POLITECNICO DI TORINO

Department of CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Mechatronic Engineering

a.a. 2022/2023

Robotics

Prof. A. Rizzo

T.A.: G. Galati and M. Rinaldi

Self-balancing robot

Aisa Filippo – 315421

Musio Maria Grazia – 315756

Russo Daniele – 308291

Ursi Mariateresa – 315428

Valenziano Gaia – 309918

Contents

Abstract

This project was aimed to design, simulate, build and study a self-balancing robot.
It is a two wheeled machine which stays in an unstable equilibrium and it returns to it after the application of a disturbance.

The research development started with a first theoretical and simplified model of the system.
Then, a detailed and precise CAD model of the robot was created in SolidWorks.
Many simulations were performed: at first in Simscape to visualise the behaviour of the system; later, as soon as the performances were satisfactory, the robot was simulated in Gazebo and finally it was physically built.

The main trouble when designing a robot of this kind was the control system.
In this project a PID controller was chosen to implement a simple but effective and efficient control algorithm. The prototype was built with an IMU element to include a state estimation, which is important for the control system itself.

Introduction

Background

The working principle of a two wheeled self-balancing robot is similar to that of the human body, which can be seen as an inverted pendulum balancing the whole upper body around the ankles.
Although the self-balancing robots have advantages with respect to other robotic configurations, e.g. the mechanical simplicity, an accurate control could be very hard to be successfully achieved.

In order to maintain a balance, a gyroscope and accelerometer in form of Inertial Measurement Unit (IMU) is used. The latter provides the angle of fall of the robot that is then used by the PID controller to calculate the motor input required to track the constant reference angle corresponding to the unstable equilibrium position.
The signal is sent as PWM to the motors, increasing and decreasing the speed of the wheels and changing their directions when needed.

Inverted pendulum applications are plenty: in the recent decade, segways making use of bodily movements for steering have emerged on the market.
All these applications share with each other the position of the centre of mass above their pivot points, thus the requirement of active control for balancing.

Purpose

The focus of this project is to build a simple working prototype of a two wheeled self-balancing robot, starting from the design of its mechanical structure and finishing with the code implementation that enables it to be balanced, at the end.

Method

At the beginning preliminary research was conducted to correctly understand the system dynamics and the electronics behind. Thus, the dynamic equations were derived.
These equations were used to design a PID controller for stabilising the robot.
Once the model was precisely created in SolidWorks, it was exported and integrated in simulation platforms to emulate the real robot's behaviour together with the developed control system before the effective construction.

Theoretical approach

System modelling

A self-balancing robot is typically modelled as an inverted pendulum.
In this project an even simple model was used at the beginning: an inverted pendulum which was balanced in the unstable equilibrium point through a PID controller.

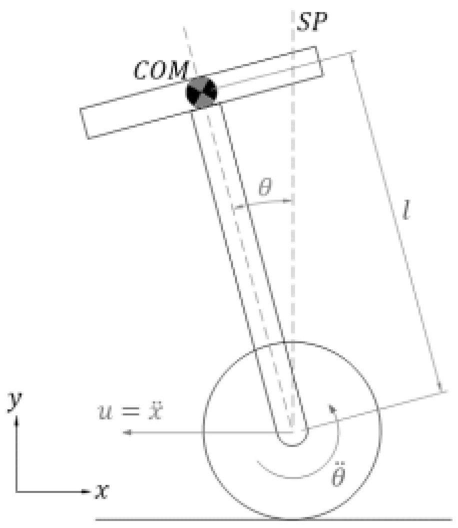


Figure 1 Dynamics of a Self-Balancing Robot

The equation of motion of the self-balancing robot was derived using Lagrange's equations (inverted pendulum), by analysing the dynamics of the robot:

$$l\ddot{\theta} - gsin\theta = \ddot{x}cos\theta$$

Here, l is the distance of the center of mass of the robot from the axis of rotation (i.e. wheel centre), g is the acceleration due to gravity, θ is the pitch angle, $\ddot{\theta}$ is the angular acceleration and $\ddot{x}$ is the linear acceleration (fed as control input u).

Then, the dynamics of the model was written as:

$$\ddot{\theta} = \frac{g}{l}sin\theta + \frac{u}{l}cos\theta$$

The state of the system was represented:

$$\begin{cases} x_1 = \theta \\ x_2 = \dot{\theta} \end{cases}; \; x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}$$

Measuring the pitch angle θ of the robot:

$$output: y = \theta$$

The control signal influenced the linear acceleration of the base of the robot, therefore:

$$control\ signal: u = \ddot{x}$$

Hence, we obtained the open-loop system dynamics:

$$\ddot{x} = f(x,u) = \begin{bmatrix}\dot{\theta} \\ \ddot{\theta}\end{bmatrix} = \begin{bmatrix} x_2 \\ \frac{g}{l} sin\theta + \frac{u}{l} cos\theta \end{bmatrix}; y = h(x) = x_1.$$

