

Modelling and controlling a reaction wheel on an inverted pendulum

Michel Vollmuller, 1809572

January 18, 2024

Abstract

In this paper an experimental setup is described in which a pendulum with a reaction wheel on top is balanced by using the torque of an reaction wheel. For the control mechanism a PID controller was used. A digital twin of the system was modelled in Python.

1 Experimental setup

The setup consists of a 3D-printed model see figure 1, with a pendulum on a stand that can be clamped to a table. The pendulum arm is 55 mm long, measured from the center of the pivot point to the center of the reaction wheel. An MPU6050 is attached to the arm, which can read the angle of the pendulum. At the top of the pendulum is a 3-phase 2804 100kv BLDC motor. Connected to this motor is an reaction wheel with a radius of 50 mm. To see the motor's position for better control, there is a magnet in the motor shaft, which an AS5600 sensor can read to find the angle of the motor. To control the entire system, an Arduino Nano is used, and a motor driver is employed to send high power levels to the motor.

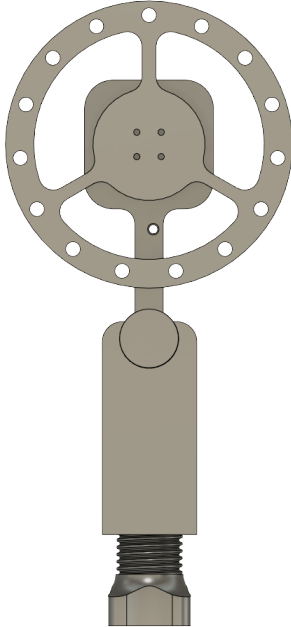


Figure 1: Ontwerp van het 3d model

2 Theory

Simulating an inverted pendulum with an reaction wheel involves modeling the dynamics of both the pendulum and the reaction wheel. In this scenario, the inverted pendulum represents a system where the pendulum is balanced. The reaction wheel is used to control and stabilize the system.

2.1 Pendulum Dynamics:

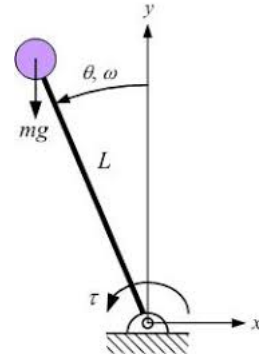


Figure 2: Pendulum Physics

The dynamics of a simple pendulum can be described using the equation:

$$I \cdot \ddot{\theta} = -m \cdot g \cdot L \cdot \sin(\theta)$$

Where:

I is the moment of inertia of the pendulum,
 $\ddot{\theta}$ is the angular acceleration,
 m is the mass of the pendulum bob,
 g is the acceleration due to gravity,
 L is the length of the pendulum,
 θ is the angular displacement.

2.2 Reaction Wheel Dynamics:

The dynamics of the reaction wheel can be described using the equation:

$$I_{rw} \cdot \dot{\omega}_{rw} = \tau_{rw}$$

Where:

I_{rw} is the moment of inertia of the reaction wheel,
 $\dot{\omega}_{rw}$ is the angular velocity of the reaction wheel,
 τ_{rw} is the torque applied by the reaction wheel.

2.3 Torque Interaction:

The torque from the reaction wheel can be used to control the angular acceleration of the pendulum. The net torque affecting the system is the sum of the torque from the reaction wheel and any external torques acting on the pendulum. Assuming no external torques, we can write:

$$\tau_{total} = \tau_{rw}$$

2.4 Combining Equations:

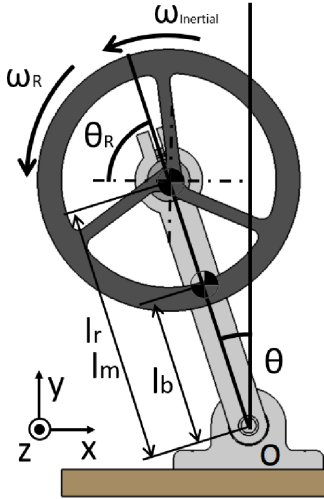


Figure 3: Pendulum + reaction wheel Physics

Combine the equations for the pendulum and reaction wheel dynamics. For simplicity, you may need to make some assumptions, such as neglecting air resistance and friction.

