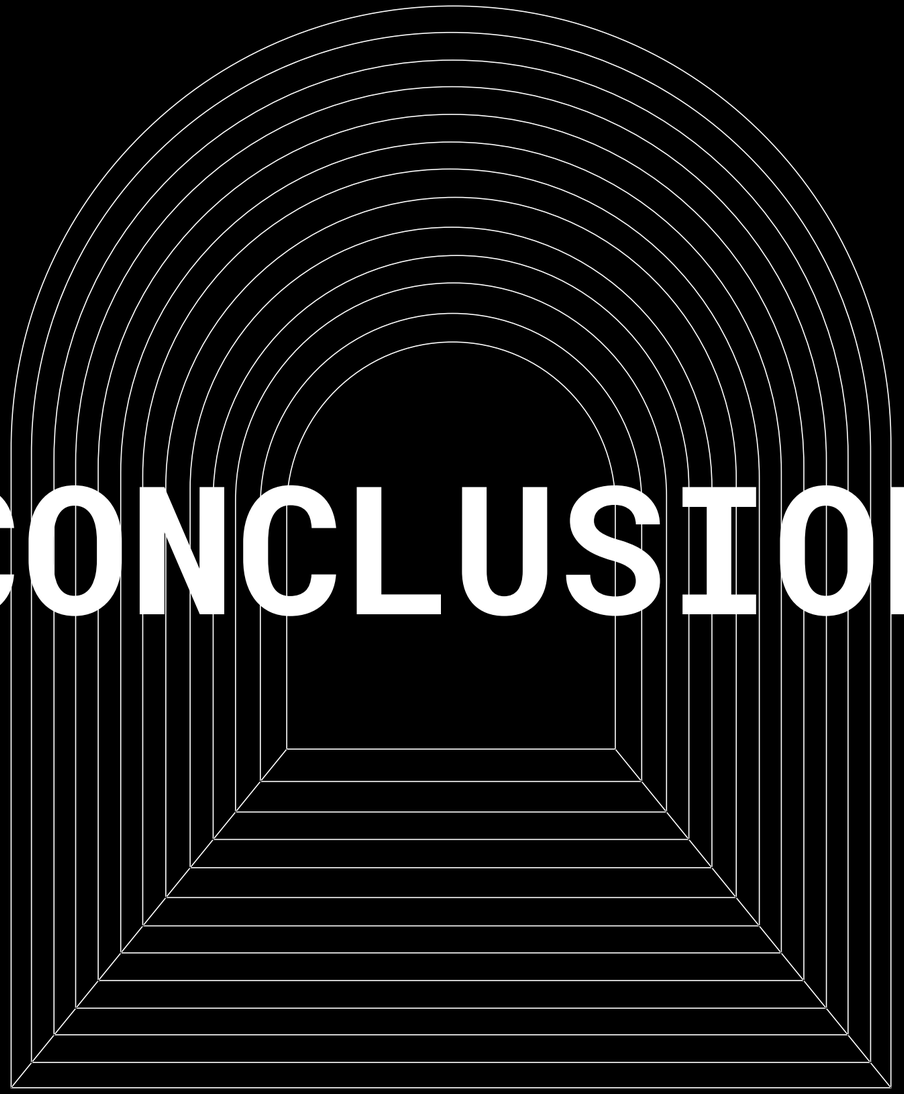
SOFTWARE - WHAT IS IT DOING?

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).

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.

```
pid.Compute();
motorController.move(output, MIN_ABS_SPEED);
```





CONCLUSION

SELF BALANCING ROBOT SHOWCASE

CONCLUSION & PERSPECTIVES

As performance limits in mobile robotics are increasing, dynamic effects are becoming ever more important. Self Balancing System could balance in limited conditions without much complex circuits. One of the major limitations was the sensing of balance.The time taken to attain the stable position is done within limited time and accuracy after the load is being placed. Due to time constraints we could not add remote controlling and obstacle avoidance. To better enhance our SBR, we could add these features, use a better and faster Micro-Processor, our motors and wheels were a also huge bottleneck because the robot can't catch itself once it gained momentum (stepper motors are more ideal).