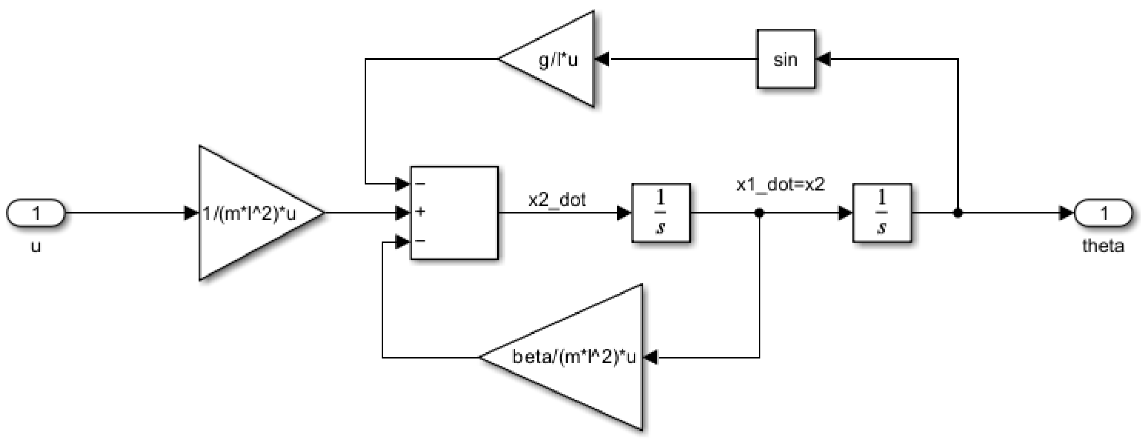


Figure 2 Dynamical system in Simulink

Control system

As previously discussed, the control system was based on a PID controller.
It was chosen among many options, which also included LQ and MPC controllers, thanks to its simplicity and the ability to stabilise a system with a high unstable dynamic, maintaining the vertical position of the robot.

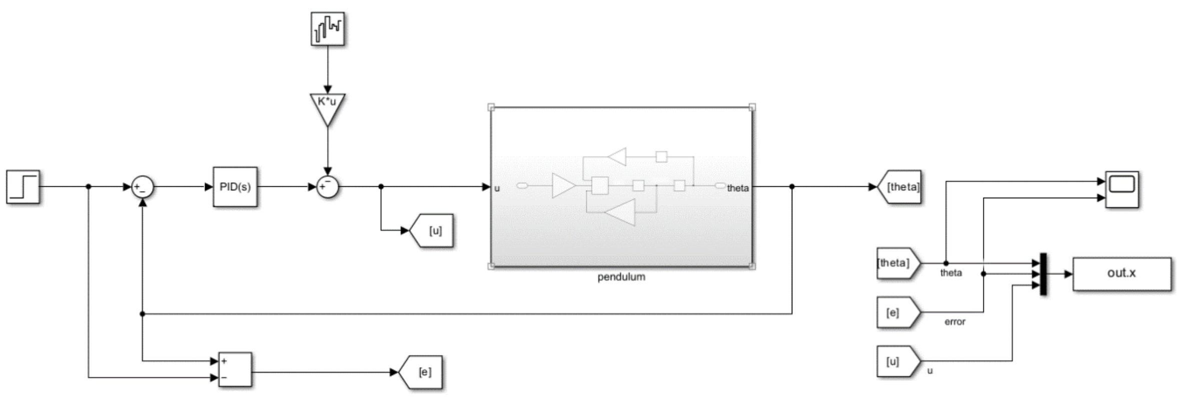


Figure 3 Closed-loop with PID controller

The primary objective of the PID controller is to maintain the value of the pitch angle θ of the robot close to the setpoint value, chosen to be 180° based on the IMU position on the robot, which is fed to the controller as a reference signal.
The error $e(t)$ is computed through negative feedback, subtracting the feedback measurement from the setpoint. Thus, it is fed as input to the PID controller.
On the basis of *Ki*, *Kp*, and *Kd* controller gains, the input of the plant $u(t)$ was computed:

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

The three control branches are weighted thanks to their gains and then summed together in order to achieve the required input of the plant.

The following explanation clarify the method used to tune those gains:

- *Kp*, the main task of the *proportional* part is to reduce the rise time, as well as the steady state error, but it increases the overshoot and generates an offset error;
- *Kd*, the *derivative* term increases the damping of the system by reducing the overshoot and the settling time, useful to smooth the behaviour of the system;
- *Ki*, the *integral* term is used to completely eliminate the steady state error, so it is useful when an offset error is generated by the system.

To summarise, the proportional term corrects the current errors, the integral term compensates the past errors, the derivative term avoids the future errors predicting them.