$$I \cdot \ddot{\theta} = -m \cdot g \cdot L \cdot \sin(\theta) + \tau_{rw}$$

2.5 Simulation:

By using these equations, it's possible to simulate the model in software tools such as 20-sim, Python,

or Excel, employing numerical methods like the Euler method to solve the coupled system of differential equations over time. Techniques such as LQR or PID can be utilized to control the model, making it stable and allowing simulation of how the model will respond to external forces.

See appendix B for the implementation in python.

2.6 Control:

In my simulation, I used PID control. This technique first calculates the error (setpoint - current output) of the system, and then a control output is computed based on proportional (Kp), integral (Ki), and derivative (Kd) gains. By tuning these values, the system can be made more or less stable.

The control output of the PID will be used to drive the motor, resulting in torque from the reaction wheel that, in turn, affects the acceleration of the pendulum. This allows the control over the system.

3 Measurements simulation

For the simulation of the system, I referred to an article by Korn, Nopparuj, Paweekorn, and Punyawat, who are students at the Institute of Field Robotics, King Mongkut's University of Technology Thonburi (FIBO). In their work, they explain the swing-up and balancing of a reaction wheel inverted pendulum, providing a simulation that demonstrates the operation with both LQR and PID control. In my own simulation, I focus only on the balancing aspect using PID. however, it's fun and interesting to observe the differences between the FIBO simulation and mine.

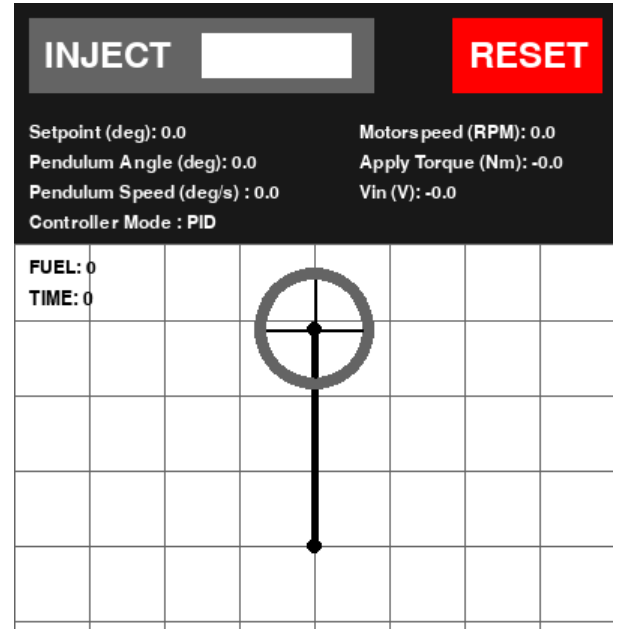


Figure 4: Graphic Simulation

In figure 5 and 6, you can see how both simulations respond to an additional, external force applied to the end of the pendulum. It isn't an fair comparison because the applied forces are not exactly the same. However, it's interesting to note that the observed behavior is nearly identical. Also, it appears that the FIBO simulation is somewhat more accurate. Nevertheless, this is still an interesting comparison, indicating that my simulation seems to be accurate.

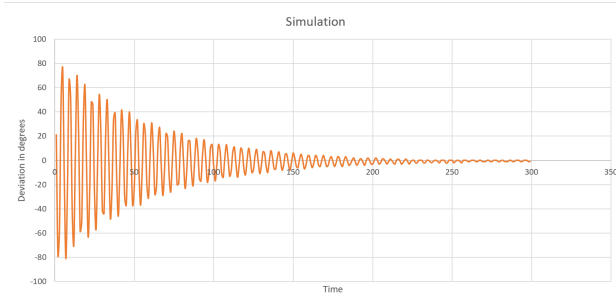


Figure 5: Stepresponse PID controlled simulation Michel.

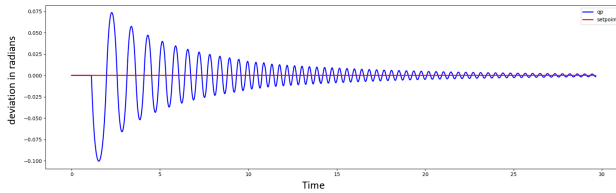


Figure 6: Stepresponse PID controlled simulation FIBO.

The interesting aspect of the FIBO article is that they have also conducted calculations for an LQR simulation. In figure 7, you can observe the simulation where an additional external force is applied to the end of the pendulum. It's clearly visible from the simulation that the system responds much better with LQR control.

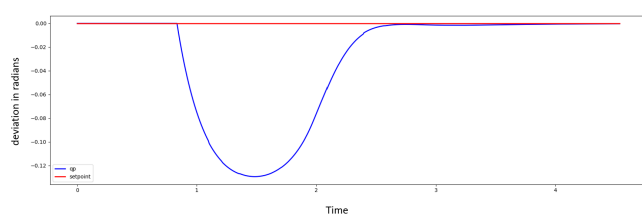


Figure 7: Stepresponse LQR controlled simulation FIBO.

4 Physical setup

At the beginning of this project, the intention was to create a relatively small and compact model of a balancing pendulum using a reaction wheel, allowing for easy transportation and demonstrations in various locations. Unfortunately, this has resulted in nothing more than numerous lessons. Due to the relatively small size of the model, it requires a high level of precision, which is not achievable with the current hardware.

In figure 8, you can see the model in real life. The plan was to attach the model to a surface using a clamp. However, the 3D-printed clamp turned out not to be strong enough to stand stable. Hence the use of tape.



Figure 8: Physical setup.

Firstly, the gyro sensor that measures the angle of the pendulum. In the graph in figure 9, you can see what the sensor measures. The deviation from the actual angle is too significant for this level of precision (up to a maximum of 2.75 degrees). However, an actual reading of the angle is crucial. If the PID controller has a delay of 2 degrees, it becomes extremely difficult to correct this.

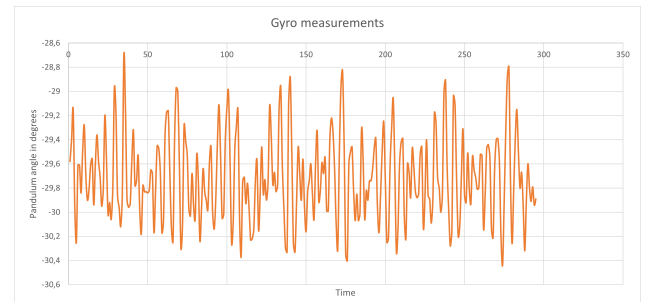


Figure 9: measurements gyro sensor.

What I didn't consider at the beginning of this project, in all my enthusiasm, is that the relatively small model results in a very small margin of error. This makes PID tuning extremely challenging. While in the simulation, you could see the pendulum beautifully oscillating towards its setpoint, achieving the same precision in reality, is proving to be quite difficult. This is partly due to the instability of the 3D-printed model and the sensor.



Figure 10: Model oscillating to its setpoint(0).

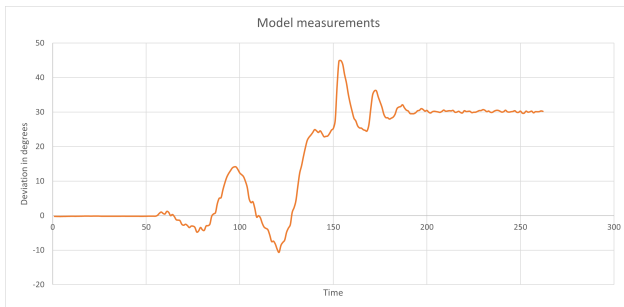


Figure 11: stepresponse of the model.