Tuning the PID controller gains is not done through any formula, but it is an empirical procedure based on the system behaviour, following a conventional order: proportional, integral and derivative, respectively. Hence, the integral removes the offset error generated by the proportional and the derivative smooths the system behaviour.
To reach a correct stabilisation, the tuning was done with the trial and error method on the Simscape simulation and then it was applied to the real robot.

Simulation

At first the model of the robot was realized in SolidWorks and then exported through a plugin to use it in Matlab and perform the simulation through Simscape.
Moreover, the same CAD model was exported in URDF format through another plugin to be uploaded in ROS and simulated in Gazebo.

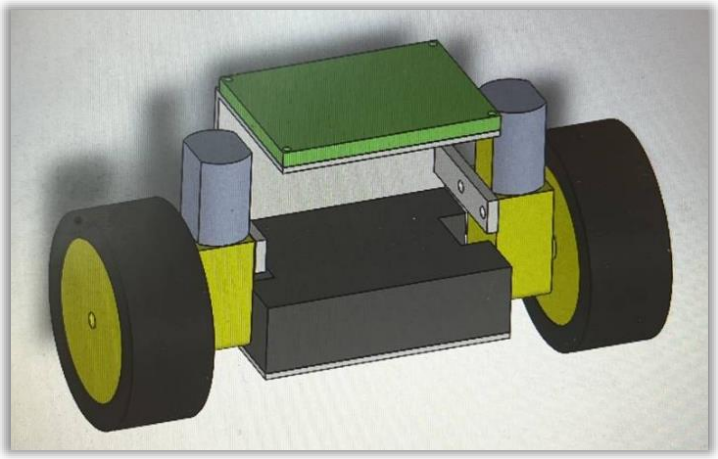


Figure 4 Self-balancing robot model in SolidWorks

Simulink and Simscape

Simscape is part of the larger MATLAB and Simulink software environment and uses a physical modelling approach to represent and simulate systems. It allows users to model and simulate a wide range of physical systems, including mechanical, electrical, hydraulic and thermal systems. It provides a library of pre-built components, such as motors, sensors, and actuators, that can be easily combined to create complex systems.
The simulation can be run and analysed using Simulink tools, such as scopes and data logging. Simscape also provides tools for analysing system performance, such as frequency response plots and transient analysis.

Our self-balancing robot was simulated in Simscape with the 3D CAD model of the real robot. Taking inspiration from other projects, the simulation was performed in Simscape by integrating the robot file obtained through the Simscape Multibody Link Simulink blocks to implement the control.

Thus, a PID controller was inserted and properly tuned. After, an initial perturbation was inserted to check the stability of the robot.

At the beginning, small arbitrary values were set as PID parameters to observe the behaviour in response to an applied disturbance. Then, step by step we tuned them, until the convergence to the equilibrium was achieved. The final values are shown in Figure 7.

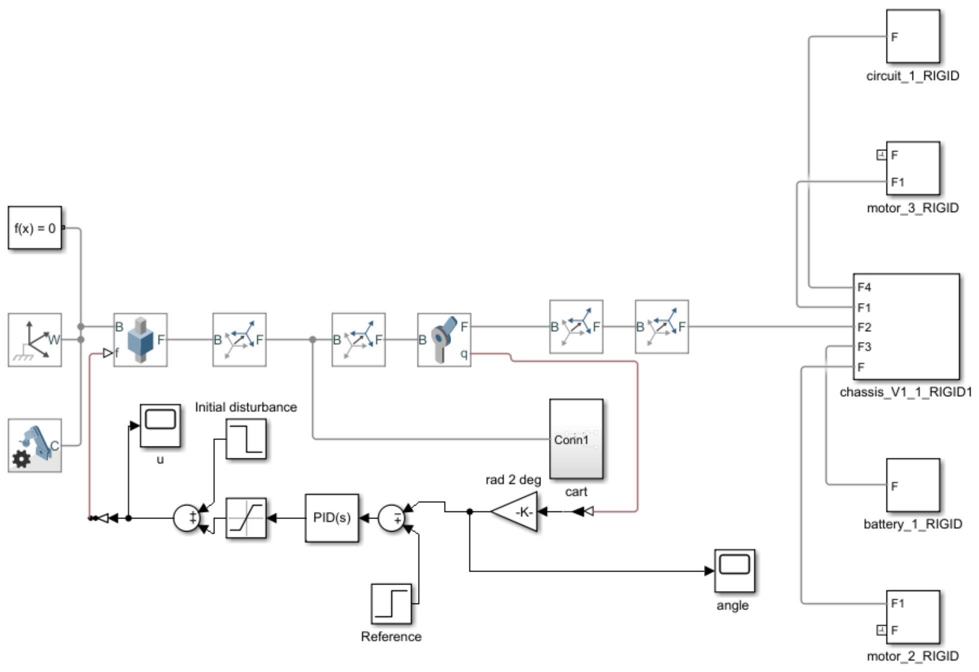


Figure 6 Block diagram in Simscape

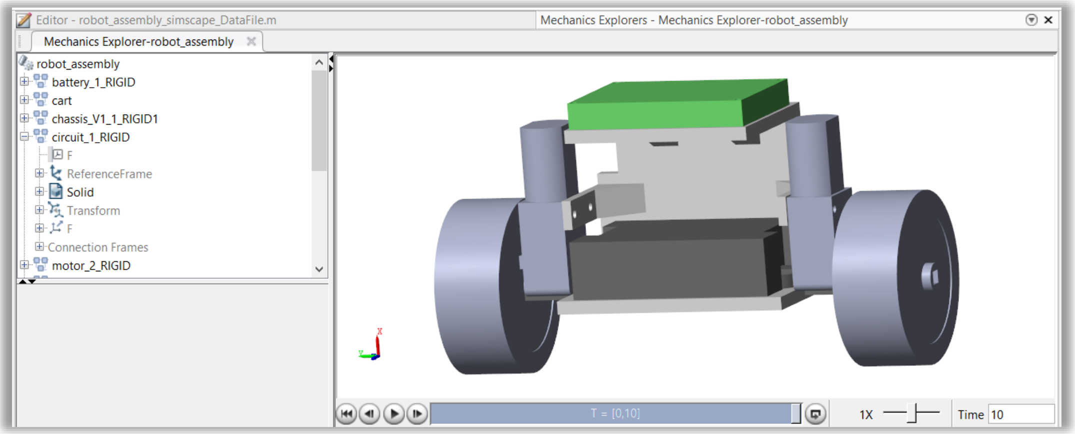
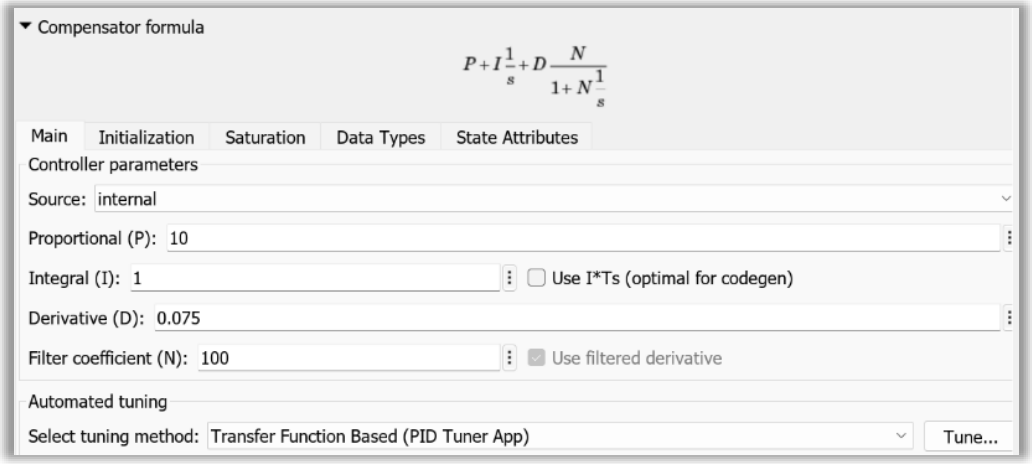


Figure 5 Self-balancing model in Simscape



Compensator formula

$$P + I\frac{1}{s} + D\frac{N}{1 + N\frac{1}{s}}$$

Main | Initialization | Saturation | Data Types | State Attributes

Controller parameters

Source: internal

Proportional (P): 10

Integral (I): 1 ☐ Use I*Ts (optimal for codegen)

Derivative (D): 0.075

Filter coefficient (N): 100 ☑ Use filtered derivative

Automated tuning

Select tuning method: Transfer Function Based (PID Tuner App) Tune...

Figure 7 PID controller final gains

ROS and Gazebo

Gazebo is an open-source 3D simulator with the ability to simulate robot groups, sensors and various objects. Different types of objects can be loaded into the simulator, from simple shapes such as cubes and spheres to complex models as buildings or animals. Each object has its own characteristics: mass, speed, friction and numerous other physical and visualization properties in order to make the simulation as realistic as possible.

There is also a community-run robot database that everyone can use and modify as they please. Gazebo can also generate data from different sensors: laser rangefinder, 2D and RGB-D cameras, contact sensors, Inertial Measurement Units (IMU) and Radio Frequency IDentification (RFID). By connecting one or more sensors to a robot model, it's possible to change the robot's actions based on the data generated by the sensors.
For this reason, it is possible to implement custom plugins that manage the data collected by the sensors and that control any aspect of the robot.

Folders organization

A new workspace called *barcollo* was created. It contains three folders: besides *build* and *devel*, one can focus particularly on the 3rd one that is the *src* folder. The latter contains the package of interest called *robot_assembly*, which in turn presents other folders among which the most important and essential for a clear comprehension are mentioned below.