In the step response in figure 11, you can see how the pendulum initially reaches its setpoint, then experiences an external force, and struggles to balance back to its setpoint. This is likely due to insufficient precision, and the small margin of error prevents the system from responding quickly enough with accurate

measurements. However, what is also noticeable, is that the sensor records very strange values at the end of the falling motion. The pendulum can only fall to an angle of 30 degrees due to a mechanical blockage. In the graph, you can see the sensor registering an angle of almost 45 degrees during the fall, which is impossible. If the sensor frequently produces such incorrect measurements during balancing, I consider it impossible to achieve balance reliably with this sensor.

If we trust the simulation, it should certainly be possible to balance a pendulum with a reaction wheel. We have also seen this in other practical examples on the internet. However, with the challenge of creating a relatively small model on a tight budget and using sensors that are not precise enough, this experiment has shown that achieving this goal is practically not feasible in the way I attempted it.

5 Conclusions and recommendations

Conclusions:

- An experimental setup was created to attempt balancing a pendulum with a reaction wheel using PID control.
- Different controller settings were tried and an optimal setting was found.
- A digital twin was made of the setup in python.

Recommendations:

- Create a larger model so that there is more room for error.
- Using a different sensor, such as the AS5600, for a more precise angle measurement.
- Constructing a sturdier model with less friction at the pivot point.

6 Bibliography

1. **YouTube** How to build self balancing Cube <https://www.youtube.com>
2. **YouTube** Self balancing bike <https://www.youtube.com>
3. **YouTube** Self balancing triangle (reaction wheel) <https://www.youtube.com>
4. **YouTube** A Lesson in Failure: Reaction Wheel PID Tuning <https://www.youtube.com>
5. **PDF** Kinematics Proposal.pdf [Reaction-wheel-inverted-pendulum/blob/main/Kinematics20Proposal.pdf](https://www.youtube.com)
6. **GitHub** GitHub Remigijus <https://github.com/remrc/>