URDF structure (*urdf* folder)

URDF (Unified Robot Description Format) is an XML format used in ROS for representing robot models. The description of the model consists of defining two main sets, one of links and the other referred to the joints. A link describes the characteristics of a rigid body with inertia, while a joint the kinematics and dynamics of the robot, allowing the relative motion of links.

Our self-balancing robot's URDF was directly exported from SolidWorks.
It is mainly composed by:

- Links: *base_link, RightWheelLink, LeftWheelLink*
- Joints: *RightWheelJoint, LeftWheelJoint* - representative of both left and right robot's wheels actuated by two motors.

The two wheels were modelled as continuous joints rotating about a pre-determined axis: the z-axes were defined in opposite direction with each other due to the symmetry between the two wheels, without specifying any upper and lower limit.

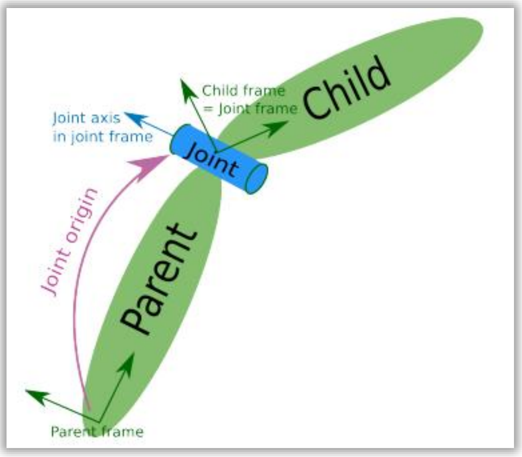


Figure 8 Joint-Link relationship in ROS

Moreover, two plugins were added to the URDF file, to enhance the capabilities of the URDF models by enabling dynamic behaviour, sensor simulation, ROS integration, and custom functionality. They bridge the gap between the static robot description in URDF and the dynamic simulation environment in Gazebo, allowing a more realistic and interactive simulations.

Particularly, in this project *imu_plugin* and *gazebo_ros_control* were used:

- "*imu_plugin"* - to simulate an Inertial Measurement Unit (IMU) sensor. It provides a way to incorporate sensor models and simulate sensor data in Gazebo, including cameras, depth sensors, lidars, inertial measurement units (IMUs) and so on, so forth. Sensor plugin generates realistic sensor data based on the simulated environment and robot interactions, allowing to develop and test the perception algorithms using the simulated sensor inputs.
- "*gazebo_ros_control*" - responsible for integrating the ROS control system with Gazebo. It enables the communication between the simulated robot and the ROS environment. This plugin can publish and subscribe to ROS topics, providing a way to exchange data between Gazebo and other ROS nodes.

To define the mechanical relationships, control interfaces and effort/torque simulation between actuators and joints, transmissions were added in URDF. They enable accurate motion simulation, control behaviour and consistency between the robot's description and its visual representation in Gazebo, again facilitating realistic and reliable robot simulations.

Two transmissions were used, *left_wheel_trans* and *right_wheel_trans,* defining simple transmissions between joints and actuators, the *motor _left* and *motor _right*, respectively.

Then, three gazebo *reference* blocks were included, one for each link, just to set the colour and the friction coefficients for the two wheels and the base_link.

.yaml file (*config* folder)

The PID gains and controller settings have been saved in the *controller.yaml* file that gets loaded to the ROS Parameter Server via the launch file. It allowed to set up two velocity controllers for both the left and right wheel joints of the robot in simulation. These controllers use PID control to regulate the joint velocities based on the desired values and the feedback received from the joint_state_controller.

By adjusting the PID gains *Kp*, *Ki* and *Kd,* the controller's performance could be fine-tuned, ideally achieving stable and accurate velocity control.

.launch files (*launch* folder)

The *gazebo.launch* and *istagger_launch.launch* files set up the Gazebo simulation environment for the whole *robot_assembly* package including the robot controllers.
In particular, they contain the reference for the URDF model file, the loading functionality to specify the needed parameters inside the *controller.yaml* file and they spawn the robot both in Rviz and Gazebo simulation environments.

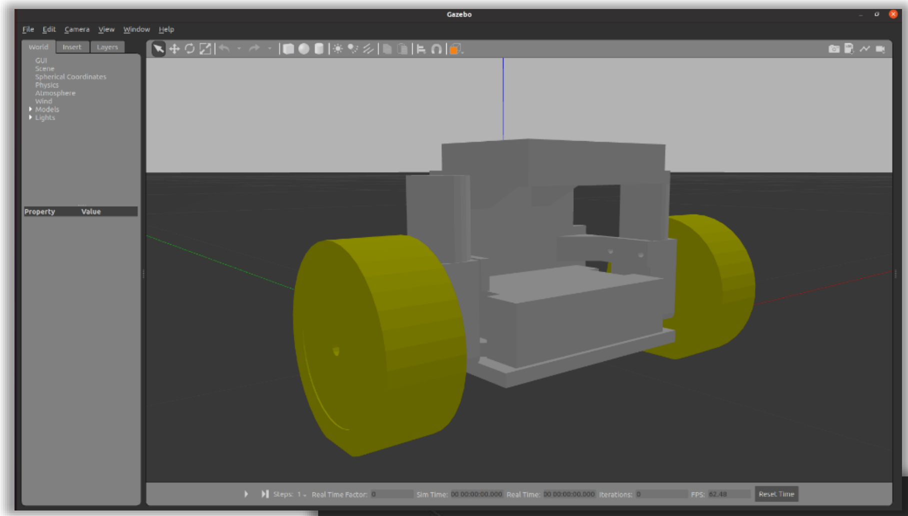


Figure 9 Self-balancing robot model in Gazebo

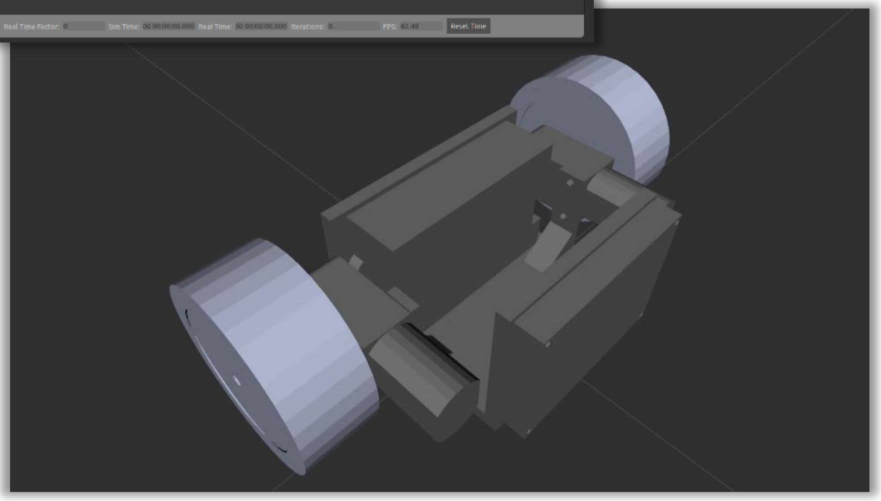


Figure 10 Self-balancing robot model in Rviz

ROS node: Publisher/Subscriber (*src* folder)

To implement the required PID controller a node was written in Python: *pid_controller.py*:

- *Two* publishers, publish velocity commands to the robot's right and left wheels;
- A function *imu_elaboration* is called when the IMU sensor data is received: it extracts the quaternion orientation from the IMU message and passes it to *quat2pitch* function to obtain the pitch angle. The latter is then used to calculate the velocity correction using a PI controller, which finally is returned as the output;
- *PidGenerator* class represents a PID controller, initialised with the three PID gains *Kp*, *Ki* and *Kd*. This class provides a method *getCorrection* to compute the velocity correction based on the current error and the target value. The *tune* method is included to dynamically update the PID gains;
- A *callback* function for the IMU subscriber receives the IMU data and passes it to *imu_elaboration* to obtain the velocity correction. The latter is then published to the right and left wheel velocity controllers;
- *PidController* is the main function of the script: it initialises the ROS node, sets up the Subscriber for IMU data and enters the ROS spin loop to process incoming messages.

Simulation in Gazebo: comments and results

The simulation was successfully performed. It's worth to notice that the PID gains set for the simulation are the same as the ones used for controlling the robot made of PLA, that is 1st prototype constructed: *Kp*=10, *Kd*=5, *Ki*=0.075. Thus, the resulting behaviour of the simulated self-balancing robot in Gazebo was coherent with the real one.

Prototyping and testing

Components

After the modelling and simulation phases, a prototype of the self-balancing robot was built with the following components:

- Arduino Nano
- IMU GY-521
- Motor driver TB6612FNG
- 50x70 PCB stripboard
- DC-DC motors x2
- 7.2 V battery
- Wheels
- Wires
- Chassis

The PCB (Printed Circuit Board) was built and manually soldered in laboratory, while the chassis was 3D printed from the CAD model.

Mechanical systems

The chassis structure was designed after choosing the components. It has two layers: the one below to host the power supply and the other above for the on-board electronics.
The actuators were fixed with screws to support the vertical part of the chassis.
The wheels were connected to the motors' shafts, again using screws.

Electrical system

Power supply

The entire on-board electronics and actuators are powered using a 7.2 V Lithium battery pack. This power supply was chosen in order to be able to supply the whole electronic and actuators for a prolonged duration.