References

A Code used for controlling the setup

```
1 void MPUsetup();
2 void MPUread(float *GyroX, float *GyroY, float *GyroZ);
3 float PID_Controller(float angle, float accel);
4
5 #include <SimpleFOC.h>
6 #include <Adafruit_MPU6050.h>
7 #include <Adafruit_Sensor.h>
8 #include <Wire.h>
9
10 Adafruit_MPU6050 mpu;
11
12 #define PIE 3.14159265358979323846 // define pi
13 #define RAD_TO_DEG 57.295779513082320876798154814105 // 180/pi
14 #define MICROSECONDS_TO_SECONDS 1000000.0 // 1 million
15
16 // global variables
17 float angleX = 0, angleY = 0, angleZ = 0;
18 float GyroX, GyroY, GyroZ;
19 float correctie = 0.0;
20 unsigned long lastTime = 0;
21 float previous_angle = 0;
22
23 // magnetic sensor instance - SPI
24 // MagneticSensorSPI sensor = MagneticSensorSPI(AS5147_SPI, 10);
25 // magnetic sensor instance - I2C
26 MagneticSensorI2C sensor = MagneticSensorI2C(AS5600_I2C);
27 // magnetic sensor instance - analog output
28 // MagneticSensorAnalog sensor = MagneticSensorAnalog(A1, 14, 1020);
29
30 // BLDC motor & driver instance
31 BLDCMotor motor = BLDCMotor(7);
32 BLDCDriver3PWM driver = BLDCDriver3PWM(9, 10, 11);
33 // Stepper motor & driver instance
34 //StepperMotor motor = StepperMotor(50);
35 //StepperDriver4PWM driver = StepperDriver4PWM(9, 5, 10, 6, 8);
36
37 // voltage set point variable
38 // // instantiate the commander
39 // Commander command = Commander(Serial);
40 // void doTarget(char* cmd) { command.scalar(&target_voltage, cmd); }
41
42 void setup() {
43
44     // initialise magnetic sensor hardware
45     sensor.init();
46     // link the motor to the sensor
47     motor.linkSensor(&sensor);
48
49     // power supply voltage
50     driver.voltage_power_supply = 12;
51     driver.init();
52     motor.linkDriver(&driver);
53
54     // aligning voltage
55     motor.voltage_sensor_align = 5;
56     // choose FOC modulation (optional)
57     motor.foc_modulation = FOCModulationType::SpaceVectorPWM;
58     // set motion control loop to be used
59     motor.controller = MotionControlType::torque;
60
61     // use monitoring with serial
62     Serial.begin(115200);
63     // // comment out if not needed
64     // motor.useMonitoring(Serial);
65 }
```

```

66 // initialize motor
67 motor.init();
68 // align sensor and start FOC
69 motor.initFOC();
70
71 MPUsetup();
72 for (int i = 0; i < 10; i++) {
73     MPUread(&GyroX, &GyroY, &GyroZ);
74 }
75
76 if (GyroX < -90){
77     correctie = -29.5-GyroX;
78 }
79 else{
80     correctie = 29.5-GyroX;
81 }
82
83 // // add target command T
84 // command.add('T', doTarget, "target voltage");
85
86 Serial.println(F("Motor ready."));
87 Serial.println(F("Set the target voltage using serial terminal:"));
88 _delay(1000);
89 }
90
91 void loop() {
92     float target_voltage = 0;
93     float acceleratie_p = 0;
94     MPUread(&GyroX, &GyroY, &GyroZ);
95     GyroX+=correctie;
96     acceleratie_p = GyroX-previous_angle;
97
98     target_voltage = PID_Controller(GyroX, acceleratie_p);
99     motor.loopFOC();
100    motor.move(target_voltage);
101
102    // Serial.print("vorige hoek: "); Serial.print(previous_angle);Serial.print(" hoek: "); Serial.
103    // print(GyroX);Serial.print(" acceleratie: "); Serial.println(acceleratie_p);
104    Serial.println(GyroX);
105    previous_angle = GyroX;
106 }
107
108 float PID_Controller(float angle, float accel) {
109     // PID gains for angle
110     float Kp_angle = 1.4; // 1.4
111     float Ki_angle = 0.0; // Add I term
112     float Kd_angle = 3.6; // 3.6
113
114     // PID gains for acceleration
115     float Kp_accel = 0.0;
116     float Ki_accel = 0.0;
117     float Kd_accel = 0.0;
118
119     int error_angle, integral_angle, derivative_angle;
120     int error_accel, integral_accel, derivative_accel;
121
122     float voltage;
123
124     // Static variables to store previous errors
125     static int prev_error_angle = 0;
126     static int prev_error_accel = 0;
127
128     // Calculate error for angle
129     error_angle = 0 - angle;
130     integral_angle += error_angle;
131     derivative_angle = error_angle - prev_error_angle;
132
133     // Calculate error for acceleration
134     error_accel = 0 - accel;
135     integral_accel += error_accel;
136     derivative_accel = error_accel - prev_error_accel;