Inertial Measurement Unit (IMU)

The whole self-balancing procedure is based on the measurements of the robot's position, hence the IMU is probably the fundamental electronical element for this project.
The IMU used is a *GY-521*, 6 DOF. It combines an accelerometer and gyroscope and MPU-6050 (Arduino Playground). The criticality of the IMU is that it is very noisy, but the data were considered good enough for our project.

Microcontroller

The on-board electronic is all controlled by an *Arduino Nano*. It was chosen because it is compact and light, besides the open-source nature of the hardware.
Moreover, it is compatible with the IMU thanks to its I2C supported communication and it is easily connectable to the motor driver used. The computational resolution of this microcontroller is enough for the use case.

Motor driver and actuators

A *TB6612FNG* driver was chosen in order to correctly drive the two motors from the power supply and to translate the PWM (Power Width Modulation) signals in speed regulation, capability that was essential for the success of the project. Due to their simplicity and cost-effectiveness, a decision was made to opt for two 2 kg-m DC motors operating at 500 rpm. Other actuators, such as stepper or servo, could give better feedback, but since it was chosen not to control the position of the robot, they were not necessary.

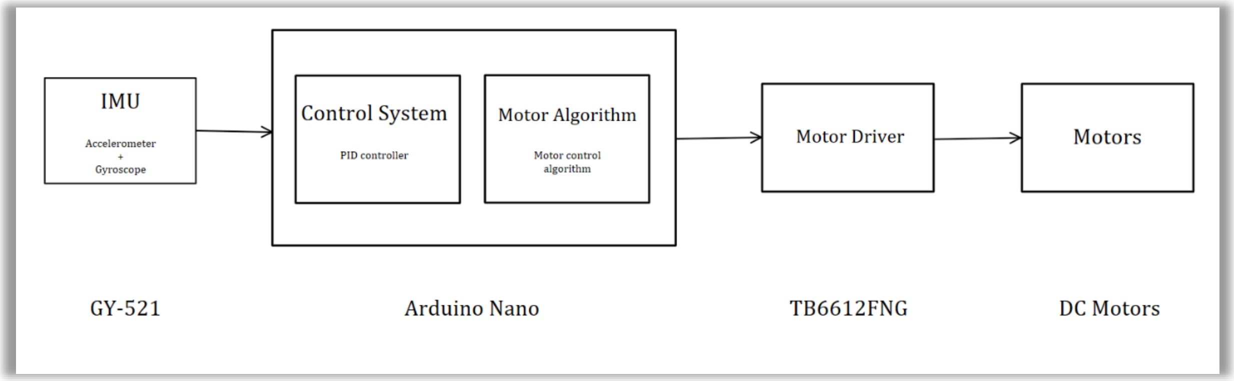


Figure 11 Electronics implementation

The self-balancing principle of the system is accomplished thanks to the help of the IMU, in particular the MPU-6050 sensor. Since the robot has only 1 DOF, three angle are measured by the gyro but only the yaw value is considered and processed by the microcontroller to calculate the PWM signal.

The tilt angle (yaw value) is compared against the required setpoint producing the error in position. The latter is fed to the PID controller, then its output is mapped between [–225, 225] and sent to the motor control algorithm. Thus, it is used to generate the PWM signal to be sent to the driver which translates it in the current, needed to maintain the robot in the upright position.

Files and code implementation

CAD, simulations and the Arduino control program can be found on the GitHub repository of the project together with some instructions on how to replicate the results:

https://github.com/Danru00/SelfBalancingRobot-BarcolloMaNonMollo

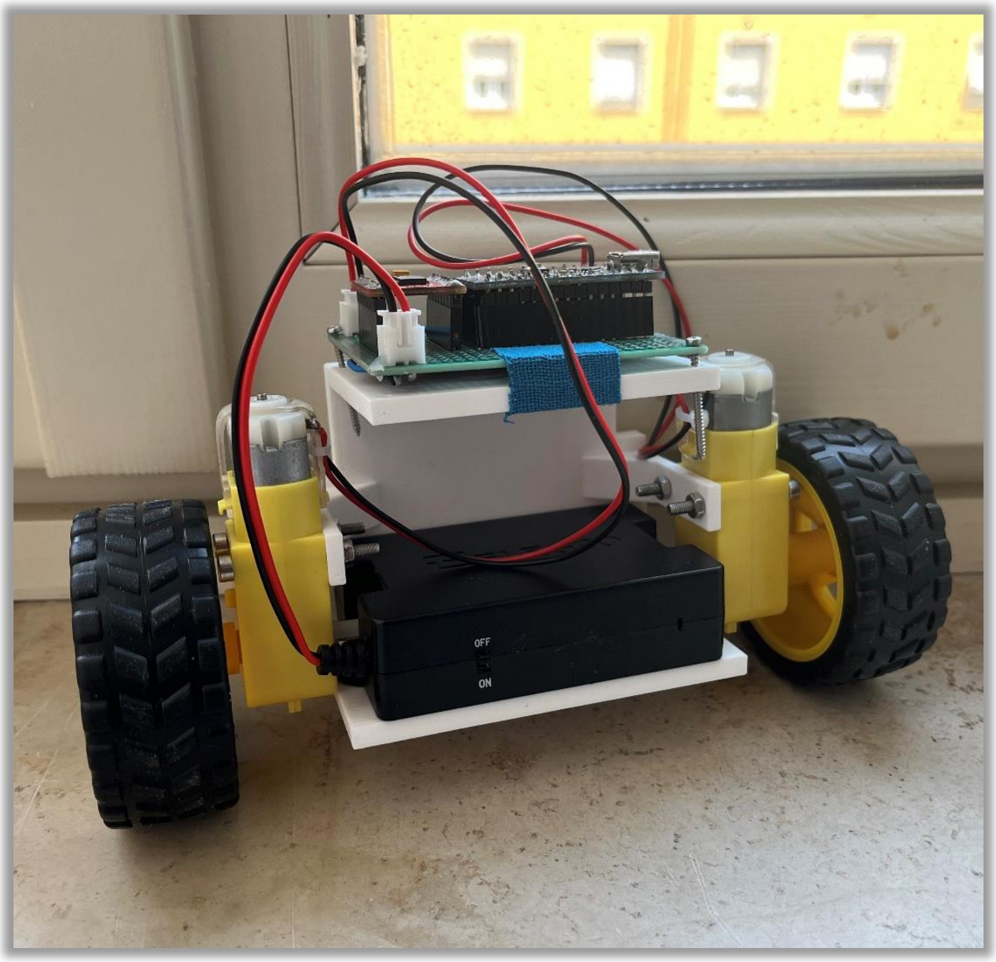


Figure 12 First working prototype of the self-balancing robot

Results and discussion

From the performance analysis of the robot, the most significant parameters were discovered: they are responsible of the behaviour of the whole system.

Controller gains

The controller gains, *Kp*, *Ki*, *Kd* are responsible for the PID response to the error between the setpoint and the feedback position. During the first MATLAB simulation they were chosen using the PID tuner app. Applying these values in both Simscape and Gazebo simulations, the robot was not working as well due to the differences in the systems, so different parameters were chosen by experimentally tuning them.
The same empirical procedure was used to set the parameters for the prototype, verifying the consistency between the values used in Gazebo and those ones in the real robot.

Moment of inertia of the robot

The moment of inertia of the robot changes varying the height and the mass distribution of the robot. In this system, knowing that the angular acceleration is inversely proportional to the moment of inertia of the body, it was chosen to put most of the weight in correspondence of the wheels to reduce the inertia at its minimum.
This decision was taken in order to minimize the effort of the actuators.

IMU Parameters

The calibration of the IMU is fundamental for the system. Since it gets very noisy signals due to its nature, its positioning on the prototype was chosen to avoid disturbances from the wheels and it was also soldered directly on the PCB board to bypass additional vibrations. An initial calibration was needed to find the component's specific offsets to insert into the control program running on the Arduino Nano.

Future implementation

Knowing the imperfections of the first prototype, the next one would have differences in both the structure and components. For instance, more precise actuators can help to get a more accurate behaviour during the stabilising transient. As said above, the IMU signals are very noisy, so a filter in the code (e.g. Kalman filter) could be used to tackle bad components in the signal which cause unprecise wheels moments. Furthermore, a higher chassis could be experimented: the inertia would increase, requiring bigger effort from the motor, but at the same time the disturbances on the IMU should decrease (due to less tilting) and control performance should increase.

Conclusions

The self-balancing robot project was ideated to apply key concepts of mechanics, electronics, robotics and control theory in order to learn how to use these knowledges horizontally.
The performance of the robot is not so perfect because of the limitations linked to the components; however it reached the expected and quite satisfying results.
Surely a second prototype with the described improvements would get more precise results.
It is interesting to note that PID parameters found to be working in Simscape or Gazebo simulations are equal or really close to the parameters actually working on the real prototype.

Future developments on the system could be position control of the robot, autonomous features such as path planning, obstacle avoidance, perimeter following and so on.

References

Inspiration for the Simscape Simulation:
https://it.mathworks.com/matlabcentral/fileexchange/88768-two-wheeled-self-balancing-robot

Solidworks to URDF exporter:
http://wiki.ros.org/sw_urdf_exporter

Simscape Multibody Link plugin:
https://it.mathworks.com/help/smlink/ref/linking-and-unlinking-simmechanics-link-software-with-solidworks.html

Inspiration for the structure and Arduino control algorithm:
https://circuitdigest.com/microcontroller-projects/arduino-based-self-balancing-robot

Inspiration for the Gazebo simulation:
https://github.com/sezan92/self_balancing_robot