```

```

136
137 // Calculate control inputs using PID
138 float control_angle = Kp_angle * error_angle + Ki_angle * integral_angle + Kd_angle *
    derivative_angle;
139 float control_accel = Kp_accel * error_accel + Ki_accel * integral_accel + Kd_accel *
    derivative_accel;
140
141 // Combine the control inputs
142 voltage = control_angle + control_accel;
143
144 // Update previous errors for the next iteration
145 prev_error_angle = error_angle;
146 prev_error_accel = error_accel;
147
148 return voltage;
149 }
150
151
152
153 void MPUsetup(){
154     // this function is called in main setup to initialize the MPU6050
155     if (!mpu.begin())
156     {
157         Serial.println(F("Failed to find MPU6050 chip"));
158         while (1)
159         {
160             delay(10);
161         }
162     }
163     Serial.println(F("MPU6050 Found!"));
164
165     // test code
166     mpu.setAccelerometerRange(MPU6050_RANGE_4_G);
167     Serial.print("Accelerometer range set to: ");
168     switch (mpu.getAccelerometerRange())
169     {
170     case MPU6050_RANGE_2_G:
171         Serial.println("+2G");
172         break;
173     case MPU6050_RANGE_4_G:
174         Serial.println("+4G");
175         break;
176     case MPU6050_RANGE_8_G:
177         Serial.println("+8G");
178         break;
179     case MPU6050_RANGE_16_G:
180         Serial.println("+16G");
181         break;
182     }
183     mpu.setGyroRange(MPU6050_RANGE_250_DEG);
184     Serial.print("Gyro range set to: ");
185     switch (mpu.getGyroRange())
186     {
187     case MPU6050_RANGE_250_DEG:
188         Serial.println("+ 250 deg/s");
189         break;
190     case MPU6050_RANGE_500_DEG:
191         Serial.println("+ 500 deg/s");
192         break;
193     case MPU6050_RANGE_1000_DEG:
194         Serial.println("+ 1000 deg/s");
195         break;
196     case MPU6050_RANGE_2000_DEG:
197         Serial.println("+ 2000 deg/s");
198         break;
199     }
200
201     mpu.setFilterBandwidth(MPU6050_BAND_5_HZ);
202     Serial.print("Filter bandwidth set to: ");
203     switch (mpu.getFilterBandwidth())
204     {

```

```

205     case MPU6050_BAND_260_HZ:
206         Serial.println("260 Hz");
207         break;
208     case MPU6050_BAND_184_HZ:
209         Serial.println("184 Hz");
210         break;
211     case MPU6050_BAND_94_HZ:
212         Serial.println("94 Hz");
213         break;
214     case MPU6050_BAND_44_HZ:
215         Serial.println("44 Hz");
216         break;
217     case MPU6050_BAND_21_HZ:
218         Serial.println("21 Hz");
219         break;
220     case MPU6050_BAND_10_HZ:
221         Serial.println("10 Hz");
222         break;
223     case MPU6050_BAND_5_HZ:
224         Serial.println("5 Hz");
225         break;
226 }
227 }
228
229 void MPUread(float *GyroX, float *GyroY, float *GyroZ){
230     // if called by main this function will read IMU data from the MPU6050 and return it
231     sensors_event_t a, g, temp;
232     mpu.getEvent(&a, &g, &temp);
233
234     unsigned long currentTime = micros();
235     float dt = (currentTime - lastTime) / MICROSECONDS_TO_SECONDS; // convert to seconds
236     lastTime = currentTime;
237
238     // calculate angle based on gyro data
239     float gyroAngleX = angleX + g.gyro.x * dt;
240     // float gyroAngleY = angleY + g.gyro.y * dt;
241     // float gyroAngleZ = angleZ + g.gyro.z * dt;
242
243     // calculate angle based on accelerometer data
244     float accelAngleX = atan2(a.acceleration.y, a.acceleration.z);
245     // float accelAngleY = atan2(a.acceleration.x, a.acceleration.z);
246
247     // complementary filter: combine the gyro and accelerometer angles
248     float alpha = 0.5; // weight factor
249     angleX = alpha * gyroAngleX + (1.0 - alpha) * accelAngleX;
250     // angleY = alpha * gyroAngleY + (1.0 - alpha) * accelAngleY;
251     // angleZ = gyroAngleZ; // gyro only on Z axis
252
253     angleX += g.gyro.x * dt;
254     // angleY += g.gyro.y * dt;
255     // angleZ += g.gyro.z * dt;
256
257     // convert angles from radians to degrees
258     *GyroX = -((angleX * RAD_TO_DEG));
259     // *GyroY = angleY * RAD_TO_DEG;
260     // *GyroZ = angleZ * RAD_TO_DEG;
261     *GyroY = 0;
262     *GyroZ = 0;
263     // Serial.print(" GyroX: ");
264     // Serial.print(*GyroX);
265     // Serial.print(" GyroY: ");
266     // Serial.print(*GyroY);
267     // Serial.println("Degrees");
268 }

```

Listing 1: Arduino Nano Code

B Code use for the simulation setup

```
1
2 import pygame
3 import math
4 import numpy as np
5
6 REFRESHRATE = 120 #[Hz]
7 pygame.init()
8 win_width, win_height = 1200, 800
9 win = pygame.display.set_mode((win_width, win_height))
10 pygame.display.set_caption("Reaction Wheel")
11 clock = pygame.time.Clock() # For controlling the frame rate
12 # Font initialization
13 font = pygame.font.Font(None, 20) # You can specify a font file or use None for default font and
    size
14 run = True
15
16 # global physics variables
17 MASSA_REACTIONWHEEL = 0.1 #[kg]
18 RADIUS_REACTIONWHEEL = 0.005 #[m]
19 LENGTE_PENDULUM = 0.1 #[m]
20 MASSA_PENDULUM = 0.2 #[kg]
21 GRAVITATIE_CONSTANTE = 9.81 #[m/s^2]
22
23 Torque_Gravity = 0 #[Nm]
24
25 ALPHA_RACTIONWHEEL = 0 #[rad/s^2] hoekversnelling
26 OMEGA_REACTIONWHEEL = 0 #[rad/s] hoeksnelheid
27 THETA_REACTIONWHEEL = 0 #[rad] hoek
28
29 ALPHA_PENDULUM = 0 #[rad/s^2] hoekversnelling
30 OMEGA_PENDULUM = 0 #[rad/s] hoeksnelheid
31 THETA_PENDULUM = 0 #[rad] hoek
32
33 THETA_PENDULUM = np.pi*-0.5
34 hoek_pendulum = 0
35
36 # Initialize global variables
37 integral = 0
38 prev_error = 0
39
40
41
42 def physics():
43     global ALPHA_PENDULUM, OMEGA_PENDULUM, THETA_PENDULUM, OMEGA_REACTIONWHEEL,
        THETA_REACTIONWHEEL, Torque_Gravity, hoek_pendulum
44
45     # Inertia calculations
46     Inertia_ReactionWheel = (MASSA_REACTIONWHEEL*(RADIUS_REACTIONWHEEL**2))/2
47     Inertia_Pendulum = (1/3)*MASSA_PENDULUM*(LENGTE_PENDULUM**2)
48
49     # Torque CALCULATIONS
50     Torque_ReactionWheel = ALPHA_RACTIONWHEEL * Inertia_ReactionWheel
51     Torque_Gravity = (MASSA_REACTIONWHEEL+MASSA_PENDULUM) * GRAVITATIE_CONSTANTE * np.cos(
        THETA_PENDULUM)
52
53     # Main CalCul
54     #ALPHA_PENDULUM = Torque_ReactionWheel+Torque_Gravity/Inertia_Pendulum+MASSA_REACTIONWHEEL*(
        LENGTE_PENDULUM**2)
55
56     total_torque = Torque_ReactionWheel + Torque_Gravity
57     total_inertia = Inertia_Pendulum + MASSA_REACTIONWHEEL * (LENGTE_PENDULUM**2)
58
59     ALPHA_PENDULUM = total_torque / total_inertia
60
61     #intergrations
62     OMEGA_PENDULUM += ALPHA_PENDULUM * 1/REFRESHRATE
63     THETA_PENDULUM += OMEGA_PENDULUM * 1/REFRESHRATE
64     hoek_pendulum = (THETA_PENDULUM*180/np.pi)+90
65
```

```

66 OMEGA_REACTIONWHEEL += ALPHA_REACTIONWHEEL*1/REFRESHRATE
67 THETA_REACTIONWHEEL += OMEGA_REACTIONWHEEL*1/REFRESHRATE
68
69 if hoek_pendulum > 360:
70     THETA_PENDULUM = np.pi*-0.5
71
72
73
74 def pid_controller(hoek):
75     global integral, prev_error # Assuming these are used elsewhere and need to be shared
76
77     # PID constants
78     Kp = 5.43
79     Ki = 0.0
80     Kd = 12.0
81
82     # Initialize variables
83     error = 0 - hoek # Calculate the error
84     integral += error # Calculate the integral
85
86     # Calculate the derivative term
87     derivative = error - prev_error
88
89     # PID calculation
90     draai = Kp * error + Ki * integral + Kd * derivative # Calculate the control output
91
92     # Update the previous error for the next iteration
93     prev_error = error
94
95     return draai
96
97
98 while run:
99     pygame.time.delay(20)
100
101     for event in pygame.event.get():
102         if event.type == pygame.QUIT:
103             run = False
104
105     keys = pygame.key.get_pressed()
106     if keys[pygame.K_LEFT]:
107         ALPHA_REACTIONWHEEL -= 1
108     elif keys[pygame.K_RIGHT]:
109         ALPHA_REACTIONWHEEL += 1
110     elif keys[pygame.K_SPACE]:
111         # Reset all relevant variables to their initial values
112         ALPHA_REACTIONWHEEL = 0
113         OMEGA_REACTIONWHEEL = 0
114         THETA_REACTIONWHEEL = 0
115         ALPHA_PENDULUM = 0
116         OMEGA_PENDULUM = 0
117         THETA_PENDULUM = np.pi*-.5 # Set to initial angle
118         Torque_Gravity = 0
119         Torque_ReactionWheel = 0
120         Inertia_ReactionWheel = 0
121         Inertia_Pendulum = 0
122
123     ALPHA_REACTIONWHEEL = pid_controller(hoek_pendulum)
124
125     physics()
126
127
128     # Fixed point coordinates (center of rotation)
129     center_x, center_y = (win_width/2), (win_height/2)
130     # Stick length
131     stick_length = 200
132     # Calculate the endpoint of the stick based on the angle
133     end_x = center_x + stick_length * np.cos(THETA_PENDULUM)
134     end_y = center_y + stick_length * np.sin(THETA_PENDULUM)
135
136     text_to_display1 = f"ALPHA_PENDULUM: {ALPHA_PENDULUM}"

```

```

137 text_1= font.render(text_to_display1, True, (255, 0, 0))
138 text_to_display2 = f"OMEGA_PENDULUM: {OMEGA_PENDULUM}"
139 text_2= font.render(text_to_display2, True, (255, 0, 0))
140 text_to_display3 = f"hoek_pendulum: {hoek_pendulum}"
141 text_3= font.render(text_to_display3, True, (255, 0, 0))
142 text_to_display4 = f"ALPHA_REACTIONWHEEL: {ALPHA_REACTIONWHEEL}"
143 text_4= font.render(text_to_display4, True, (255, 0, 0))
144 text_to_display5 = f"OMEGA_REACTIONWHEEL: {OMEGA_REACTIONWHEEL}"
145 text_5= font.render(text_to_display5, True, (255, 0, 0))
146 text_to_display6 = f"THETA_REACTIONWHEEL: {THETA_REACTIONWHEEL}"
147 text_6= font.render(text_to_display6, True, (255, 0, 0))
148 text_to_display7 = f"Torque_Gravity: {Torque_Gravity}"
149 text_7= font.render(text_to_display7, True, (255, 0, 0))
150
151 win.fill((0, 0, 0))
152
153 # Blit the text onto the window surface at (x, y) position
154 win.blit(text_1, (50, 50)) # Adjust position as needed
155 win.blit(text_2, (50, 60)) # Adjust position as needed
156 win.blit(text_3, (50, 70)) # Adjust position as needed
157 win.blit(text_4, (50, 80)) # Adjust position as needed
158 win.blit(text_5, (50, 90)) # Adjust position as needed
159 win.blit(text_6, (50, 100)) # Adjust position as needed
160 win.blit(text_7, (50, 110)) # Adjust position as needed
161
162 pygame.draw.line(win, (0, 255, 0), (0, center_y+25), (win_width, center_y+25), 50)
163 pygame.draw.line(win, (255, 255, 255), (center_x, center_y), (end_x, end_y), 5)
164 pygame.display.update()
165 clock.tick(REFRESHRATE) # Limit frame rate
166
167 pygame.quit()

```

Listing 2: Python simulation